Foreword

Learn web services and Apache Axis2 easily

If you'd like to learn how to create web services (in particular, using Apache Axis2) and make some sense of various standards like SOAP, WSDL, MTOM, WS-Addressing, WS-Security, WS-Policy, XML Encryption and XML Signature, then this book is for you. Why?

• It has a tutorial style that walks you through in a step-by-step manner.
• It is concise. There is no lengthy, abstract description.
• Many diagrams are used to show the flow of processing and high level concepts so that you get a whole picture of what's happening.
• The first 46 pages are freely available on http://www.agileskills2.org. You can judge it yourself.

Unique contents in this book

This book covers the following topics not found in other books on Axis:

• How to work with Axis2 1.3.
• How to use Eclipse Europa (WTP 2.0) with Axis2.
• How to invoke asynchronous operations using WS-Addressing.
• How to encrypt and sign SOAP messages using Rampart.
• How to send user authentication information using Rampart.
• How to send and receive binary files using MTOM.
• How to integrate Axis2 with Spring.

Target audience and prerequisites

This book is suitable for those who would like to learn how to develop web services in Java.

In order to understand what's in the book, you need to know Java and to have edited XML files. However, you do NOT need to know the more advanced XML concepts (e.g., XML schema, XML namespace), servlet, Tomcat or PKI.
Acknowledgments

I'd like to thank:

• The Axis developers for creating Axis.
• The WSS4J developers for creating WSS4J.
• Anne Thomas Manes, an expert in web services, for reviewing the book (first edition).
• Helena Lei for proofreading this book.
• Eugenia Chan Peng U for doing book cover and layout design.
# Table of Contents

Foreword.................................................................................................3  
Learn web services and Apache Axis2 easily........................................3  
Unique contents in this book.................................................................3  
Target audience and prerequisites.......................................................3  
Acknowledgments..................................................................................4  
Chapter 1 Designing the interface for a simple web service.................9  
What's in this chapter?............................................................................10  
Providing cross platform operations across the Internet....................10  
RPC style web service...........................................................................11  
Document style web service.................................................................14  
Determining the operation for a document style web service. ............17  
Port type.................................................................................................18  
Binding.................................................................................................19  
Port.........................................................................................................20  
Target namespace..................................................................................22  
WSDL.....................................................................................................24  
Summary.................................................................................................25  
Chapter 2 Implementing a web service..................................................27  
What's in this chapter?............................................................................28  
Installing Eclipse....................................................................................28  
Installing Axis2.....................................................................................28  
Installing the Axis2 plugin for Eclipse.................................................30  
WSDL file for the web service.............................................................31  
RPC version of the web service............................................................35  
Creating the WSDL file visually..........................................................36  
Validating the WSDL file.......................................................................45  
Generating a service stub.......................................................................46  
Implementing the web service...............................................................52  
Deploying a web service........................................................................53  
Creating a client using a client stub......................................................55  
Undeploying a web service.................................................................58  
Summary.................................................................................................59  
Chapter 3 Optimizing the development environment.........................61  
What's in this chapter?............................................................................62  
Placing the class files into Axis directly..............................................62  
Making changes take effect immediately............................................64
Debugging a web service.................................................................66
Generating code automatically......................................................69
Generating client code automatically............................................75
Summary.....................................................................................76
Chapter 4 Understanding the calling process..............................77
What's in this chapter?..................................................................78
Calling a web service without a client stub...............................78
Seeing the SOAP messages.........................................................79
Summary.....................................................................................83
Chapter 5 Accepting multiple parameters.................................85
What's in this chapter?..................................................................86
Accepting multiple parameters.....................................................86
Interoperability............................................................................93
Summary.....................................................................................93
Chapter 6 Sending and receiving complex data structures.............95
What's in this chapter?..................................................................96
Product query..............................................................................96
Avoiding the type suffix.............................................................105
Sending more data in a message.................................................107
Returning faults..........................................................................108
Using encoded.............................................................................116
Referring to existing XML elements...........................................117
Retrieving WSDL files using HTTP..........................................122
Summary....................................................................................122
Chapter 7 Sending binary files.....................................................123
What's in this chapter?..................................................................124
Providing the image of a product................................................124
Enabling MTOM in the service....................................................129
Interoperability............................................................................129
Summary....................................................................................130
Chapter 8 Invoking lengthy operations........................................131
What's in this chapter?..................................................................132
Providing lengthy operations.......................................................132
Creating the WSDL for business registrations............................135
Creating a new thread for lengthy processing............................139
Creating an asynchronous client.................................................141
Inspecting the WS-Addressing header blocks............................144
Avoiding modifications to the message receiver.........................145
Summary....................................................................................146
Chapter 1
Designing the interface for a simple web service
What's in this chapter?

In this chapter you'll learn how to design the interface for a simple web service.

Providing cross platform operations across the Internet

Suppose that you'd like to provide a service to the public or to some business partners: They can send you two strings and you will concatenate them and return the string. Of course, in the real world you provide a more useful service.

There are several major requirements: First, the users may be using different languages (Java, C# and etc.) and using different platforms (Windows, Linux and etc.). Your service must be accessible by different languages and platforms. Second, they will call your service across the Internet and there may be firewalls in between. Your service must be able to go through firewalls.

Given these requirements, the best solution is to provide a so-called "web service". For example, you may make a web service accessible on the host www.ttdev.com and accessible as /SimpleService (see the diagram below), so the full URL is http://www.ttdev.com/SimpleService. This is called the "endpoint" of the web service. Your web service may support one or more operations. One operation may be named "concat":

However, you hope to provide a globally unique name to each operation so that you can have your "concat" operation while another person may have his
"concat" operation. So, in addition to the name, you may declare that the "concat" name above is in the "namespace" of http://ttdev.com/ss (see the diagram below). A namespace is just like a Java package, but it is not in a dot format like com.ttdev.foo; it is in the format of a URL. So, the full name of the operation will be "concat" in namespace http://ttdev.com/ss. The name "concat" is called the "local name". The full name is called a "QName (qualified name)":

You may wonder what this http://ttdev.com/ss namespace means. The answer is that it has no particular meaning. Even though it is a URL, it does NOT mean that you can use a browser to access this URL to get a web page (if you do, you may get a file not found error). The only important thing is that it must be globally unique. As I have registered the domain name ttdev.com, it must be globally unique.

Note that the namespace is a completely different concept from the endpoint. The endpoint really is the location, while the namespace is just a unique id. I could easily move the web service to another web server and thus it will have a different endpoint, but the namespaces of its operations will remain unchanged.

**RPC style web service**

Your concat operation may take two parameters. One is named "s1" and is a string. The other is named "s2" and is also a string. The return value is also a string:
Chapter 1  Designing the interface for a simple web service

However, what does the above "string" type mean? Is it the Java string type? No, you can't say that because it must be language neutral. Fortunately, the XML schema specification defines some basic data types including a string type. Each of these data types has a QName as its id. For example:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Local name</th>
<th>namespace</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>string</td>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
</tr>
<tr>
<td>integer</td>
<td>int</td>
<td><a href="http://www.w3.org/2001/XMLSchema">http://www.w3.org/2001/XMLSchema</a></td>
</tr>
</tbody>
</table>

So, the interface of your operation should be written as:

An operation
Local name: concat
Namespace: http://ttdev.com/ss
Parameters:
  sl: string in http://www.w3.org/2001/XMLSchema
  s2: string in http://www.w3.org/2001/XMLSchema
Return:
  string in http://www.w3.org/2001/XMLSchema

Actually, in web services, a method call is called an "input message" and a parameter is called a "part". The return value is called an "output message" and may contain multiple parts. So, it is more correct to say:

An operation
Local name: concat
Namespace: http://ttdev.com/ss
Input message:
  Part 1:
    Name: s1
    Type: string in http://www.w3.org/2001/XMLSchema
  Part 2:
    Name: s2
    Type: string in http://www.w3.org/2001/XMLSchema
Output message:
  Part 1:
    Name: return
    Type: string in http://www.w3.org/2001/XMLSchema

When someone calls this operation, he can send you an XML element as the input message like:
When you return, the output message may be like:

```
<foo:concat xmlns:foo="http://ttdev.com/ss">
  <s1>abc</s1>
  <s2>123</s2>
</foo:concat>
```

This kind of web service is called "RPC style" web service (RPC stands for...
"Remote Procedure Call"). That is, the operation QName and the names of the parts are used to create the input and output messages.

**Document style web service**

The above way is not the only way you design the interface of your web service. For example, you may say that its input message only contains a single part (see the diagram below) which is an element defined in a schema. In that schema, it is defined as an element named "concatRequest" that contains two child elements <s1> and <s2>:

An operation

Local name: concat
Namespace: http://ttdev.com/ss
Input message:
  Part 1:
  Name: concatRequest
  Element:
Output message:

... 

Note that the schema is included in the interface of your web service:
As you can see above, a part may be declared as a particular element (<concatRequest> defined in your schema) or as any element having a particular type (string defined in XML schema specification). In either case it is identified using a QName.

When someone calls this operation, he will send you a <concatRequest> element as the input message like:

```
<foo:concatRequest xmlns:foo="http://ttdev.com/ss">
  <s1>abc</s1>
  <s2>123</s2>
</foo:concatRequest>
```

Similarly, for the output message, you may specify that it contains only one part and that part is a <concatResponse> element:
This kind of web service is called "document style" web service. That is, the input message will contain a single part only which is well defined in a schema. The same is true of the output message.

If you go back to check the input message for the RPC style service, it should be revised as:
Chapter 1  Designing the interface for a simple web service

This is because `<foo:concat>`, `<s1>` and `<s2>` are not defined in any schema and therefore you must explicitly state the XML element types of the content of `<s1>` and `<s2>`.

Now, let’s compare the input messages of the RPC style web service and the document style web service:

**RPC style**

```xml
<foo:concat>
  xmlns:foo="http://ttdev.com/ss"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance">
  <s1 xsi:type="xsd:string">abc</s1>
  <s2 xsi:type="xsd:string">123</s2>
</foo:concat>
```

**Document style**

```xml
<foo:concatRequest xmlns:foo="http://ttdev.com/ss">
  <s1>abc</s1>
  <s2>123</s2>
</foo:concatRequest>
```

Not much difference, right? The significant difference is that the former can’t be validated with a schema while the latter can. Therefore, document style web service is becoming the dominant style. According to an organization called "WS-I (web services interoperability organization)", you should use document style web services only.

**Determining the operation for a document style web service**

To call an operation in a document style web service, one will send the single part of the input message only. Note that it does NOT send the operation name in any way. Then if there are more than one operations in the web service (see the diagram below), how can it determine which one is being called? In that
case, it will see if the input message is a \(<\text{concatRequest}\)> or a \(<\text{someElement}\)> to determine. What if both take a \(<\text{someElement}\)>? Then it is an error and it won't work:

Port type

Actually, a web service doesn't directly contain a list of operations. Instead (see the diagram below), operations are grouped into one or more "port types". A port type is like a Java class and each operation in it is like a static method. For example, in the web service above, you could have a port type named "stringUtil" containing operations for strings, while having another port type named "dateUtil" containing operations for dates. The name of a port type must also be a QName:
Binding

Actually, a port type may allow you to access it using different message formats. The message format that you have seen is called the "Simple Object Access Protocol (SOAP)" format. It is possible that, say, the stringUtil port type may also support a plain text format:

```
concat(s1='abc', s2='123')
```

In addition to the message format, a port type may allow the message to be carried (transported) in an HTTP POST request or in an email. Each supported combination is called a "binding":

```java
An operation
Local name: bar
Namespace: http://ttdev.com/ss
...

An operation
Local name: concat
Namespace: http://ttdev.com/ss
...
```
Chapter 1 Designing the interface for a simple web service

What bindings should your port type support? SOAP+HTTP is the most common combination. So, you should probably use this binding in practice.

**Port**

Suppose that there are just too many people using your web service, you decide to make it available on more than one computers. For example (see the diagram below), you may deploy the above binding 1 on computers c1, c2 and c3 and deploy binding 2 on c3. In that case it is said that you have four ports. Three ports are using binding 1 and one using binding 2:
Note that it does NOT mean that the requests received by these three computers will be forwarded to a computer hiding behind for processing. Instead, it means that there is some software implementing the port type installed on these three computers. There is no requirement that the same piece of software is installed onto the different computers. For example, on c1, port 1 may be written in Java, while on c2, port 2 may be written in C#. The important point is that they both support the operations specified in port type stringUtil and the message format and transport specified in the binding 1. Port 4 must also implement the same operations too (same port type) but the message format and transport are different.

To tell others about this arrangement, you include these ports in the interface of the web service:
Target namespace

You have been using the same namespace for the operation names, port type names and etc. in this web service. Do they have to be in the same namespace? By default, this is the case: There is a single namespace for a web service to put the names into. This is called the “target namespace” for the web service:
You've been using http://ttdev.com/ss as the target namespace. Is it a good choice? Basically a namespace is good as long as it is globally unique. So this one should be good. However, people may try to download a web page from this URL. When it doesn't work, they may suspect that your web service is out of order. To avoid this confusion, you may use something called URN (Uniform Resource Name) as the namespace.

A namespace must be a URI. URI stands for Uniform Resource Identifier. There are two kinds of URI. One is URL such as http://www.foo.com/bar. The other is URN. A URN takes the format of urn:<some-object-type>::<some-object-id>. For example, International ISBN Agency has made a request to the IANA (International Assigned Numbers Association) that it would like to manage the object type named "isbn". After the request has been approved, the International ISBN Agency can declare that a URN urn:isbn:1-23-456789-0 will identify a book whose ISBN is 1-23-456789-0. It can determine the meaning of the object id without consulting IANA at all.

Similarly, you may submit a request to IANA to register your Internet domain name such as foo.com as the object type. Then on approval you can use URNs like urn:foo.com:xyz to identify an object xyz in your company. What xyz means or its format is completely up to you to decide. For example, you may use urn:foo.com:product:123 (so xyz is product:123) to mean the product #123 produced by your company, or urn:foo.com:patent/123 (so xyz is patent/123) to mean a patent coded 123 in your company.
However, this will create a lot of workload on you and on IANA (one registration per company!). As you have already registered the domain name foo.com, it is unlikely that someone will use it in their URN's. So, you may want to go ahead and use foo.com, or, as many people do, foo-com as the object type without registration with IANA and hope that there won't be any collision.

An XML namespace must be a URI. You can use a URL or a URN. Functionally there is no difference at all. For example, you may use say urn:ttdev.com:ss as the target namespace for your web service instead of http://ttdev.com/ss without changing any functionality.

By the way, if you are going to lookup references on URN, do NOT try to find terms like "object type" or "object id". The official terms are:

`urn:isbn:1-23-456789-0`

URN namespace specific string (NSS)

URN namespace identifier (NID). This namespace is NOT the namespace in XML!

**WSDL**

By now you have finished designing the interface for your web service:
A web service

Target namespace: http://ttdev.com/ss

A schema

Port type: stringUtil
concat

Binding
Name: binding1
Port type:
Format: SOAP
Transport: HTTP

Binding
Name: binding2
Port type:
Format: TEXT
Transport: SMTP

A web service is platform neutral, language neutral and can be accessed across the Internet.

A web service has one or more ports. Each port is a binding deployed at a certain network address (endpoint). A binding is a port type using a particular message format and a particular transport protocol. A port type contains one or more operations. An operation has an input message and an output message. Each message has one or more parts. Each part is either a certain element defined in the schema of the web service, or any element belonging to a certain element type in that schema. All this information is fully described in WSDL.

To call a RPC style web service, one will create an XML element with the name of the operation and a child element for each of its input message part. To call a document style web service, one will just send the one and only part of its input message. Because the XML element used to call a RPC style web service is not defined in any schema, for better interoperability, one should create document style web services.

The web service, and each of its ports, bindings, port types and operations, has a QName uniquely identifying it. A QName has a local part and an XML
namespace. An XML namespace is a URI that is globally unique. By default the names of all these components are put into the target namespace of the web service.

There are two kinds of URI: URL and URN. URN takes the form of urn:<NID>:<NSS>. You can use either as an XML namespace. The only difference is that a URL is suggesting that it is the location of an object, while a URN is purely an id of the object.
Chapter 2
Implementing a web service
What's in this chapter?

In this chapter you'll learn how to implement the web service interface designed in the previous chapter.

Installing Eclipse

You need to make sure you have Eclipse v3.3 (or later) installed and it is the bundle for Java EE (the bundle for Java SE is NOT enough). If not, go to http://www.eclipse.org to download the Eclipse IDE for Java EE Developers (e.g., eclipse-jee-europa-fall-win32.zip). Unzip it into c:\eclipse. Then, create a shortcut to run "c:\eclipse\eclipse -data c:\workspace". This way, it will store your projects under the c:\workspace folder. To see if it's working, run it and make sure you can switch to the Java EE perspective:

BUG ALERT: If you're using Eclipse 3.3.1, there is a serious bug in it: When visually editing WSDL files Eclipse will frequently crash with an OutOfMemoryError. To fix it, modify c:\eclipse\eclipse.ini:

```
-showsplash
org.eclipse.platform
 launcher.\XXMaxPermSize 256m
-vmargs
-Xms40m
-Xmx256m
-XX:MaxPermSize=256m
```

Deleting them

This line must be put after -vmargs

Installing Axis2

Next, go to http://ws.apache.org/axis2 to download the "Standard Binary Distribution" (e.g. axis2-1.3-bin.zip). Unzip it into c:\axis. To run the Axis server, change into c:\axis\bin and run axis2server.bat. You should see:
Then open a browser and access http://localhost:8080. You should see:

![Deployed services](image)

**Version**

<table>
<thead>
<tr>
<th>Available operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>* getVersion</td>
</tr>
</tbody>
</table>

It means that there is an existing web service called "Version" available. Click on that "Version" link and you should see its WSDL file:
Chapter 2  Implementing a web service

Installing the Axis2 plugin for Eclipse

Go to http://ws.apache.org/axis2/tools/index.html and download the Code Generator Wizard - Eclipse Plug-in. BUG ALERT: v1.4 of the plugin contains a critical bug. Use v1.3 instead! Suppose that it is axis2-eclipse-codegen-wizard.zip. Unzip it into the c:\eclipse\plugins folder. Restart Eclipse if required. To check if it's working, choose "File | New | Other" and you should see the "Axis2 Code Generator":

```
<wsdl:definitions targetNamespace="http://ws.apache.org/axis2"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  targetNamespace="http://example.com"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/en
WSDL file for the web service

Suppose that you'd like to create a web service described in the previous chapter:
Implementing a web service

To write it using the real WSDL language, it should be:

```
<wsdl:definitions targetNamespace="http://ttdev.com/ss"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <wsdl:types>
    <xsd:schema targetNamespace="http://ttdev.com/ss"
                xmlns:tns="http://ttdev.com/ss"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="concatRequest">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="s1" type="xsd:string"/>
            <xsd:element name="s2" type="xsd:string"/>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="concatResponse" type="xsd:string"/>
    </xsd:schema>
  </wsdl:types>

  <wsdl:message name="concatRequest">
    <wsdl:part name="concatRequest" type="tns:concatRequest"/>
  </wsdl:message>

  <wsdl:message name="concatResponse">
    <wsdl:part name="concatResponse" type="tns:concatResponse"/>
  </wsdl:message>

  <wsdl:operation name="concat">
    <wsdl:input message="tns:concatRequest"/>
    <wsdl:output message="tns:concatResponse"/>
  </wsdl:operation>

  <wsdl:binding name="concat" type="tns:concat">
    <wsdl:soap12binding operation="tns:concat" style="document"/>
  </wsdl:binding>

  <wsdl:port name="concatPort" binding="tns:concat">
    <wsdl:address location="/concat"/>
  </wsdl:port>

</wsdl:definitions>
```
This defines the schema and the port type. To define the binding and the port:
In fact, in a SOAP binding, you need to specify some more details:
RPC version of the web service

If the web service was a RPC style service, then the WSDL file would be like:

```xml
<wsdl:definitions ...>
  ...
  <wsdl:message name="concatRequest">
    <wsdl:part name="concatRequest" element="tns:concatRequest" />
  </wsdl:message>
  <wsdl:message name="concatResponse">
    <wsdl:part name="concatResponse" element="tns:concatResponse" />
  </wsdl:message>
  ...
  <wsdl:binding name="SimpleServiceSOAP" type="tns:SimpleService">
    <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="concat">
      <soap:operation soapAction="http://ttdev.com/ss/concat" />
      <wsdl:input>
        <soap:body parts="concatRequest" use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body parts="concatResponse" use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  ...
</wsdl:definitions>
```
Implementing a web service

As RPC style is not good for interoperability, you'll continue to use the document style version.

Creating the WSDL file visually

It may be error prone to manually create such a WSDL file. Instead, you may
use the Eclipse to do it. First, create a new Java project named SimpleService in Eclipse:

Make sure you use separate folders for sources and class files. Then go ahead and complete the creation of the project. Next, right click the project and choose "New | Other" and then "Web Services | WSDL":

If you don't see this option, it means that you haven't installed the Java EE version of Eclipse. If it is working, click "Next" and enter SimpleService.wsdl as the filename:
Click "Next". Then input as shown below:

Click "Finish". Then you will see something like:
This is the WSDL code. To edit it visually, click the "Design" tab at the bottom of the editor window. Then you'll see:

Double click on the endpoint to change it to http://localhost:8080/axis2/services/
SimpleService:

Double click on the name of operation and change it to "concat":

Set the name of the operation. The XML element names for the input and output parts will be changed automatically:

For the moment, the input part is an <concat> element. You'd like to change it to <concatRequest>. But for now, put the cursor on the arrow to its right first. The arrow will turn into blue color. Wait a couple of seconds then a preview window will appear showing the definition of the <concat> element:

Clicking anywhere else will make that preview window disappear. To edit the schema definition, click on the blue arrow. A new editor window will appear:
To edit it visually, click the "Design" tab at the bottom, you'll see:

Double click on "in" and change it to "s1":

Right click it and choose "Add Element" and set the name to "s2":

By default the type is already set to string. If you wanted it to be say an int instead, you would double click on the type and it would become a combo box
and then you could choose "int":

If you wanted \( s_2 \) to appear before \( s_1 \) in the sequence, you could drag it and drop it before \( s_1 \):

But for now, make sure it is \( s_1 \) first and then \( s_2 \). Next, right click on the \(<\text{concat}>\) element and choose "Refactor | Rename", then change its name to concatRequest:

You're done with the \(<\text{concatRequest}>\) element. Now return to the WSDL editor to work on the response message. For the moment, the \(<\text{concatResponse}>\) is like:
That is, it is an element that contains a sequence of `<out>` element:

```xml
<foo:concatResponse>
  <foo:out>abc</foo:out>
</foo:concatResponse>
```

However, in your design, the response is simple type element, not a complex type element:

```
```

Its body contains a string instead of other elements.

To do that, go into the schema editor to edit the `<concatResponse>` element:

Right click it and choose "Set Type | Browse":

```
```
Choose "string":

Then it will be like:

That's it. To review the whole schema, click on the icon at the upper left corner:
Then you'll see:

This looks fine. Now, save the file.

**Validating the WSDL file**

The next step is to validate the WSDL file to make sure it conforms to the various web services standards. To do that, right click the SimpleService.wsdl file in Eclipse and choose "Validate". If there were anything wrong, they would be reported in the Problems window. For example, here I had introduced an error into the file:
Chapter 2 Implementing a web service

Implementing a web service

Generating a service stub

Next, in order to implement the web service, you will generate a "service stub" (see the diagram below). When a request message comes in, the service stub will convert the <concatRequest> XML element into a ConcatRequest Java object. Then it will pass it to the concat() method in a service skeleton to be supplied by you. Your concat() method will create and return a ConcatResponse Java object. The service stub will convert it into a <concatResponse> XML element and return it to the client:

```xml
<wsdl:binding name="SimpleServiceSOAP" type="tns:SimpleService">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org
  <soap:operation name="concat">
  <soap:operation soapAction="http://ttdev.com/ss/NewOperation"/>
  <wsdl:input>
    <soap:body parts="bar" use="literal"/>
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal"/>
  </wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="SimpleService">
  <wsdl:port binding="tns:SimpleServiceSOAP" name="SimpleServiceSOAP">
    <soap:address location="http://localhost:8080/axis2/services/"/>
  </wsdl:port>
</wsdl:service>

Try to include an unknown part into the SOAP body

Design Source

1 error, 0 warnings, 0 info
Description
Errors (1 item)
The bar part specified as a part for the SOAP body is undefined. You can only specify defined parts.

Generating a service stub

Next, in order to implement the web service, you will generate a "service stub" (see the diagram below). When a request message comes in, the service stub will convert the <concatRequest> XML element into a ConcatRequest Java object. Then it will pass it to the concat() method in a service skeleton to be supplied by you. Your concat() method will create and return a ConcatResponse Java object. The service stub will convert it into a <concatResponse> XML element and return it to the client:
To implement this idea, in Eclipse choose "File | New | Other" and choose "Axis2 Code Generator" (see below). The default is to generate Java code from WSDL. This is what you want:

Click "Next" (see below). Click "Browse" to locate your SimpleService.wsdl file:
Click "Next" (see below). Set the options as shown below:
Chapter 2  Implementing a web service

Note that by default how the namespaces will be mapped to Java packages. For example, your SimpleService port type in http://ttdev.com/ss namespace will be mapped to a SimpleService Java interface in the com.ttdev.ss Java package:

http://ttdev.com/ss

Reverse it so that it goes from broad (com) to narrow (ttdev)

com.ttdev.ss

Of course this is just the default. You can change the Java package names in
the dialog box above. But for our purpose the default mapping is just fine. So, click "Next" (see below) and enter the information as shown below:

![Axis2 Codegen Wizard](image)

Click "Finish". Right click your project and choose "Refresh". Then you'll see some files have been generated:
The Java source files are in errors because they are referring to the Axis2 jar files but they are not on the build path. So, go to the build path dialog and click "Add JARs":

Choose all the jar files in the "lib" folder in your project:
Then the errors will disappear.

**Implementing the web service**

To implement the web service, modify the SimpleServiceSkeleton.java which is the service skeleton:

```java
public class SimpleServiceSkeleton implements SimpleServiceSkeletonInterface {
    public com.ttdev.ss.ConcatResponse concat(
            com.ttdev.ss.ConcatRequest concatRequest0) {
        String result = concatRequest0.getS1()+concatRequest0.getS2();
        ConcatResponse response = new ConcatResponse();
        response.setConcatResponse(result);
        return response;
    }
}
```

Where do the ConcatRequest class and ConcatResponse class come from? They were generated by the Axis2 Code Generator Wizard:
Deploying a web service

To deploy the web service with the Axis2 server, copy the files as shown below:

```
c: \workspace \SimpleService
  \resources
    \services.xml
    \SimpleService.wsdl

bin
  \com \ttdev
    \ss
      ...

\c: \axis\repository\services
  \SimpleService
    \META-INF
      \services.xml
      \SimpleService.wsdl
```

Now, start the Axis2 server by running `c:\axis\bin\axis2server.bat`. You should see that it is picking up your SimpleService:
Go to http://localhost:8080 and you should see your SimpleService listed:

![Deployed services]

**Deployed services**

**SimpleService**

Available operations

- concat

**Version**

Available operations

- getVersion

To see its WSDL file, just click the "SimpleService" link:
Creating a client using a client stub

To call this web service, you can use the Axis2 Code Generator Wizard to generate a "client stub". When you call a method on it (see the diagram below), it will convert your Java data/objects into the right format (XML), create a request message in the right format (SOAP), send it over the Internet to the right endpoint using the right transport protocol (HTTP) to invoke that operation, wait for the response message, convert the XML back into Java data/object and then return it to you:
To implement this idea, run the Axis2 Code Generator Wizard as before until you see the follow screen. Then tell it to generate client side code instead of server side code:

Then tell it to put the code into your SimpleService project. This time, no need to copy the Axis2 jar files again:

Click "Finish" and then refresh the project. You'll see a couple of new Java
Among them, SimpleServiceStub.java is the client stub. As you're simulating someone else calling your web service, they should not be mixed with the code implementing the web service. Therefore, move them into another package such as com.ttdev.ss.client.

Next, create a SimpleClient.java file in the com.ttdev.ss.client package:

```java
package com.ttdev.ss.client;

import java.rmi.RemoteException;
import com.ttdev.ss.client.SimpleServiceStub.ConcatRequest;
import com.ttdev.ss.client.SimpleServiceStub.ConcatResponse;

public class SimpleClient {
    public static void main(String[] args) throws RemoteException {
        SimpleServiceStub service = new SimpleServiceStub();
        ConcatRequest request = new ConcatRequest();
        request.setS1("abc");
        request.setS2("123");
        ConcatResponse response = service.concat(request);
        System.out.println(response.getConcatResponse());
    }
}
```

Run it and it should work:
Chapter 2  Implementing a web service

Undeploying a web service

If you’d like to undeploy a web service, all you need to do is to delete the SimpleService folder:

```
c:  
  axis
  repository
  services
  SimpleService
      META-INF
      services.xml
      SimpleService.wsdl
  com
      ttdev
      ss
      ...
```

This works even when the Axis2 server is running. It will note the removal of the folder and undeploy the service:
If you put the folder back, it will be deployed again:

```
Using JAVA_HOME  C:\Program Files\Java\jdk1.5.0_02
Using AXIS2_HOME  c:\axis2-1.3\bin\..
[INFO] [SimpleAxisServer] Starting
[INFO] [SimpleAxisServer] Using the Axis2 Repository:\axis2-1.3\repo
[SimpleAxisServer] Using the Axis2 Repository:\axis2-1.3\bin\..
[SimpleAxisServer] Using the Axis2 Configuration File:\axis2-1.3\axis2.xml
[INFO] Deploying module: addressing-1.3
[INFO] Deploying module: metadataExchange-1.3
[INFO] Deploying module: ping-1.3
[INFO] Deploying module: script-1.3
[INFO] Deploying module: soapmonitor-1.3
[INFO] script module activated
[INFO] Deploying Web service: SimpleService
[INFO] Deploying Web service: version.aar
[INFO] [SimpleAxisServer] Started
[SimpleAxisServer] Started
[INFO] Listening on port 8080
[INFO] Undeploying Web service: SimpleService
```

This is called "hot deployment".

**Summary**

Tomcat hosts one or more web applications. The Axis server is installed as one
of the web applications. It in turn hosts one or more web services.

Most usually your input message or output message is sent in a SOAP message. A SOAP message is always an <Envelope> element. It may contain a <Header> which contains one or more header entries/elements. The <Envelope> must contain a <Body> which may contain one or more body entries/elements. For a document style web service, the one and only input message part is usually the single body entry. For a RPC style web service, the element named after the operation will usually contain all message parts and is then included as the single body entry.

To create a web service, you first create a WSDL file describing its interface. This can be done manually or using a tool like Eclipse. Then use the Axis Code Generator Wizard on the WSDL file to generate a service stub. Then fill in the code in the service skeleton. The service stub will convert the XML elements in a request message into Java data/objects, call your skeleton and convert the Java objects returned into XML elements and put them into the response message.

To deploy a web service, copy the class files and the services.xml file to the Axis2 server according to a specific folder structure. To undeploy a web service, just delete that folder. The Axis2 server supports hot deployment. It means you can deploy or undeploy a service while it is running.

The endpoint of the deployed web service is http://localhost:8080/axis2/services/<name-of-your-service>.

To call a web service, run the Axis Code Generator Wizard on the WSDL file to generate a client stub. Then, in your code create an instance of the client stub and call its methods as if it were the web service. The client stub will convert the Java data/objects into XML elements, create the request message in the right format, send it to the right endpoint using the right transport protocol and convert the XML elements in the response message back into Java data/objects.
Chapter 3

Optimizing the development environment
What's in this chapter?

In this chapter you'll learn how to optimize the development environment.

Placing the class files into Axis directly

At the moment, whenever you make changes to say your web service Java code (SimpleServiceSkeleton.java), you will have to copy the class file into the Axis server again. This is troublesome. To solve this problem, you can tell Eclipse to put the class files directly into the Axis repository:

However, there is a problem: Eclipse can only use a folder inside the project as the output folder. To allow you to work around this restriction, fortunately Eclipse allows you to link such a folder to an outside folder:

To implement this idea, right click the project and choose "New | Folder". Enter the information as shown below:
To set the output folder, right click the project in Eclipse and choose "Properties", then choose "Java Build Path" and choose the "Source" tab:

Click "Browse" and choose the "out" folder. Then confirm to delete the existing "bin" folder as it is no longer used.
Now the class files are in the right place. The next step is to make the META-INF folder appear in the service folder. To do that, you need to have such a folder in the "src" folder:

Therefore, rename your "resources" folder as META-INF and move it into "src". To verify that this setup is working, modify the code to turn the result string into upper case:

```java
public class SimpleServiceSkeleton implements SimpleServiceSkeletonInterface {
    public com.ttdev.ss.ConcatResponse concat(
        com.ttdev.ss.ConcatRequest concatRequest0) {
        String result = concatRequest0.getS1() + concatRequest0.getS2();
        ConcatResponse response = new ConcatResponse();
        response.setConcatResponse(result.toUpperCase());
        return response;
    }
}
```

Now start the Axis2 server. Run the client and the output should be in upper case:

```
Making changes take effect immediately
Let's restore the code now:

```java
public class SimpleServiceSkeleton implements SimpleServiceSkeletonInterface {
    public com.ttdev.ss.ConcatResponse concat(
        com.ttdev.ss.ConcatRequest concatRequest0) {
        String result = concatRequest0.getS1() + concatRequest0.getS2();
        ConcatResponse response = new ConcatResponse();
        response.setConcatResponse(result.toUpperCase());
        return response;
    }
}
```

Will it take effect while the Axis server is running? No. It will still output ABC123. This is because by default once the Axis server loads a web service, it will not monitor changes to its file any more. To change this behavior, modify c:\axis\conf\axis2.xml:
You may have noticed that there is also a hot deployment option in addition to the hot update option. What's the difference? It is explained in the diagram:

You need to restart the Axis server so that this hot update option takes effect. Then modify the Java code:

```java
public class SimpleServiceSkeleton implements SimpleServiceSkeletonInterface {  
    public com.ttdev.ss.ConcatResponse concat(  
        com.ttdev.ss.ConcatRequest concatRequest0) {  
        String result = concatRequest0.getS1()+concatRequest0.getS2();  
        ConcatResponse response = new ConcatResponse();  
        response.setConcatResponse("hello: "+result);  
        return response;  
    }
}
```

Save the code. Then you should see that the Axis server redeploying your service:
Run the client and it should work:

```
C:\> set JAVA_OPTS=-Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
C:\> axis2server.bat
```

Type it all on one line

Launch it as usual

```
Console   Problems  @ Javadoc  Declaration
<terminated> SimpleClient (1) [Java Application] C:\Program Files\log4j\warn: No appenders could be found
log4j:warn Please initialize the log4j
hello: abc123
```

Note that the Axis server looks for changes every 10 seconds. So it may take some time before the web service is redeployed.

**Debugging a web service**

To debug your web service in Eclipse, you need to set an environment variable before launching the Axis server (shut it down first if it's running):

```
C:\> set JAVA_OPTS=-Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
C:\> axis2server.bat
```

This way the Axis server will run the JVM in debug mode so that the JVM will listen for connections on port 8000. Later you'll tell Eclipse to connect to this port. Now, set a breakpoint here:
Choose "Open Debug Dialog":

Right click "Remote Java Application" and choose "New". Name this configuration "Debug Axis" (it doesn't really matter). Make sure your SimpleService project is selected and make sure the port is 8000:
Click "Debug" to connect to the JVM running the Axis server. Now run the client to call the web service. Eclipse will stop at the breakpoint:

```java
package com.ttdev.ss;

/**
 * SimpleServiceSkeleton java skeleton for the axisService
 */

public class SimpleServiceSkeleton implements SimpleServiceSkeletonInterface {
    public com.ttdev.ss.ConcatResponse concat {
        com.ttdev.ss.ConcatRequest concatRequest = 
        String result = concatRequest0.getSeq() + concatRequest0.getSeq2();
        ConcatResponse response = new ConcatResponse();
        response.setConcatResponse("hello: "+result);
        return response;
    };
```
Having to set this environment variable every time is not fun. So, you may create a batch file `c:\axis\bin\debug.bat`:

```
set JAVA_OPTS=-Xdebug -Xrunjdwp:transport=dt_socket,address=8000,server=y,suspend=n
```

Then in the future you can just run it to start the Axis server in debug mode.

**Generating code automatically**

For the moment you're using the Code Generator Wizard to generate the code from the WSDL file. If you modify the WSDL file, you'll have to do it once again. This is troublesome. You need an automated process to generate the code. To do that, you'll edit the build.xml file that was generated by the Code Generator Wizard. But first, you need to understand the structure of the build.xml (see below). A build.xml file contains a project, which is like a class in a Java file. A project contains one or more targets. A target is like a method in a Java class. A target contains one or more tasks. A task is like a statement in a Java method:

```
<project>
  <target name="target1" depends="task1,task2,task3">
    <task name="task1" />
    <task name="task2" />
    <task name="task3" />
  </target>
  <target name="target2" depends="task1,task2,task3">
    <task name="task1" />
    <task name="task2" />
    <task name="task3" />
  </target>
</project>
```

Now, let's edit the build.xml file:
The project

<?xml version="1.0" encoding="UTF-8"?>
<project ...>

<target name="generate-service">
  <wsdl2code
      wsdlfilename="SimpleService.wsdl"
      serverside="true"
      generateservicexml="true"
      skipbuildxml="true"
      serversideinterface="true"
      namespacetopackages="http://ttdev.com/ss=com.ttdev.ss"
      targetsourcefolderlocation="src"
      targetresourcesfolderlocation="src/META-INF"/>
</target>
</project>

Next, you are about to run this build.xml file using a program called "Ant". However, the <wsdl2code> task is not a built-in task in Ant and therefore Ant doesn't know how to execute it. It is implemented by a Java class named AntCodegenTask in c:\axis\lib\axis2-ant-plugin-1.3.jar. To tell Ant how the <wsdl2code> task is implemented, modify build.xml:

public interface SimpleServiceSkeletonInterface {
  public ConcatResponse concat(...);
}

public class SimpleServiceSkeleton
  implements SimpleServiceSkeletonInterface {
    public ConcatResponse concat(...) { 
      ...
    }
  }

Put the Java files into the "src" folder which is a relative path to the project root.

Put the "resources files" (e.g., services.xml) into the "src/META-INF" folder which is a relative path to the project root.

Map the http://ttdev.com/ss namespace to the com.ttdev.ss package. This is not really needed here as it is the default. It is here just to show you the syntax.

Put the path to the WSDL file. Here you are using a relative path. It is relative to the build.xml file (project root).

Generate code for the service. Otherwise it will generate code for the client.

Generate the services.xml file

Don't generate the build.xml. Otherwise it will overwrite this file!

Generate an interface in addition to the skeleton:
To define an environment variable AXIS2_HOME, you can either do it in Windows or in Eclipse. Let's do it in Eclipse. Choose "Window | Preferences | Ant | Runtime", choose the "Properties" tab:

Click "Add Property" and enter the data as shown below:
Chapter 3  Optimizing the development environment

Now you're about to run Ant. To verify that it is really working, rename your SimpleServiceSkeleton.java file as SimpleServiceImpl file. Then delete all the other Java files in the package. Delete the files in the META-INF folder too.

**BUG ALERT:** In Axis2 1.3 there is a bug in the Code Generator Wizard. After installing it, you'll be unable to run Ant in Eclipse. To workaround the problem, in the Ant Runtime window above, choose the "Classpath" tab and click "Ant Home" and browse to choose the org.apache.ant folder in c:\eclipse\plugins:

To run Ant, right click the build.xml file and then choose "Run As | Ant Build..." as shown below:

Then choose the "generate-service" target and click "Run":

You should see that it is working in the console:

```
<wsdl2code> log4j:WARN No appenders could be found for logger ( :
<wsdl2code> log4j:WARN Please initialize the log4j system proper
BUILD SUCCESSFUL
Total time: 5 seconds
```

Then refresh the project and you’ll see that the Java files and the files in META-INF have been recreated. Now, ideally if your WSDL file is modified, all you need to do is to run the build.xml file again. However, this is not the default behavior. By default, the `<wsdl2code>` task will not overwrite any existing file! To tell it to do so, set an option:

```
<target name="generate-service">
<wsdl2code
  wsdlfilename="SimpleService.wsdl"
  serverside="true"
  generateservicexml="true"
  skipbuildxml="true"
  serversideinterface="true"
  nam espacetoppackages="http://ttdev.com/ss=com.ttdev.ss"
  targetsourcefolderlocation="src"
  targetresourcesfolderlocation="src/META-INF"
  overwrite="true"/>
```
But this introduces another problem: If you fill your code into SimpleServiceSkeleton, when you run build.xml, the file will be overwritten and your code will be lost! The idea is not to use SimpleServiceSkeleton any more. Instead, create your own SimpleServiceImpl that implements the same interface:

In order to use your SimpleServiceImpl to implement the web service, you need to know how the Axis server knows which Java class implements your web service. It looks up the class name in the services.xml file:

```
<serviceGroup>
  <service name="SimpleService">
    <messageReceivers>
      <messageReceiver
        mep="http://www.w3.org/ns/wsdl/in-out"
        class="com.ttdev.ss.SimpleServiceMessageReceiverInOut" />
    </messageReceivers>
    <parameter name="ServiceClass">com.ttdev.ss.SimpleServiceSkeleton</parameter>
    <parameter name="useOriginalwSDL">true</parameter>
    <parameter name="modifyUserWSDLPortAddress">true</parameter>
    <operation name="concat"
      mep="http://www.w3.org/ns/wsdl/in-out">
      <actionMapping>
        http://ttdev.com/ss/NewOperation
      </actionMapping>
      <outputActionMapping>
        http://ttdev.com/ss/SimpleService/concatResponse
      </outputActionMapping>
    </operation>
  </service>
</serviceGroup>
```

You could modify this services.xml file every time it is generated, but it is too troublesome and easy to forget. A much better way is to let Ant do it for you automatically:
Chapter 3 Optimizing the development environment

75

Run it and refresh the project. Check the services.xml file and it should be using your SimpleServiceImpl:

```xml
<serviceGroup>
  <service name="SimpleService">
    <messageReceivers>
      <messageReceiver mep="http://www.w3.org/ns/wsdl/in-out"
                       class="com.ttdev.ss.SimpleServiceMessageReceiverInOut" />
    </messageReceivers>
    <parameter name="ServiceClass">com.ttdev.ss.SimpleServiceImpl</parameter>
    <parameter name="useOriginalwsdl">true</parameter>
    <parameter name="modifyUserWSDLPortAddress">true</parameter>
    <operation name="concat">
      <actionMapping>
        http://ttdev.com/ss/NewOperation
      </actionMapping>
      <outputActionMapping>
        http://ttdev.com/ss/Service/concatResponse
      </outputActionMapping>
    </operation>
  </service>
</serviceGroup>
```

Generating client code automatically

To generate the client code, it is very similar:

```xml
<project ...
<target name="generate-service">
  <wsdl2code
    wsdlfilename="SimpleService.wsdl"
    serverside="true"
    generateservicexml="true"
    serversideinterface="true"
    usesourcesideinterface="true"
    targetsourcefolderlocation="src"
    targetresourcesfolderlocation="src/META-INF"
    overwrite="true" />

  <replaceregexp
    file="src/META-INF/services.xml"
    match="SimpleServiceSkeleton"
    replace="SimpleServiceImpl"/>
</target>
</project>
```
<project ...>
...</project>

<target name="generate-service">
<wsdl2code
  wsdlfilename="SimpleService.wsdl"
  serverside="true"
  generateservicexml="true"
  serversideinterface="true"
  namespacetopackages="http://ttdev.com/ss=com.ttdev.ss"
  targetsourcefolderlocation="src"
  targetresourcesfolderlocation="src/META-INF"
  overwrite="true"/>
<replaceregexp
  file="src/META-INF/services.xml"
  match="SimpleServiceSkeleton"
  replace="SimpleServiceImpl"/>
</target>

<target name="generate-client">
<wsdl2code
  wsdlfilename="SimpleService.wsdl"
  skipbuildxml="true"
  namespacetopackages="http://ttdev.com/ss=com.ttdev.ss.client"
  targetsourcefolderlocation="src"
  overwrite="true"/>
</target>

Delete the files in the client package except SimpleClient.java which was created by you. Run build.xml and choose the "generate-client" target. Refresh the project and you'll see the Java files in the client package again.

To make sure everything is working, start the Axis server and run the client. It should continue to work.

Summary

You can set the output folder in Eclipse so that you don't need to copy the files into the service folder in Axis manually.

To make sure the changes to your Java code take effect immediately, you can enable hot update in the Axis server.

To debug a web service, tell the Axis server to run the JVM in debug mode, set a breakpoint in the Java code and make a Debug configuration in Eclipse to connect to that JVM.

To automate the process of generating Java code from a WSDL file, you can use the <wsdl2code> Ant task. In general you'll want it to overwrite existing files. To prevent from overwriting your own code, you should never modify the code generated. Instead, create your own service implementation class that implements the service interface and modify services.xml to tell the Axis server to use that class.
Chapter 4

Understanding the calling process
What's in this chapter?

In this chapter you'll learn what is happening internally when you call a web service.

Calling a web service without a client stub

Suppose that you'd like to call a web service without a client stub. To do that, in the SimpleService project in Eclipse, create a file LowLevelClient.java in a new com.ttdev.ss.lowlevel package:

```java
import org.apache.axiom.om.OMElement;
import org.apache.axis2.AxisFault;
import org.apache.axis2.addressing.EndpointReference;
import org.apache.axis2.client.Options;
import org.apache.axis2.client.ServiceClient;

public class LowLevelClient {
    public static void main(String[] args) throws AxisFault {
        ServiceClient client = new ServiceClient();
        Options options = new Options();
        options.setTo(new EndpointReference(
            "http://localhost:8080/axis2/services/SimpleService"));
        client.setOptions(options);
        OMElement request = makeRequest();
        OMElement response = client.sendReceive(request);
        System.out.println(response.toString());
    }
}
```

Define the makeRequest() method:
Now run it and it should work. This low level API is called AXIOM (Axis2 Object Model). Usually it is far easier to use the generated stub. However, if you need to do some special customizations, you may have to use AXIOM.

### Seeing the SOAP messages

Next, let's see the actual SOAP messages. To do that, you'll use a program called "TCP Monitor". It works like this (see the diagram below). You tell the client to treat the TCP Monitor as the destination. Then when the client needs to send the request message, it will send it to the TCP Monitor. Then TCP Monitor will print it to the console and then forward it to the real destination (the web service). When the web service returns a response message, it will return it to the TCP Monitor. It will print it to the console and then forward it to the client:
To implement this idea, go to http://ws.apache.org/commons/tcpmon to download the binary distribution of TCP Monitor. Suppose that it is tcpmon-1.0-bin.zip. Unzip it into say c:\tcpmon. Then change into the c:\tcpmon\build folder and run tcpmon.bat:

```
C:\>cd tcpmon\build
C:\tcpmon\build> tcpmon.bat
```

Note that directly running c:\tcpmon\build\tcpmon.bat will NOT work; it requires the current folder to be c:\tcpmon\build. Next, you'll see a window. Enter the data as shown below:
Click "Add". This will open a new tab (shown below). Then it will listen on port 1234. Check the "XML Format" option. This way it will format the content of the TCP connection (an HTTP request containing a SOAP request, but it doesn't know that) nicely as XML:
Chapter 4  Understanding the calling process

For the client, you need to tell it to use localhost:1234 as the endpoint. For example, in LowLevelClient.java:

```java
public class LowLevelClient {
    public static void main(String[] args) throws AxisFault {
        ServiceClient client = new ServiceClient();
        Options options = new Options();
        options.setTo(new EndpointReference("http://localhost:80801234/axis2/services/SimpleService"));
        client.setOptions(options);
        OMElement request = makeRequest();
        OMElement response = client.sendReceive(request);
        System.out.println(response.toString());
    }
}
```

Run it and you will see the messages in TCP Monitor:
Similarly, for the SimpleClient that is using the generated client stub, you can specify the endpoint address to override the default:

```java
class SimpleClient {
    public static void main(String[] args) throws RemoteException {
        SimpleServiceStub service = new SimpleServiceStub(
            "http://localhost:1234/axis2/services/SimpleService");
        ConcatRequest request = new ConcatRequest();
        request.setS1("abc");
        request.setS2("123");
        ConcatResponse response = service.concat(request);
        System.out.println(response.getConcatResponse());
    }
}
```

**Summary**

To call a web service without using a generated stub, you may use the AXIOM interface. It is a lower level interface and thus is harder to use, but it provides a
lot of flexibility.

To check the SOAP messages, you can use the TCP Monitor.
Chapter 5
Accepting multiple parameters
What's in this chapter?

In this chapter you'll learn how to accept multiple parameters in your implementation class.

Accepting multiple parameters

Consider the SimpleServiceImpl class:

```java
public class SimpleServiceImpl implements SimpleServiceSkeletonInterface {
    public ConcatResponse concat(ConcatRequest concatRequest0) {
        String result = concatRequest0.getS1() + concatRequest0.getS2();
        ConcatResponse response = new ConcatResponse();
        response.setConcatResponse(result);
        return response;
    }
}
```

Because it's a document style web service, you can have a single part in the input message. Therefore, you have a single parameter only. The same is true for the output message. It would be nice if you could write:

```java
public class SimpleServiceImpl implements SimpleServiceSkeletonInterface {
    public String concat(String s1, String s2) {
        return s1 + s2;
    }
}
```

while still accepting a single part (<concatRequest>) in the message. To do that, you just need to make two changes to the WSDL file:
Similarly, for the output message, the element name must be the name of the operation with the word "Response" appended and it must be a sequence (containing a single child element):

```xml
<wsdl:definitions ...>
  <wsdl:types>
    <xsd:schema ...>
      <xsd:element name="concatResponse" type="xsd:string" />
    </xsd:schema>
    <xsd:element name="concatRequest concat">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="s1" type="xsd:string" />
          <xsd:element name="s2" type="xsd:string" />
        </xsd:sequence>
      </xsd:complexType>
      <xsd:element name="concatResponse" type="xsd:string" />
    </xsd:element>
    <xsd:element name="concatResponse" type="xsd:string" />
  </xsd:schema>
  <wsdl:message name="concatRequest">
    <wsdl:part name="parameters" element="tns:concatRequest concat" />
  </wsdl:message>
  <wsdl:message name="concatResponse">
    <wsdl:part name="parameters" element="tns:concatResponse" />
  </wsdl:message>
  <wsdl:portType name="SimpleService">
    <wsdl:operation name="concat">
      <wsdl:input message="tns:concatRequest" />
      <wsdl:output message="tns:concatResponse" />
    </wsdl:operation>
  </wsdl:portType>
  ...
</wsdl:definitions>
```

The element must be a sequence, which is indeed the case here.
To test it, copy the SimpleService project and paste it as WrappedService. Delete all the Java files. The "out" folder is still linking to the old location (c:\axis\repository\services\SimpleService). So go to the Navigator view in Eclipse and open the .project file:

Choose the Navigator view

Edit the .project file

Then change the path to c:\axis\repository\services\WrappedService:
Rename SimpleService.wsdl to WrappedService.wsdl and modify it:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<projectDescription>
  <name>WrappedService</name>
  <comment></comment>
  <projects>
    </projects>
  <buildSpec>
    <buildCommand>
      <name>org.eclipse.jdt.core.javabuilder</name>
      <arguments>
        </arguments>
    </buildCommand>
    </buildSpec>
  <natures>
    <nature>org.eclipse.jdt.core.javannature</nature>
  </natures>
  <linkedResources>
    <link>
      <name>out</name>
      <type>2</type>
      <location>C:/axis/repository/services/WrappedService</location>
    </link>
  </linkedResources>
</projectDescription>
```

```
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://ttdev.com/ss"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="WrappedService"
targetNamespace="http://ttdev.com/ss">
  <wsdl:types>
    <xsd:schema targetNamespace="http://ttdev.com/ss"
                xmlns:xs="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="concat">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="s1" type="xsd:string" />
            <xsd:element name="s2" type="xsd:string" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="concatResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="r" type="xsd:string" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="concatRequest">
    <wsdl:part name="parameters" element="tns:concat" />
  </wsdl:message>
  <wsdl:message name="concatResponse">
    <wsdl:part name="parameters" element="tns:concatResponse" />
```
Chapter 5 Accepting multiple parameters

Modify build.xml:

```xml
<project basedir="." default="jar.server">
...
<target name="generate-service">
  <wsdl2code
    wsdlfilename="SimpleService${name}.wsdl"
    serverside=true
    generateservicexml=true
    skipbuildxml=true
    serversideinterface=true
    namespacetopackages="http://ttdev.com/ss=com.ttdev.wrap"
    targetsourcefolderlocation="src"
    targetresourcesfolderlocation="src/META-INF"
    overwrite="true"/>
  <replaceregexp
    file="src/META-INF/services.xml"
    match="SimpleService${name}Skeleton"
    replace="SimpleService${name}Impl"/>
</target>
<target name="generate-client">
  <wsdl2code
    wsdlfilename="SimpleService${name}.wsdl"
    skipbuildxml=true
    namespacetopackages="http://ttdev.com/ss=com.ttdev.wrap.client"
    targetsourcefolderlocation="src"
    overwrite="true"/>
</target>
</project>
```

There is a property telling the name of the project
Refer to the property
Put the code into another package
Refer to the property

Next is an important step: You need a service stub that performs some special processing (see the diagram below). When an incoming <concat> element
arrives, the service stub will extract the <s1> and <s2> elements from the <concat> element and use them as values for the two parameters ("unwrapping"). When the service implementation returns a string, the stub will use it as the value for the <r> element and put the <r> element into a <concatResponse> element ("wrapping"): 

```
Service stub
```

1: An incoming <concat> element arrives

```
<concat>
  <s1>abc</s1>
  <s2>123</s2>
</concat>
```

2: Extract <s1>, from <concat> use it as parameter s1. Do the same thing for <s2>. This is called "unwrapping".

```
String concat(String s1, String s2) {
  return "xyz";
}
```

3: Use the return value as element <r>. Put <r> into <concatResponse>. This is called "wrapping".

```
<concatResponse>
  <r>abc</r>
</concatResponse>
```

Note that this service is still a 100% document style service. The clients can still call it the same way (except that <concatRequest> is changed to <concat>). The difference is how the service stub calls your implementation and how it handles your return value. There is no difference seen by the client. To generate such a service stub, add an option to the <wsdl2code> Ant task:
Accepting multiple parameters

Run build.xml to generate the service stub and client stub. BUG ALERT: In Axis2 1.3 there is a bug preventing <wsdl2code> to overwrite the services.xml file. So, delete it first before running build.xml. Refresh the project. Check the WrappedServiceSkeleton.java:

```java
public class WrappedServiceSkeleton implements WrappedServiceSkeletonInterface {
    public String concat(String s11, String s22) {
        ...  
    }
}
```

To see it working, create a WrappedServiceImpl class:

```java
public class WrappedServiceImpl implements WrappedServiceSkeletonInterface {
    public String concat(String s1, String s2) {
        return s1 + s2;
    }
}
```

Start the Axis server. Create a WrappedClient.java in the client package:
public class WrappedClient {
    public static void main(String[] args) throws RemoteException {
        WrappedServiceStub wrappedService = new WrappedServiceStub();
        String result = wrappedService.concat("xyz", "111");
        System.out.println(result);
    }
}

Run it and it should work.

**Interoperability**

The wrapped convention is a good idea. It is the only kind of web service supported by the .NET framework. Obviously Axis has also implemented this convention. The good news is, from the viewpoint of the caller, it is just a document+literal style service. So if the caller doesn't understand the wrapped convention, it can still access it as a regular document style service.

**Summary**

You can use the wrapped convention support in <wsdl2code> so that your back end Java method can have multiple parameters. The clients understanding this convention can also call it using multiple parameters. For those not understanding it, they can still call it as a regular document style service.

To ensure interoperability with .NET, you should use this convention.
Chapter 6

Sending and receiving complex data structures
What's in this chapter?

In this chapter you'll learn how to send and receive complex data structures to and from a web service.

Product query

Suppose that your company would like to use web service to let your customers query the product availability and place orders with you. For this you need to discuss with them to decide on the interface. It doesn't make sense to say that "When doing query, please send me an object of such a Java class. In this class there are this and that fields..." because perhaps the people involved aren't programmers or don't use Java. Instead, XML is what is designed for this. It is platform neutral and programming language neutral. So, suppose that you all agree on the following schema:

```
<?xml version="1.0"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://foo.com">
    <element name="productQuery">
        <complexType>
            <sequence>
                <element name="queryItem" minOccurs="1" maxOccurs="unbounded">
                    <complexType>
                        <attribute name="productId" type="string"/>
                        <attribute name="qty" type="int"/>
                    </complexType>
                </element>
            </sequence>
        </complexType>
    </element>
</schema>
```

A `productQuery` contains one or more `queryItem` elements. Here is an example:

```
<?xml version="1.0"?>
<foo:productQuery xmlns:foo="http://foo.com">
    <queryItem productId="p01" qty="100"/>
    <queryItem productId="p02" qty="200"/>
    <queryItem productId="p03" qty="500"/>
</foo:productQuery>
```

Use the XML schema namespace as the default namespace. It defines elements such as `<element>`, `<complexType>` needed for you to define new elements.

The string type and int type are defined in the XML schema. They are usually shown as xsd:string and xsd:int, but the XML schema namespace here is the default namespace, so no prefix is needed.

That is, when they need to find out the availability of some products, they will send you a `<productQuery>` element. For example if they'd like to check if you
have 100 pieces of p01, 200 pieces of p02 and 500 pieces of p03, they may send you a request like this:

```
<foo:productQuery
    xmlns:foo="http://foo.com">
    <queryItem productId="p01" qty="100"/>
    <queryItem productId="p02" qty="200"/>
    <queryItem productId="p03" qty="500"/>
</foo:productQuery>
```

How does your web service reply? Use an XML element of course. So, in the schema you may have:

```
<?xml version="1.0"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:tns="http://foo.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="BizService"
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://foo.com">
    <element name="productQuery">
        ...
    </element>
    <element name="productQueryResult">
        <complexType>
            <sequence>
                <element name="resultItem" minOccurs="1" maxOccurs="unbounded">
                    <complexType>
                        <attribute name="productId" type="string"/>
                        <attribute name="price" type="int"/>
                    </complexType>
                </element>
            </sequence>
        </complexType>
    </element>
</schema>
```

For each <queryItem>, if the product is available, create a <resultItem> telling the unit price.

```
<foo:productQueryResult
    xmlns:foo="http://foo.com">
    <resultItem productId="p01" price="5"/>
    <resultItem productId="p03" price="8"/>
</foo:productQueryResult>
```

So, for the sample query above, if you have over 100 pieces of p01 and 500 pieces of p03 but only 150 pieces of p02, and you're willing to sell p01 at 5 dollars each and p03 at 8 dollars each, you may reply:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:tns="http://foo.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="BizService"
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://foo.com">
    <element name="productQuery">
        ...
    </element>
    <element name="productQueryResult">
        <complexType>
            <sequence>
                <element name="resultItem" minOccurs="1" maxOccurs="unbounded">
                    <complexType>
                        <attribute name="productId" type="string"/>
                        <attribute name="price" type="int"/>
                    </complexType>
                </element>
            </sequence>
        </complexType>
    </element>
</wsdl:definitions>
```

To implement this idea, create a new project named BizService as usual (You may copy an old one). Make sure the "out" folder links to c:\axis\repository\services\BizService. Delete the existing WSDL file and create a BizService.wsdl file (use Eclipse or manually):

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:tns="http://foo.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="BizService"
    xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://foo.com">
    <element name="productQuery">
        ...
    </element>
    <element name="productQueryResult">
        <complexType>
            <sequence>
                <element name="resultItem" minOccurs="1" maxOccurs="unbounded">
                    <complexType>
                        <attribute name="productId" type="string"/>
                        <attribute name="price" type="int"/>
                    </complexType>
                </element>
            </sequence>
        </complexType>
    </element>
</wsdl:definitions>
```
targetNamespace="http://foo.com">
<wsdl:types>
  <xsd:schema targetNamespace="http://foo.com"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="productQuery">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="queryItem" maxOccurs="unbounded" minOccurs="1">
            <xsd:complexType>
              <xsd:attribute name="productId" type="xsd:string" />
              <xsd:attribute name="qty" type="xsd:int" />
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="productQueryResult">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="resultItem" maxOccurs="unbounded" minOccurs="1">
            <xsd:complexType>
              <xsd:attribute name="productId" type="xsd:string" />
              <xsd:attribute name="price" type="xsd:int" />
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</wsdl:types>

<wsdl:message name="queryRequest">
  <wsdl:part name="parameters" element="tns:productQuery" />
</wsdl:message>

<wsdl:message name="queryResponse">
  <wsdl:part name="parameters" element="tns:productQueryResult" />
</wsdl:message>

<wsdl:portType name="BizService">
  <wsdl:operation name="query">
    <wsdl:input message="tns:queryRequest" />
    <wsdl:output message="tns:queryResponse" />
  </wsdl:operation>
</wsdl:portType>

<wsdl:binding name="BizServiceSOAP" type="tns:BizService">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http" />
  <soap:operation soapAction="http://foo.com/NewOperation" />
  <wsdl:input>
    <soap:body use="literal" />
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal" />
  </wsdl:output>
</wsdl:binding>

<wsdl:service name="BizService">
  <wsdl:port binding="tns:BizServiceSOAP" name="BizServiceSOAP">
    <soap:address location="http://localhost:8080/axis2/services/BizService" />
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>

If you edit it visually, here are the key steps: First, rename the operation to "query". The input element is automatically renamed to <query>. Double click on
the arrow to right of the <query> element in order to edit it. Then right click on it and choose "Refactor | Rename":

Rename it to "productQuery":

Rename the "in" element to "queryItem":

For the moment it is a string. Right click on it and choose "Set Type | New":
Chapter 6  Sending and receiving complex data structures

Choose to create an anonymous local complex type:

It will be like:

Next, you’d like to edit the (queryItemType). But clicking on it will NOT allow you to edit it. Instead, it will only let you choose another type for <queryItem>: 
This is because Eclipse will not allow you to directly edit something too deep. Instead, it requires you to drill down by one level. So, double click on (productQueryType) [Note: NOT (queryItemType)] to drill down. You’ll see that the (queryItemType) is available for editing:

Right click on (queryItemType) and choose "Add Attribute":

Rename the attribute to "productId". The type is by default string which is what you want:

Similarly, add another attribute "qty" and set its type to int:
To tell that there can be 1 to many <queryItem> elements, right click the <queryItem> element and choose "Set Multiplicity | 1..*":

You'll see:

Now, it is done. To return to one level up, click the left arrow icon as if it were a browser:
Similarly, create the <productQueryResult> element. As usual, validate it when you're done.

Next, update the build.xml file:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="jar.server">
  ...
  <target name="generate-service">
    <wsdl2code
      wsdlfilename="${name}.wsdl"
      serverside="true"
      generateservicexml="true"
      skipbuildxml="true"
      serversideinterface="true"
      namespacepackages="http://ttdev.com/ss=com.ttdev.wrapped"
      namespacepackages="http://foo.com=com.ttdev.biz"
      targetsourcefolderlocation="src"
      targetresourcesfolderlocation="src/META-INF"
      overwrite="true"
      unwrap="true"/>
    <replaceregexp
      file="src/META-INF/services.xml"
      match="${name}Skeleton"
      replace="${name}Impl"/>
  </target>
  
  <target name="generate-client">
    <wsdl2code
      wsdlfilename="${name}.wsdl"
      namespacepackages="http://ttdev.com/ss=com.ttdev.wrapped.client"
      namespacepackages="http://foo.com=com.ttdev.biz.client"
      targetsourcefolderlocation="src"
      overwrite="true"
      unwrap="true"/>
  </target>
</project>
```

Generate the service stub and client stub. BUG ALERT: In Axis2 1.3 there is a bug preventing <wsdl2code> to overwrite the services.xml file. So, delete it first before running build.xml. Then create a BizServiceImpl class in the com.ttdev.biz package:
If you inspect the ProductQuery class and the ProductQueryResult class, you'll note the mapping is like this:

```java
public class BizServiceImpl implements BizServiceSkeletonInterface {
    public ProductQueryResult query(ProductQuery productQuery) {
    }
}
```

Let Eclipse add the unimplemented methods.

XML elements are mapped to Java classes

```xml
<xsd:schema ...>
    <xsd:element name="productQuery">
        ...
    </xsd:element>
    <xsd:element name="productQueryResult">
        ...
    </xsd:element>
</xsd:schema>
```

If you inspect the ProductQuery class and the ProductQueryResult class, you'll note the mapping is like this:

- Each element in a sequence is mapped to a field in the class.
- An element that can occur multiple times (i.e., `maxOccurs > 1`) is mapped to a Java array.
- Attributes are also mapped to fields, just like elements in a sequence.
- The type for `<queryItem>` is mapped to a Java class. But why the class is not simply named QueryItem? Why the `_type0` suffix? This is because the element is a local element and therefore it is possible for another top level element to contain another local `<queryItem>` element as shown below. Then that could be mapped to `QueryItem_type1`.

```xml
<xsd:element name="userQuery">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="queryItem" maxOccurs="unbounded" minOccurs="1">
                ...attr productId, qty...
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Then fill in the code to complete the implementation:
public class BizServiceImpl implements BizServiceSkeletonInterface {
    public ProductQueryResult query(ProductQuery productQuery) {
        ProductQueryResult result = new ProductQueryResult();
        QueryItem_type0[] queryItems = productQuery.getQueryItem();
        for (int i = 0; i < queryItems.length; i++) {
            QueryItem_type0 queryItem = queryItems[i];
            if (queryItem.getQty() <= 200) {
                ResultItem_type0 resultItem = new ResultItem_type0();
                resultItem.setProductId(queryItem.getProductId());
                resultItem.setPrice(20);
                result.addResultItem(resultItem);
            }
        }
        return result;
    }
}

Deploy it. Create a BizClient.java in the com ttdev biz client package:

```java
public class BizClient {
    public static void main(String[] args) throws RemoteException {
        BizServiceStub bizService = new BizServiceStub();
        ProductQuery query = new ProductQuery();
        QueryItem_type0 queryItem1 = new QueryItem_type0();
        queryItem1.setProductId("p01");
        queryItem1.setQty(100);
        query.addQueryItem(queryItem1);
        QueryItem_type0 queryItem2 = new QueryItem_type0();
        queryItem2.setProductId("p02");
        queryItem2.setQty(200);
        query.addQueryItem(queryItem2);
        QueryItem_type0 queryItem3 = new QueryItem_type0();
        queryItem3.setProductId("p03");
        queryItem3.setQty(500);
        query.addQueryItem(queryItem3);
        ProductQueryResult result = bizService.query(query);
        for (ResultItem_type0 resultItem : result.getResultItem()) {
            System.out.println(resultItem.getProductId() + ": " + resultItem.getPrice());
        }
    }
}
```

Run the client and it should work:

```
Avoiding the type suffix
```
If you don’t like the type suffixes like _type0, you can turn the type for <queryItem> into a top level type. To do that, right click (queryItemType) and choose “Refactor | Make Anonymous Type Global”:

```
<xsd:schema ...>
  <xsd:element name="productQuery">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="queryItem" 
          minOccurs="1" 
          maxOccurs="unbounded" 
          type="tns:queryItemComplexType">
        <xsd:element>
          <xsd:sequence>
            </xsd:element>
        </xsd:complexType>
      </xsd:element>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

The WSDL code will become:

```
<xsd:element name="productQuery">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="queryItem" 
        minOccurs="1" 
        maxOccurs="unbounded" 
        type="tns:queryItemComplexType">
      </xsd:element>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

"type" means that this element conforms to an existing type

Rename the type from queryItemComplexType to queryItemType:
Generate the service code and client code again. The QueryItem_type0 class will be gone and you'll have a QueryItemType class instead. You'll need to update your code accordingly:

```java
public class BizServiceImpl implements BizServiceSkeletonInterface {
    public ProductQueryResult query(ProductQuery productQuery) {
        ProductQueryResult result = new ProductQueryResult();
        for (int i = 0; i < queryItems.length; i++) {
            QueryItem_type0 queryItem = queryItems[i];
            if (queryItem.getQty() <= 200) {
                ResultItem_type0 resultItem = new ResultItem_type0();
                resultItem.setProductId(queryItem.getProductId());
                resultItem.setPrice(20);
                result.addResultItem(resultItem);
            }
        }
        return result;
    }
}
```

Make similar changes to the BizClient class. Run it and it should continue to work.

### Sending more data in a message

By the way, this query operation demonstrates a good practice in web services: You generally hope to send more data in a message. For example, you may be sending many query items in a single response message. This is more efficient than sending a single query item object in a message. This is because there is a certain overhead involved in sending a message, even if it contains no data:
Returning faults

Suppose that a client is calling your query operation but a product id is invalid (not just out of stock, but absolutely unknown) or the quantity is zero or negative. You may want to throw an exception. To return an exception to the client, you send a “fault message”, which is very much like an output message. To do that, modify the WSDL file:
Chapter 6  Sending and receiving complex data structures

How to include the fault message in a SOAP message? It is included in the SOAP body, but not directly:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ...>
  <wsdl:types>
    <xsd:schema ...>
      <xsd:element name="productQuery"/>
      <xsd:element name="productQueryResult"/>
      <xsd:complexType name="queryItemType">
        <xsd:element name="invalidProductId" type="xsd:string"/>
        <xsd:element name="invalidQty" type="xsd:int"/>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="queryRequest">
    <wsdl:part name="parameters" element="tns:productQuery"/>
  </wsdl:message>
  <wsdl:message name="queryResponse">
    <wsdl:part name="parameters" element="tns:productQueryResult"/>
  </wsdl:message>
  <wsdl:message name="queryInvalidProductId">
    <wsdl:part name="parameters" element="tns:invalidProductId"/>
  </wsdl:message>
  <wsdl:message name="queryInvalidQty">
    <wsdl:part name="parameters" element="tns:invalidQty"/>
  </wsdl:message>
  <wsdl:portType name="BizService">
    <wsdl:operation name="query">
      <wsdl:input message="tns:queryRequest"/>
      <wsdl:output message="tns:queryResponse"/>
      <wsdl:fault name="f01" message="tns:queryInvalidProductId"/>
      <wsdl:fault name="f02" message="tns:queryInvalidQty"/>
    </wsdl:operation>
  </wsdl:portType>
  ...
</wsdl:definitions>
```

A fault message is like an output message, but it indicates an error. The one and only part is a well defined element in the schema. Unlike an input or output message which doesn't need a name, a fault needs a unique name because there can be multiple fault messages (here you have 2). Later you'll refer to a fault using its name.
Chapter 6  Sending and receiving complex data structures

The SOAP <Fault> element tells the caller that something is wrong. The <faultcode> is a QName acting as an error code. The <faultstring> is an error message for human reading. The <detail> will contain any information that both sides agree on. In this case, it contains your fault message part.

To make the above changes to the WSDL file visually, right click the query operation and choose "Add Fault";

```xml
<wsdl:definitions ...>
  ...
  <wsdl:portType name="BizService">
    <wsdl:operation name="query">
      <wsdl:input message="tns:queryRequest" />
      <wsdl:output message="tns:queryResponse" />
      <wsdl:fault name="f01" message="tns:queryInvalidProductId" />
      <wsdl:fault name="f02" message="tns:queryInvalidQty" />
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="BizServiceSOAP" type="tns:BizService">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="query">
      <soap:operation soapAction="http://foo.com/NewOperation" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" />
      </wsdl:output>
      <wsdl:fault name="f01 " use="literal"/>
      <wsdl:fault name="f02 " use="literal"/>
    </wsdl:operation>
  </wsdl:binding>
  ...
</wsdl:definitions>
```

The SOAP <Fault> element tells the caller that something is wrong. The <faultcode> is a QName acting as an error code. The <faultstring> is an error message for human reading. The <detail> will contain any information that both sides agree on. In this case, it contains your fault message part.

To make the above changes to the WSDL file visually, right click the query operation and choose "Add Fault":

```xml
<soap-env:Envelope
  xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
  <soap-env:Header>
    ...
  </soap-env:Header>
  <soap-env:Body>
    <soap-env:Fault>
      <soap-env:faultcode>...</soap-env:faultcode>
      <soap-env:faultstring>...</soap-env:faultstring>
      <soap-env:detail>
        <foo:invalidProductId xmlns:foo="http://foo.com">
          p1000
        </foo:invalidProductId>
      </soap-env:detail>
    </soap-env:Fault>
  </soap-env:Body>
</soap-env:Envelope>
```
Choose the fault, in the Properties window, set its name to f01:

Choose to create a new message:

Enter the name for the message:

Set the one and only part to a new XML element <invalidProductId>. By default it should be of type xsd:string which is what you want here. Create the second fault similarly. Set the message name to queryInvalidQty, set the XML element to <invalidQty> whose type is xsd:int. Finally it should be like:
Next, create the binding for the two faults. Choose the binding and click "Generate Binding Content" in the Properties window:

Check "Overwrite existing binding information" and then click "Finish":

Check "Overwrite existing binding information" and then click "Finish":
This will generate the binding portion:

```xml
<wsdl:binding name="BizServiceSOAP" type="tns:BizService">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="query">
        <soap:operation soapAction="http://foo.com/query" />
        <wsdl:input>
            <soap:body use="literal" />
        </wsdl:input>
        <wsdl:output>
            <soap:body use="literal" />
        </wsdl:output>
        <wsdl:fault name="f01">
            <soap:fault use="literal" name="f01" />
        </wsdl:fault>
        <wsdl:fault name="f02">
            <soap:fault use="literal" name="f02" />
        </wsdl:fault>
    </wsdl:operation>
</wsdl:binding>
```

Finally go into the schema index to delete the unused elements created by Eclipse:
Similarly, choose "Window | Show View | Outline" to show the outline of the WSDL file as shown below. Right click and delete the unused messages such as query_faultMsg and query_faultMsg1:

Now, generate the service and client stubs and refresh the files in Eclipse. You will find some new Java classes:
The method signature in BizServiceSkeletonInterface has also been updated to throw such exceptions:

```java
public interface BizServiceSkeletonInterface {
    public ProductQueryResult query(ProductQuery productQuery)
        throws QueryInvalidProductId, QueryInvalidQty;
}
```

Now modify your implementation code:

```java
public class BizServiceImpl implements BizServiceSkeletonInterface {
    public ProductQueryResult query(ProductQuery productQuery)
        throws QueryInvalidProductId, QueryInvalidQty {
        ProductQueryResult result = new ProductQueryResult();
        QueryItemType[] queryItems = productQuery.getQueryItem();
        for (int i = 0; i < queryItems.length; i++) {
            QueryItemType queryItem = queryItems[i];
            if (!queryItem.getProductId().startsWith("p")) {
                QueryInvalidProductId fault = new QueryInvalidProductId();
                InvalidProductId part = new InvalidProductId();
                part.setInvalidProductId(queryItem.getProductId());
                fault.setFaultMessage(part);
                throw fault;
            }
            if (queryItem.getQty() <= 0) {
                QueryInvalidQty fault = new QueryInvalidQty();
                InvalidQty part = new InvalidQty();
                part.setInvalidQty(queryItem.getQty());
                fault.setFaultMessage(part);
                throw fault;
            }
            if (queryItem.getQty() <= 200) {
                ResultItem_type0 resultItem = new ResultItem_type0();
                resultItem.setProductId(queryItem.getProductId());
                resultItem.setPrice(20);
                result.addResultItem(resultItem);
            }
        }
        return result;
    }
}
```

To see if it's working, modify BizClient.java:

```java
public class BizClient {
    class QueryInvalidProductId extends Exception {
        InvalidProductID faultMessage;
        ...
    }
    class InvalidProductID {
        String invalidProductID;
        ...
    }
    class QueryInvalidQty extends Exception {
        InvalidQty faultMessage;
        ...
    }
    class InvalidQty {
        int invalidQty;
        ...
    }
    A fault message is mapped to a Java exception. Its one and only part (an XML element) is mapped to a field.
    As usual, an XML element such as the <invalidProductId> element is mapped to a Java class. It wanted to extend String, but String is a final class. So the string is mapped to a field.
    The method signature in BizServiceSkeletonInterface has also been updated to throw such exceptions:
    public interface BizServiceSkeletonInterface {
        public ProductQueryResult query(ProductQuery productQuery)
            throws QueryInvalidProductId, QueryInvalidQty;
    }
    Now modify your implementation code:
    public class BizServiceImpl implements BizServiceSkeletonInterface {
        public ProductQueryResult query(ProductQuery productQuery)
            throws QueryInvalidProductId, QueryInvalidQty {
            ProductQueryResult result = new ProductQueryResult();
            QueryItemType[] queryItems = productQuery.getQueryItem();
            for (int i = 0; i < queryItems.length; i++) {
                QueryItemType queryItem = queryItems[i];
                if (!queryItem.getProductId().startsWith("p")) {
                    QueryInvalidProductId fault = new QueryInvalidProductId();
                    InvalidProductID part = new InvalidProductID();
                    part.setInvalidProductId(queryItem.getProductId());
                    fault.setFaultMessage(part);
                    throw fault;
                }
                if (queryItem.getQty() <= 0) {
                    QueryInvalidQty fault = new QueryInvalidQty();
                    InvalidQty part = new InvalidQty();
                    part.setInvalidQty(queryItem.getQty());
                    fault.setFaultMessage(part);
                    throw fault;
                }
                if (queryItem.getQty() <= 200) {
                    ResultItem_type0 resultItem = new ResultItem_type0();
                    resultItem.setProductId(queryItem.getProductId());
                    resultItem.setPrice(20);
                    result.addResultItem(resultItem);
                }
            }
            return result;
        }
    }
    To see if it's working, modify BizClient.java:
    public class BizClient {
        ```
public static void main(String[] args) throws RemoteException {
    BizServiceStub bizService = new BizServiceStub();
    ProductQuery query = new ProductQuery();
    QueryItemType queryItem = new QueryItemType();
    queryItem.setProductId("p01");
    query.addQueryItem(queryItem);
    queryItem = new QueryItemType();
    queryItem.setProductId("p02");
    queryItem.setQty(-200);
    query.addQueryItem(queryItem);
    queryItem = new QueryItemType();
    queryItem.setProductId("p03");
    queryItem.setQty(500);
    query.addQueryItem(queryItem);

    try {
        ProductQueryResult result = bizService.query(query);
        for (ResultItem resultItem : result.getResultItem()) {
            System.out.println(resultItem.getProductId() + ": " + resultItem.getPrice());
        }
    } catch (QueryInvalidProductId e) {
        System.out.println("Invalid product id: " + e.getFaultMessage().getInvalidProductId());
    } catch (QueryInvalidQty e) {
        System.out.println("Invalid qty: " + e.getFaultMessage().getInvalidQty());
    }
}

Start the Axis server, then run the BizClient and it should work:

If you'd like, you can see the messages in TCP Monitor:

```xml
<soap-env:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <soap-env:Fault>
      <faultcode>soap-env:Server</faultcode>
      <faultstring>QueryInvalidQty</faultstring>
      <detail>
        <ns1:invalidQty xmlns:ns1="http://foo.com">-200</ns1:invalidQty>
      </detail>
    </soap-env:Fault>
  </soapenv:Body>
</soapenv:Envelope>
```

Using encoded

You have been writing document style services. In addition, the parts are sent as "literal":

```xml
<wSDL:binding name="BizServiceSOAP" type="tns:BizService">
  <soap:binding style="document"
```
What does literal means? If you don't use literal, you may set it to "encoded". Then Axis will perform some extra encoding of the data in order to convert it into XML. For example, it will be able to handle multi-dimension arrays and data structures containing loops (e.g., a circular linked-list). These kind of data structures don't have direct counter-parts in XML. In fact, if you start from a WSDL, you will never get these data types from the <wsdl2code> Ant task. So, "encoded" is useful only when you have some legacy code that uses such data structures and you'd like to expose it as a web service.

The resulting XML is XML but can't be validated by any schema. This is prohibited in document style services. Therefore, in order to use "encoded", you must use the RPC style.

To use RPC+encoded, in theory you only need to change the WSDL and then generate the stubs again. However, as of Axis2 1.3, Axis2 doesn't support the encoded use as it is not good for interoperability and is getting phased out (in the next version of WSDL, namely WSDL 2.0, only document+literal is supported).

**Referring to existing XML elements**

For the moment you're defining XML elements such as <productQuery> directly in the WSDL file. However, in practice, most likely such elements are defined by a 3rd party such as an industrial consortium or neutral association. Suppose that they are provided in a file purchasing.xsd such as this:
How to refer to those XML elements in your WSDL file? First, put the purchasing.xsd file into the same folder as the WSDL file (i.e., the project root). Then modify the WSDL file:
You're saying: I'd like to refer to the XML elements defined in the http://bar.org/purchasing namespace. Then the XML elements will be visible to this WSDL file. This is like the import statement in Java used to import a package or a class.

How can the WSDL parser find out the XML elements defined there? It will work if the person parsing the WSDL have set up a table like below. Such a table is called an XML catalog.

<table>
<thead>
<tr>
<th>Namespace</th>
<th>Path to its xsd file</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://bar.org/purchasing">http://bar.org/purchasing</a></td>
<td>c:\schema\f1.xsd</td>
</tr>
<tr>
<td>http://...</td>
<td>c:...</td>
</tr>
</tbody>
</table>

As you'll be giving away this WSDL to many people, it may be too difficult to ask everyone to set up the XML catalog. So you may simply distribute the XSD file and make sure it is in the same folder as the WSDL file and specify the relative path here. In addition to the XML catalog, their WSDL processor will follow this path to find the XSD file.

The elements are now defined in another namespace.

Modify build.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:tns="http://foo.com"
xmlns:p="http://bar.org/purchasing"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
tARGET_NAMESPACE="http://foo.com">
<wsdl:types>
<xsd:schema targetNamespace="http://foo.com"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:import
namespace="http://bar.org/purchasing"
schemaLocation="purchasing.xsd">
<xsd:element name="productQuery">
...
</xsd:element>
...
</xsd:schema>
</wsdl:types>
<wsdl:message name="queryRequest">
<wsdl:part name="parameters" element="p:productQuery" />
</wsdl:message>
<wsdl:message name="queryResponse">
<wsdl:part name="parameters" element="p:productQueryResult" />
</wsdl:message>
<wsdl:message name="queryInvalidProductId">
<wsdl:part name="NewPart" element="p:invalidProductId" />
</wsdl:message>
<wsdl:message name="queryInvalidQty">
<wsdl:part name="NewPart" element="p:invalidQty" />
</wsdl:message>
<wsdl:portType name="BizService">
...
</wsdl:portType>
<wsdl:binding name="BizServiceSOAP" type="tns:BizService">
...
</wsdl:binding>
<wsdl:service name="BizService">
...
</wsdl:service>
</wsdl:definitions>

You don't need to define your own elements anymore.

As you'll be giving away this WSDL to many people, it may be too difficult to ask everyone to set up the XML catalog. So you may simply distribute the XSD file and make sure it is in the same folder as the WSDL file and specify the relative path here. In addition to the XML catalog, their WSDL processor will follow this path to find the XSD file.

The elements are now defined in another namespace.

Modify build.xml:
Delete all the Java files generated. That is, all files except your BizServiceImpl and BizClient. Also delete all the files in META-INF. Run build.xml. You should see the following output in the console:

Buildfile: C:\workspace\BizService\build.xml
generate-service:
  [wsdl2code] Retrieving schema at 'purchasing.xsd', relative to 'file:C:\workspace/BizService/'.
generate-client:
  [wsdl2code] Retrieving schema at 'purchasing.xsd', relative to 'file:C:\workspace/BizService/'.
BUILD SUCCESSFUL
Total time: 10 seconds

Refresh the project. Note that the XSD file will have been copied into the META-INF folder to be accessed by potential clients:
The BizServiceImpl class should still be in error as the XML element classes are now in a different package. Fix this. For example, in Eclipse, open the BizServiceImpl file and press Ctrl-Shift-O and then choose the classes in the com.ttdev.biz.purchasing package (do NOT choose those inner classes in the com.ttdev.biz.client.BizServiceStub):

Run the BizClient and it should continue to work.
Retrieving WSDL files using HTTP

To really simulate the client side, it should retrieve the WSDL file using http://localhost:8080/axis2/services/BizService?wsdl instead of a local file. It should also be able to retrieve the XSD file automatically. To verify that, modify build.xml:

```xml
<project ...>
  ...
  <target name="generate-client">
    <wsdl2code
      wsdlfilename="http://localhost:8080/axis2/services/BizService?wsdl"
      skipbuildxml="true"
      namespacesTopackages="http://foo.com=com.ttdev.biz.client"
      targetsourcefolderlocation="src"
      overwrite="true"/>
  </target>
</project>
```

Make sure the Axis server is running. Then run build.xml to generate the client stub again. It should work and display something like that in the console:

```
Buildfile: C:\workspace\BizService\build.xml
generate-client:
  [wsdl2code] Retrieving schema at 'BizService?xsd=xsd0.xsd', relative to 'http://localhost:8080/axis2/services/'.
BUILD SUCCESSFUL
Total time: 7 seconds
```

Run the client and it should continue to work.

Summary

You can freely use XML schema elements to express complex data structures. The `<wsdl2code>` Ant task will translate them into Java types.

For better performance, you should design the interfaces of your web service operations so that more data is sent in a message.

To report an error from your operation, define a message in the WSDL file and use it as a fault message in the operation. Then add a corresponding child element in the SOAP binding to store it into the SOAP Fault element. The fault message should contain one and only one part which is an XML element describing the fault. The `<wsdl2code>` Ant task will map a fault message to a Java exception class and the part as a field. The operation will be mapped to a Java method throwing that exception.

If you need to send weird data structures, you can use RPC+encoded but interoperability will be affected. The encoded use is not supported by Axis2 as of 1.3.

If you have existing XML elements in an XSD file that you’d like to use in a WSDL file, you can use `<import>` to import them. You can specify the relative path to the XSD file so that the WSDL parser can find it.
Chapter 7
Sending binary files
What's in this chapter?

In this chapter you'll learn how to receive and return binary files in your web service.

Providing the image of a product

Suppose that you'd like to have a web service to allow people to upload the image (jpeg) of a product (identified by a product id). The SOAP message may be like:

```xml
<Envelope>
  <Body>
    <uploadImage>
      <productId>p01</productId>
      <image>kdubn87kamindy...</image>
    </uploadImage>
  </Body>
</Envelope>
```

Typically binary data such as the image is encoded using the base64 encoding.

The problem is that the base64 encoded data will be much larger than the binary version. This wastes processing time, network bandwidth and transmission time. In fact, if the image is huge, then many XML parsers may not be able to handle it properly. To solve this problem, instead of always representing an XML document as text, people state that it can be represented as a MIME message. For example, the above XML document (SOAP envelope) can be represented as below without changing its meaning:
To implement this idea, create a new project named ImageService as usual (You may copy an old one. If so, change the linked folder). Modify the WSDL file:

```
Content-Type: Multipart/Related

--MIME_boundary
Content-Type: text/xml

<Envelope>
  <Body>
    <uploadImage>
      <productId>p01</productId>
      <image>
        <xop:Include
          xmlns:xop="http://www.w3.org/2004/08/xop/include"
          href="cid:abc"/>
      </image>
    </uploadImage>
  </Body>
</Envelope>

--MIME_boundary
Content-Type: image/jpeg
Content-ID: abc

...binary data here...

--MIME_boundary
```

This is a MIME message. It can contain multiple parts. Here it contains 2 parts. This MIME message represents the XML document (the SOAP envelope).

- A part that contains the "core" of the XML document as text
- A part that contains binary data (the image)

This is the xop namespace. xop stands for XML-binary optimized packaging.

Refer to the actual data by content id

Binary data is allowed in a MIME part
Chapter 7  Sending binary files

Although this is not required, it uses the wrapped convention. Next, update build.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="jar.server">
...
<property name="name" value="ImageService" />
...
<target name="generate-service">
  <wsdl2code
    wsdlfilename="${name}.wsdl"
    serverside="true"
...}
```

```xml
  It will contain binary data. It is basically to be encoded using base64. Later you will tell Axis to use XOP for it.
  The operation doesn't return anything, so there is no output message.
</target>
</project>
</wsdl:definitions>

Use a urn as the target namespace

```xml
<wSDL:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsd1/"
  xmlns:soap="http://schemas.xmlsoap.org/wsd1/soap/
  xmlns:tns="urn:ttdev.com:service/img"
  targetNamespace="urn:ttdev.com:service/img">
  <wsdl:types>
    <xsd:schema targetNamespace="urn:ttdev.com:service/img"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <xsd:element name="uploadImage">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="productId" type="xsd:string" />
            <xsd:element name="image" type="xsd:base64Binary" />}
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="uploadImageRequest">
    <wsdl:part name="parameters" element="tns:uploadImage" />
  </wsdl:message>
  <wsdl:portType name="ImageService">
    <wsdl:operation name="uploadImage">
      <soap:operation
        soapAction="urn:ttdev.com:service/img/uploadImage" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="ImageServiceSOAP" type="tns:ImageService">
    <soap:binding style="document"
      transport="http://schemas.xmlsoap.org/soap/http" />
    <wsdl:operation name="uploadImage">
      <soap:operation
        soapAction="urn:ttdev.com:service/img/uploadImage" />
      <wsdl:input>
        <soap:body use="literal" />
      </wsdl:input>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="ImageService">
    <wsdl:port binding="tns:ImageServiceSOAP" name="ImageServiceSOAP">
      <soap:address
        location="http://localhost:8080/axis2/services/ImageService" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Although this is not required, it uses the wrapped convention. Next, update build.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="jar.server">
...}
```
Generateservicexml="true"
skipbuildxml="true"
serversideinterface="true"
namespacepackages="urn:ttdev.com:service/img=com.ttdev.image"
targetsourcefolderlocation="src"
targetresourcesfolderlocation="src/META-INF"
overwrite="true"
unwrap="true" />
<replaceregexp
    file="src/META-INF/services.xml"
    match="\${name}Skeleton"
    replace="\${name}Impl" />
</target>
</target name="generate-client">
<wsdl2code
    wsdllfilename="\${name}.wsdl"
    skipbuildxml="true"
    namespacepackages="urn:ttdev.com:service/img=com.ttdev.image.client"
    targetsourcefolderlocation="src"
    overwrite="true"
    unwrap="true" />
</target>
</target>
</project>

Generate the service stub and client stub. Check the implementation class:

```java
public class ImageServiceSkeleton implements ImageServiceSkeletonInterface {
    public void uploadImage(
        java.lang.String productId1,
        javax.activation.DataHandler image2) {
    }
}
```

Note that the binary image data is presented as a DataHandler object. To read the data from it, create an ImageServiceImpl class:

```java
public class ImageServiceImpl implements ImageServiceSkeletonInterface {
    public void uploadImage(String productId, DataHandler image) {
        System.out.println(image.getContentType());
        try {
            InputStream in = image.getInputStream();
            String imageDir = "c:/tmp";
            FileOutputStream out = new FileOutputStream(new File(imageDir, productId));
            try {
                byte buf[] = new byte[1024];
                for (;;) {
                    int noBytesRead = in.read(buf);
                    out.write(buf, 0, noBytesRead);
                    if (noBytesRead < buf.length) {
                        break;
                    }
                }
            } finally {
                out.close();
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Create an ImageClient.java file in the client package:

Start the Axis server (if it is not yet started). Create the c:\tmp folder. Run the
Chapter 7  Sending binary files

Client. Then check c:\tmp and you should find a new file p01 there. You can verify that it's a copy of axis.jpg by opening it in a browser:

To be sure that it is using XOP, use the TCP Monitor. You should see:

```java
import javax.activation.DataHandler;
import javax.activation.DataSource;
import javax.activation.FileDataSource;

public class ImageClient {
    public static void main(String[] args) throws RemoteException {
        ImageServiceStub service = new ImageServiceStub();
        service.getServiceClient().getOptions().setProperty(
            Constants.Configuration.ENABLE_MTOM, "true");
        DataSource source = new FileDataSource("c:/axis/docs/xdocs/1_3/images/axis.jpg");
        DataHandler handler = new DataHandler(source);
        service.uploadImage("p01", handler);
        System.out.println("Done!");
    }
}
```

Create a DataSource object that will read the data from the file. It will also find out the MIME type (image/jpeg in this case) from the file extension (.jpg).

Create a DataHandler object that reads that DataSource object

Critical point: Enable MTOM. MTOM stands for message transmission optimization mechanism. It means the same thing as XOP when it is applied to SOAP messages. The effect is, whenever it needs to send base64 encoded data, it will send it using XOP.
Chapter 7  Sending binary files

Enabling MTOM in the service

For the moment, it is your client that needs to send a file. If it was your web service that needed to do that, you would need to enable MTOM in the service. To do that, modify services.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<serviceGroup>
  <service name="ImageService">
    <messageReceivers>
      <messageReceiver mep="http://www.w3.org/ns/wsdl/in-only"/>
    </messageReceivers>
    <parameter name="ServiceClass">com.ttdev.image.ImageServiceImpl</parameter>
    <parameter name="useOriginalwsdl">true</parameter>
    <parameter name="modifyUserWSDLPortAddress">true</parameter>
    <parameter name="enableMTOM">true</parameter>
    <operation name="uploadImage" mep="http://www.w3.org/ns/wsdl/in-only">
      <actionMapping>urn:ttdev.com:service/img/uploadImage</actionMapping>
    </operation>
  </service>
</serviceGroup>
```

Note that no matter the setting is there or not, the service can always handle incoming messages using MTOM. This setting affects its outgoing messages only.

Interoperability

If you need to send binary files to others, make sure the other side supports

---

**Content-Type:** multipart/related; boundary=BoundaryName

**SOAPAction:** "urn:ttdev.com:service/img/uploadImage"

**User-Agent:** axis2

Host: 127.0.0.1:1134

Transfer-Encoding: chunked

---

**Content-Type:** application/xop+xml; charset=UTF-8; type="text/xml"

Content-Transfer-Encoding: binary

Content-ID: <urn:uuid:60e7b295b9f8f9c5b1195966468539@apache.org>

Refer to the binary data using cid (content id)

The binary data

---
MTOM. For example, for .NET, MTOM is supported with WSE (Web Services Enhancements) 3.0 or later.

**Summary**

XOP stores XML elements that is of the type xsd:base64Binary as MIME parts and represents the whole XML document as a MIME message. When the XML document is a SOAP envelope, it is called MTOM.

To receive a binary file using MTOM, if the receiver is written with Axis2, for maximum interoperability, it can always handle incoming messages using MTOM without any configuration.

To send a binary file using MTOM, enable MTOM in the sender.
Chapter 8

Invoking lengthy operations
What's in this chapter?

What if your web service involves manual processing that could take days to finish? In this chapter you'll learn what the problems are and how to deal with them.

Providing lengthy operations

Suppose that you have a web service that processes business registration requests and that each request must be manually reviewed by a human being before it is approved. Then a business registration number is provided to the client. The problem is that this review process could take days and the web service client will be kept waiting for the HTTP response (assuming it is using SOAP over HTTP):

In that case, the HTTP client code in the client will think something may be wrong in the server. In order to avoid holding up the resources used by the connection, it will time out and terminate the connection. To solve this problem (see the diagram below), you can tell the client to send a request and then immediately listen on a port for incoming connection. On the server side, the web service will immediately return a short response saying that the request has been received for processing (not approved yet), then create a new thread to wait for the manual approval (so that the web service is free to serve other requests). When that thread gets the manual approval, it connects to the client and tells it that it has been approved and tells it the business registration number:

However, in step c above, how does it know the host name and port of the client? Therefore, when the client sends the request (see the diagram below), it could pick a random port and then include its host name and the port number in the reply-to URL and include that URL in a SOAP header entry. This way, the
background thread created by the web service can send the result to that URL. This is very much like having a From address or Reply-To address in an email. This is called "WS-Addressing":

2: When sending the SOAP message, add a header entry to tell the reply-to URL.

```xml
<Envelope>
  <Header>
    <Reply-To>
      http://localhost:3344
    </Reply-To>
  </Header>
  <Body>
    <request>
      ...
    </request>
  </Body>
</Envelope>
```

1: Pick a random port (say 3344) and listen on it

b: Create new thread

c: Send the real result to http://localhost:3344

However, there is still a problem. If the client sends multiple requests to the web service or to different web services, if it opens a new port for each request, then it will use a lot of ports and will waste a lot of resources. Therefore, it will open a single port only and let a single background thread listening on it:
However, if multiple requests were sent, then multiple responses will arrive. Then in step c above, how can the background thread tell the response is for which request?

To solve this problem, when sending the request, the client will generate a unique message ID (e.g., m001) and include it in a header block (see the diagram below). When the web service generates the response message, it will copy the message ID m001 into the <Relates-To> header block. This way, when the background thread receives the response, it knows that it is the response for request m001:
All these <Reply-To>, <Message-ID>, <Relates-To> header blocks are part of the WS-Addressing standard.

Creating the WSDL for business registrations

To implement this idea, create a new project named ManualService as usual (You may copy an old one. If so, change the linked folder). Modify the WSDL file:
To create the `<choice>` visually, right click the (registerResponseType) and choose "Add Choice".
Then it will become:

Right click the <choice> symbol and choose "Add Element Ref":

Then it will look like:
If you have created the `<approved>` and `<rejected>` elements, they may appear for selection, or you can choose "Browse" to select one of them. If you haven’t created them yet, choose "New" to create them.

The rest of the WSDL file is as usual:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="urn:fake.gov.biz/reg"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" name="ManualService"
targetNamespace="urn:fake.gov.biz/reg">
<wsdl:types>
    <xsd:schema targetNamespace="urn:fake.gov.biz/reg"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <xsd:element name="register">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="bizName" type="xsd:string"/>
                    <xsd:element name="ownerId" type="xsd:string"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="registerResponse">
            <xsd:complexType>
                <xsd:choice>
                    <xsd:element ref="tns:approved"></xsd:element>
                    <xsd:element ref="tns:rejected"></xsd:element>
                </xsd:choice>
            </xsd:complexType>
        </xsd:element>
        <xsd:element name="approved" type="xsd:string"></xsd:element>
        <xsd:element name="rejected" type="xsd:string"></xsd:element>
    </xsd:schema>
</wsdl:types>
<wsdl:message name="registerRequest">
    <wsdl:part name="parameters" element="tns:register"/>
</wsdl:message>
<wsdl:message name="registerResponse">
    <wsdl:part name="parameters" element="tns:registerResponse"/>
</wsdl:message>
<wsdl:portType name="ManualService">
    <wsdl:operation name="register">
        <wsdl:input message="tns:registerRequest"/>
        <wsdl:output message="tns:registerResponse"/>
    </wsdl:operation>
</wsdl:portType>
<wsdl:binding name="ManualServiceSOAP" type="tns:ManualService">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
</wsdl:binding>
</wsdl:definitions>
```
Chapter 8 Invoking lengthy operations

```xml
<soap:operation soapAction="urn:fake.gov.biz/reg/register" />
<wsdl:input>
  <soap:body use="literal" />
</wsdl:input>
<wsdl:output>
  <soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="ManualService">
  <wsdl:port binding="tns:ManualServiceSOAP"
    name="ManualServiceSOAP">
    <soap:address
      location="http://localhost:8080/axis2/services/ManualService" />
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>

Next, update build.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="jar.server">
  ...<property name="name" value="ManualService" />
  ...
  <target name="generate-service">
    <wsdl2code
      wsdllfilename="${name}.wsdl"
      serverside="true"
      generateservicexml="true"
      skipbuildxml="true"
      serversideinterface="true"
      namespacetopackages="urn:fake.gov.biz/reg=gov.fake.bizreg"
      targetsourcefolderlocation="src"
      targetresourcesfolderlocation="src/META-INF"
      overwrite="true"
      unwrap="true" />
    <replaceregexp
      file="src/META-INF/services.xml"
      match="${name}Skeleton"
      replace="${name}Impl" />
  </target>
  
  <target name="generate-client">
    <wsdl2code
      wsdllfilename="${name}.wsdl"
      skipbuildxml="true"
      namespaces="urn:fake.gov.biz/reg=gov.fake.bizreg.client"
      targetsourcefolderlocation="src"
      overwrite="true"
      unwrap="true" />
  </target>
</project>
```

Because the response uses a `<choice>`, you can't use the wrap convention anymore. Then, generate the service stub and client stub. All these are pretty standard stuff. The next step is to make the web service create a new thread for lengthy processing.

### Creating a new thread for lengthy processing

In order to let the web service create a new thread to do the lengthy processing, you need to understand the concept of message receiver in Axis. There is a message receiver for each web service. When a request for your web service arrives (see the diagram below), the message receiver will be handed the message. It will check your services.xml file to find out the implementation class
name (gov.fake.bizreg.ManualServiceImpl here). Then it will create an instance of this class, convert XML to Java objects, pass them as parameters to the right method on that object instance. Finally, it converts the return value back to XML and return it in a response:

All these are happening in the same thread by default. Now, you will tell your message receiver to create a new thread to call your implementation class, while returning an "accepted" response at the same time. To do that, you can modify your message receiver, which is the ManualServiceMessageReceiverInOut class generated by the <wsdl2code> Ant task:

```
import org.apache.axis2.AxisFault;
import org.apache.axis2.context.MessageContext;

public class ManualServiceMessageReceiverInOut extends AbstractInOutSyncMessageReceiver {
    public void receive(MessageContext messageCtx) throws AxisFault {
        messageCtx.setProperty(DO_ASYNC, "true");
        super.receive(messageCtx);
    }
    public void invokeBusinessLogic(
        ...
    }  

    This method will perform data decoding and encoding and call your implementation class. Now it will be executed in a new thread.
}
```

Create ManualServiceImpl.java to implement your web service:
Now the message receiver will call your register() method in a new thread. The next step is to work on the client: It should kick start the background thread and include the <Reply-To> and <Message-ID> headers in the request.

Creating an asynchronous client

To create the client, create a BizRegClient.java file in the client package:
To encode the reply-to URL and message ID using the WS-Addressing standard, Axis provides a "module" to do that. This module is named "addressing". You can simply enable ("engage") it.

```java
public class BizRegClient {
    public static void main(String[] args) throws RemoteException {
        ManualServiceStub stub = new ManualServiceStub();
        ServiceClient serviceClient = stub._getServiceClient();
        serviceClient.engageModule("addressing");
        Options options = serviceClient.getOptions();
        options.setUseSeparateListener(true);
        Register request = new Register();
        request.setBizName("Foo Ltd.");
        request.setOwnerId("Kent");
        ManualServiceCallbackHandler callback =
            new ManualServiceCallbackHandler() {
                public void receiveResult(RegisterResponse result) {
                    System.out.println("Got result: " + result.getApproved());
                }
            };
        stub.startRegister(request, callback);
        System.out.println("Request sent");
    }
}
```

This is the critical step. It causes the client to kick start the background thread to listen on port 6060 for the response. Conceptually, the background thread maintains an internal table like this:

<table>
<thead>
<tr>
<th>Message ID</th>
<th>Callback</th>
</tr>
</thead>
<tbody>
<tr>
<td>m001</td>
<td>Callback1</td>
</tr>
<tr>
<td>m002</td>
<td>Callback2</td>
</tr>
</tbody>
</table>

When it receives a response and finds that it is related to m001, it will call the callback for m001.

Note the difference between "using a callback" and "using a separate listener". Using a callback means the API is asynchronous, no matter one or two HTTP connections are used. For example, you can use a callback without using a separate listener:

1: Call it without waiting for the result (provide a callback object)
2: It uses a single HTTP connection (synchronous) to invoke the web service and wait for the response.
3: Call the callback

In this case, the API is asynchronous and your code seems to be asynchronous, but as only one HTTP connection is used, it is still subject to the timeout problem. So this is suitable when the processing is not too lengthy.
(won't cause a timeout) and your client code really wants to proceed without getting the result immediately.

You've already seen the case of using a callback and a separate listener (business registration). This is the ultimate asynchronous situation. It is good for lengthy processing when your client code can proceed without getting the result.

You've also seen the case of not using a callback and not using a separate listener (the normal case). This is the ultimate synchronous situation. It is good for fast processing and your client code needs to wait for the result.

Finally, it is also possible to not use a callback while using a separate listener:

This is good for lengthy processing when your client code must wait for the result before proceeding.

Now the client is done. For the web service to decode the message ID and reply-to URL from the SOAP message, you need to engage the addressing module in the web service. This is the case by default. You can verify that in global configuration file for Axis, c:\axis\conf\axis2.xml:

```xml
<axisconfig name="AxisJava2.0">
  ...
  <module ref="addressing"/>
  ...
</axisconfig>
```

Start the Axis server (if it is not yet started). Run the client and it should work:

```
Request sent
log4j:WARN No appenders could be found for logger
log4j:WARN Please initialize the log4j system pro
Got result: 123
```

However, there are still two issues left. First, once started, the background thread will not terminate and will continue to listen on that port. So if you run it again, it will fail to grab the port and will fail to receive the response. Second, it will prevent your JVM from terminating. You can verify that with the red button in Eclipse in the above screen shot. Now, click that red button to terminate it. To fix these problems, modify the code:

```java
public class BizRegClient {
  public static void main(String[] args) throws RemoteException {
    ManualServiceStub stub = new ManualServiceStub();
    final ServiceClient serviceClient = stub.getServiceClient();
    serviceClient.engageModule("addressing");
```
Options options = serviceClient.getOptions();
options.setUseSeparateListener(true);
Register request = new Register();
request.setBizName("Foo Ltd.");
request.setOwnerId("Kent");
ManualServiceCallbackHandler callback = new ManualServiceCallbackHandler() {
    public void receiveResultregister(RegisterResponse result) {
        System.out.println("Got result: "+result.getApproved());
        finished();
    }
    private void finished() {
        try {
            serviceClient.cleanup();
        } catch (AxisFault e) {
            throw new RuntimeException(e);
        } finally {
            System.exit(0);
        }
    }
};
stub.startregister(request, callback);
System.out.println("Request sent");
}
}

What if the web service returns an error? You can catch it this way:

public class BizRegClient {
    public static void main(String[] args) throws RemoteException {
        ManualServiceStub stub = new ManualServiceStub();
        final ServiceClient serviceClient = stub._getServiceClient();
        serviceClient.engageModule("addressing");
        Options options = serviceClient.getOptions();
        options.setUseSeparateListener(true);
        Register request = new Register();
        request.setBizName("Foo Ltd.");
        request.setOwnerId("Kent");
        ManualServiceCallbackHandler callback = new ManualServiceCallbackHandler() {
            public void receiveResultregister(RegisterResponse result) {
                System.out.println("Got result: "+result.getApproved());
                finished();
            }
        }
        stub.startregister(request, callback);
        System.out.println("Request sent");
    }
}

Inspecting the WS-Addressing header blocks

You can also check the WS-Addressing header blocks using the TCP Monitor. The request should be like:
Note that TCP Monitor will get a dummy response as the real response is sent to port 6060.

Avoiding modifications to the message receiver

Currently you're modifying ManualServiceMessageReceiverInOut.java which is generated by <wsdl2code>. This is no good as it will be overwritten if you run <wsdl2code> again. Therefore, a better way is to extend it. For example, create ManualServiceReceiver.java and move the receive() method into there:

```java
public class ManualServiceReceiver extends ManualServiceMessageReceiverInOut {
    public void receive(MessageContext messageCtx) throws AxisFault {
        messageCtx.setProperty(DO_ASYNC, "true");
    }
}
```

This is the target URL. Why is it needed? This allows routing the request message through intermediate hops because the target URL is maintained in the message.

As described before, it allows the client to uniquely specify the operation it wants to call. This is also specified by the WS-Addressing standard.

The WS-Addressing namespace

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/",
    xmlns:wsa="http://www.w3.org/2005/08/addressing">
    <soapenv:Header>
        <wsa:To>
            http://localhost:1234/axis2/services/ManualService
        </wsa:To>
        <wsa:ReplyTo>
            <wsa:Address>
                http://192.168.0.146:6060/axis2/services/ManualService25107363
            </wsa:Address>
        </wsa:ReplyTo>
        <wsa:MessageID>
            urn:uuid:E8A307C115655F0CFC1197866807896
        </wsa:MessageID>
    </soapenv:Header>
    <soapenv:Body>
        <ns1:register xmlns:ns1="urn:fake.gov:biz/reg">
            <bizName>Foo Ltd.</bizName>
            <ownerId>Kent</ownerId>
        </ns1:register>
    </soapenv:Body>
</soapenv:Envelope>
```

Note that TCP Monitor will get a dummy response as the real response is sent to port 6060.
Then delete the receive() method from ManualServiceMessageReceiverInOut:

```java
public class ManualServiceMessageReceiverInOut ...
{
    public void receive(MessageContext messageCtx) throws AxisFault {
        messageCtx.setProperty(DO_ASYNC, "true");
        super.receive(messageCtx);
    }

    public void invokeBusinessLogic(...) {
        ...
    }
}
```

Modify build.xml to fix services.xml so that it uses ManualServiceReceiver as the message receiver:

```xml
<target name="generate-service">
    <wsdl2code
        wsdlfilename="${name}.wsdl"
        serverside="true"
        generateservicexml="true"
        skipbuildxml="true"
        serversideinterface="true"
        namespacespackages="urn:fake.gov:biz:reg=gov.fake.bizreg"
        targetsourcefolderlocation="src"
        targetresourcesfolderlocation="src/META-INF"
        overwrite="true"/>
    <replaceregexp
        file="src/META-INF/services.xml"
        match="${name}Skeleton"
        replace="${name}Impl" />
    <replaceregexp
        file="src/META-INF/services.xml"
        match="${name}MessageReceiverInOut"
        replace="${name}Receiver" />
</target>
```

Delete the services.xml and run build.xml again. Everything should continue to work.

### Summary

To support a lengthy operation in a web service, its message receiver needs to enable the DO_ASYNC flag so that it creates a new thread to call your business logic and return the response in that thread. For this to work, the client needs to kick start a background thread to listen on a certain port for the response and include a reply-to URL in a header block in the request SOAP message. To distinguish which response is for which request, the client also needs to include a unique message ID into the message and the web service needs to copy that into a relates-to header block. WS-Addressing supports the encoding and decoding of the message ID, relates-to and reply-to URL.

WS-Addressing is implemented by a module called "addressing" in Axis. A module is just some functionality that can be enabled or disabled. When it is enabled, it is said to be "engaged".

The client API can be synchronous or asynchronous, independent of whether the transport is synchronous or not. If your code can and should proceed without waiting for the result, use the asynchronous API. If it must wait for the
result, use the synchronous API.
Chapter 9

Signing and encrypting SOAP messages
What's in this chapter?

In this chapter you'll learn how to sign and encrypt SOAP messages.

Private key and public key

Usually when you encrypt some text using a key, you need the same key to decrypt it:

This is called "symmetric encryption". If you would like to send something to me in private, then we need to agree on a key. If you need to send something private to 100 individuals, then you'll need to negotiate with each such individual to agree on a key (so 100 keys in total). This is troublesome.

To solve the problem, an individual may use something called a "private key" and a "public key". First, he uses some software to generate a pair of keys: One is the private key and the other is the public key. There is an interesting relationship between these two keys: If you use the private key to encrypt something, then it can only be decrypted using the public key (using the private key won't work). The reverse is also true: If you use the public key to encrypt something, then it can only be decrypted using the private key:
After generating the key pair, he will keep the private key really private (won't tell anyone), but he will tell everyone his public key. Can other people find out the private key from the public key? It is extremely difficult, so there is no worry about it. Now, suppose that you'd like to send something confidential to an individual Paul (see the diagram below), you can use his public key to encrypt it. Even though other people know his public key, they can't decrypt it (as it is encrypted using the public key, only the private key can decrypt it). Only Paul knows the private key and so only he can decrypt it:

This kind of encryption is called "asymmetric encryption".

Paul's key pair

```
  k1-priv  k1-pub
   Hello, world!
   
   Encrypt
   k1-priv
   
    kfjdih9368dhfj

   Decrypt
   k1-pub
   
```

```
  k1-pub
   Hello, world!
   
   Encrypt
   
    kfjdih9368dhfj

   Decrypt
   k1-priv
   
```

```
  k1-priv  k1-pub
   Hello, world!
   
   Encrypt
   k1-pub
   
    kfjdih9368dhfj

   Decrypt
   k1-priv
   
```
Digital signature

Suppose that the message you send to Paul is not confidential. However, Paul really needs to be sure that it is really from you. How to do that? You need to prove to Paul that the creator of the message knows your private key. If he does, then he must be you (remember, nobody else is supposed to know your private key). To prove that, you can use your private key to encrypt the message, then send it to Paul. Paul can try to decrypt it using your public key. If it works, then the creator of the message must know your private key and must be you.

However, this is not a good solution, because if the message is long, the encrypted message may double in size and the encryption takes a lot of time. To solve this problem, you can feed the message to a "one way hash function" (see the diagram below). No matter how long the input is, the output from the one way hash function is always the same small size (e.g., 128 bits). In addition, if two input messages are different (maybe just a single bit is different), then the output will be completely different. Therefore, the output message can be considered a small-sized snapshot of the input message. It is therefore called the "message digest" of the original message:

Another feature of the one way hash function is that it is very fast to calculate the digest of a given message, but it is extremely difficult to calculate a message given a digest. Otherwise people would find different messages for a given digest and it is no longer a good snapshot for the message:
Now, to prove to Paul that you know your private key, you can use your private key to encrypt the message digest (because the digest is small, the result is also small and the encryption process will be fast), then send both the message and the message digest to Paul. He can try to decrypt the digest using your public key. Then he can calculate the digest from the message and compare the two. If the two match, then the person producing the encrypted digest must be you:

The encrypted digest is called the "digital signature". The whole process of calculating the digest and then encrypting it is called "signing the message".

**Signing and encrypting**

What if you'd like to sign the message, while keeping the message available to Paul only? Just sign it as usual (see the diagram below) and then encrypt the message and the digest using Paul's public key. When Paul receives it, he uses his private key to decrypt it and then go on to verify the signature as usual:
Certificate and CA

This seems to work very well. However, when you need to say send a confidential message to Paul, you'll need his public key. But how can you find out his public key? You can call him on the phone to ask him. But how can you be sure that the person on the phone is really Paul? If he is a hacker, he will tell you his public key. When you send the message to Paul using the hacker's public key, the hacker will be able to decrypt it using his private key.

If you need to communicate with many different individuals, this will get even more troublesome. To solve the problem, Paul may go to a government authority, show his ID card and etc and tell the authority his public key. Then the authority will generate an electronic message (like an email) stating Paul's public key. Finally, it signs that message using its own private key:

| Name: Paul |
| Public key: 666888 |
| Signature |
Such a signed message is called a "certificate". That authority is called a "certificate authority (CA)". Then Paul can put his certificate on his personal web site, email it to you directly or put it onto some 3rd party public web site. From where you get the certificate is unimportant. What is important is that if you can verify the signature of that CA and you trust what the CA says, then you can trust that public key in the certificate. In order to verify the signature, you will need the public key of that CA. What?! You're back to the origin of the problem. However, you only need to find out a single public key for a single entity (the CA), not a public key for everyone you need to communicate with. How to obtain that public key? Usually it is already configured in your browser or you can download it from a trusted web site, newspaper or other sources that you trust.

A CA doesn't really need to be a government authority. It can be well known commercial organizations such as VeriSign.

It means that in order to use asymmetric encryption and digital signature, people need private keys, public keys, a CA and certificates. All these elements combined together is called a “public key infrastructure (PKI)” because it provides a platform for us to use public keys.

**Distinguished name**

If you review the certificate:

```
Name: Paul  
Public key: 666888  
Signature
```

you will see that it is not that useful because there are probably millions of people named “Paul” in the world. Therefore, in a real certificate, usually the country, city and the company of that individual are also included like:

```
CN means common name  
Organization (company)  
State  
Country
```

Now if you're looking for the public key of Paul McNeil who works at IBM, you know that the certificate above should NOT be used.

**Performance issue with asymmetric encryption**

Suppose that you'd like to send an encrypted message to Paul. You can use
Paul's public key to do that. However, in practice few people would do it this way, because asymmetric encryption is very slow. In contrast, symmetric encryption is a lot faster. To solve this problem, you can generate a random symmetric key, use it to encrypt the message, then use Paul's public key to encrypt that symmetric key and send it to Paul along with the encrypted message. Paul can use his private key to get back the symmetric key and then use it to decrypt the message:

**Keeping key pair and certificates in Java**

In order to use PKI, typically you should have a private key for yourself (see the diagram below), a certificate for yourself so that you can send to others, a certificate for each person that you need to send something confidential to (e.g., Paul and Mary) and the public keys of the CA's that you trust. For the public key of the CA, you don't directly store its public key. Instead, you store its certificate which contains its public key. But who issued that certificate to it? It was issued by itself (signed by its own private key):
Such a table is called a “keystore” in Java (see the diagram below). A keystore is stored in a file. In addition, each entry in the table has a name called the “alias” of the entry. This way you can, e.g., tell the software to sign a particular message using the private key in the “john” entry (yourself), or encrypt the message using the public key in “paul” entry. Without the alias you will have to use the DN to refer to an entry:

```
keystore

<table>
<thead>
<tr>
<th>Alias</th>
<th>Private key</th>
<th>Certificate</th>
</tr>
</thead>
<tbody>
<tr>
<td>john</td>
<td>k1-priv</td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>paul</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>mary</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>
```

### Generating a key pair

In order to generate a key pair, you can use the keytool program in JDK. For example, if your JDK is in `c:\Program Files\Java\jdk`, then you can find keytool.exe in the bin sub-folder (i.e., `c:\Program Files\Java\jdk\bin`). For convenience, let's add `c:\Program Files\Java\jdk\bin` to the PATH:
Note that this PATH setting affects this command prompt only. If later you use a
new command prompt, you'll need to set the PATH again. Next, create a folder
c:\keys to hold the keys and change into there:

Now, generate a key pair for your web service client:

Let's run it:

The certificate
will be signed
by k1-priv (self-
signed)

The path to the keystore. A keystore is just a file. Here only the filename (client.ks)
is specified, so assume it's in the current directory (c:\keys). As it doesn't exist
yet, it will be created by keytool.

The name (alias) of
the entry

The key generation algorithm. Commonly it is either DSA or RSA. Java supports
both but some of the libraries you use later only support RSA, so use it here.

The signature algorithm. Here, hash the message using SHA1 first and then encrypt it
using the RSA private key. If you don't specify it here, keytool will use MD5withRSA. But MD5
is known to be insecure nowadays, so don't use MD5 anymore.

Let's run it:
You need to provide a keystore password to protect the keystore. You can consider that keytool will append this password to the content of the keystore and then generate a hash and store it into the keystore. If someone modifies the keystore without this password, he won't be able to update the hash. The next time you run keytool on this keystore, it will note the mismatch and warn you not to use this keystore anymore.

```
C:\keys>keytool -genkey -alias c1 -keystore client.ks -keyalg RSA -sigalg SHA1withRSA
Enter keystore password: client-ks-pass
What is your first and last name?
  [Unknown]: John
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]: US
Is CN=c1, OU=Unknown, O=Bar, L=Unknown, ST=Unknown, C-US correct?
  [Yes]: yes
Enter key password for <c1>
  [RETURN if same as keystore password]: c1-pass
```

The DN of John

You need to provide an entry password to protect the entry for c1. You can consider that keytool will use this password to encrypt c1’s private key. This way other people won’t be able to read c1’s private key.

To verify that the entry has been added, you can list the entries:

```
C:\keys>keytool -list -keystore client.ks
Enter keystore password: client-ks-pass

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry
   c1, Dec 12, 2007, keyEntry,
```

Note that it asks for the keystore password so that it can verify the hash. If you'd like to see more details in the entries, use the -v option:
You can see that both the "Owner" and the "Issuer" are set to the DN of c1. It shows that it is indeed a self-signed certificate. Having a self-signed certificate is not useful. You need to ask a CA to sign it. To do that, generate a certificate request first:

```
C:\keys>keytool -certreq -alias c1 -keystore client.ks -file c1.csr
```

You need to send to a CA. In real life, you should send it to VeriSign or some well known CA to get a certificate (of course a payment is required). Here you'll setup your own CA.
Setting up a CA

Go to http://www.openssl.org/related/binaries.html to download the Windows version of OpenSSL. Suppose the file is Win32OpenSSL-v0.9.8a.exe. Login as the Administrator and run it. Follow the instruction to complete the installation. Suppose that it has been installed into c:\OpenSSL. To make it easier to run, add c:\OpenSSL\bin to the PATH:

```
c:\WINDOWS\system32\cmd.exe
C:\keys>set PATH=%PATH%;c:\OpenSSL\bin
C:\keys>
```

Next, create a folder say c:\CA to contain the files of the CA. Then create a private key for the CA itself:

Some openssl commands need to save a random seed information to a file ("random file"). You need to tell it the path to that file. Here, just tell it to use a file named "rand" in the current folder.

```
c:\>cd CA

C:\CA>set RANDFILE=rand
C:\CA>openssl req -new -keyout cakey.pem -out careq.pem
```

Work on a request
Create a new private key and a certificate request
Put the private key into this file
Put the certificate request into this file

Run it and it will prompt you for the DN of the CA and a password to encrypt the private key (e.g., you may use "ca-pass"): 
Next, generate a self-signed certificate for it:
Run it and enter "ca-pass" as the password for the CA key:

Now you're about to use this CA to sign the certificate request from John (john.csr). However, before that, you need to note that when a CA issues a new certificate, it will put a unique serial number into that certificate. So you need to tell OpenSSL what is the next serial number to use. To do that:

Store the string "02" into a file serial.txt. The file will be created. This way OpenSSL will use 02 as the next serial number. Then it will set it to 03 automatically.

Note that the "0" is necessary. Using "2" will NOT work because OpenSSL expects a hexadecimal number that contains an even number of digits.

To sign c1's certificate request:
Chapter 9  Signing and encrypting SOAP messages

Run it and enter "ca-pass" as the password for the CA key:

```
c:\CA>openssl x509 -CA cacert.pem -CAkey cakey.pem -CAserial serial.txt -req
   -in c:\keys\c1.csr -out c:\keys\c1.cer -days 1095
```

Importing the certificate into the keystore

Now you have got the certificate in c1.cer, you can import it into the keystore. However, before doing that, you must first import the certificate of the CA itself into your keystore as a trusted CA certificate, otherwise it will refuse to import John's certificate. To do that:

```
c:\CA>openssl x509 -CA cacert.pem -CAkey cakey.pem -CAserial serial.txt -req -in c:\keys\c1.csr -out c:\keys\c1.cer -days 1095
Loading ‘creeen’ into random state - done
Signature ok
subject=/C=US/ST=Unknown/L=Unknown/O=Bar/OU=Unknown/CN=c1
Getting CA Private Key
Enter pass phrase for cakey.pem:
```

Run it:

```
c:\CA>cd \keys
```

Import a certificate into the keystore

Create a certificate entry named "testCA". You can use any name that you like and it won't make any difference.

```
c\keys>keytool -import -alias testCA -file c:\CA\cacert.pem -keystore client.ks
```

The CA's certificate is in this file. In real world, when you receive your certificate from the CA (e.g., VeriSign), it will also give you its own certificate. Or you can probably download it from its web site.

Run it:
Note that it asked you to trust this certificate or not. This is a very important decision. If you trust this certificate as a CA certificate, you will trust all certificates issued by it. Next, add John's certificate to the keystore to replace his self-signed certificate. This is also done using the -import option:

```
C:\keys>keytool -import -alias c1 -file c1.cer -keystore client.ks
```

When keytool finds an existing entry with the named "c1" in the keystore, it knows you're trying to replace a certificate issued by a CA for the existing self-signed one.

```
c:\keys>keytool -import -alias c1 -file c1.cer -keystore client.ks
```

The certificate is in this file

Run it:

```
C:\keys>keytool -import -alias c1 -file c1.cer -keystore client.ks
Enter keystore password: client-ks-pass
Enter key password for <c1>c1-pass
Certificate reply was installed in keystore
```

To verify, you can list the entries in the keystore:
C:\keys>keytool -list -v -keystore client.ks
Enter keystore password:  client-ks-pass

Keystore type: jks
Keystore provider: SUN

Your keystore contains 2 entries

<table>
<thead>
<tr>
<th>Alias name: testca</th>
<th>Entry type: trustedCertEntry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation date: Dec 12, 2007</td>
<td></td>
</tr>
<tr>
<td>Owner: CN=CA, O=Test CA, ST=Some-State, C=US</td>
<td></td>
</tr>
<tr>
<td>Issuer: CN=CA, O=Test CA, ST=Some-State, C=US</td>
<td></td>
</tr>
<tr>
<td>Serial number: d4bf644c2e6ae6eb694</td>
<td></td>
</tr>
<tr>
<td>Valid from: Sat Dec 08 10:26:14 CST 2007 until: Tue Dec 05 10:26:14 CST 2017</td>
<td></td>
</tr>
<tr>
<td>Certificate fingerprints:</td>
<td></td>
</tr>
</tbody>
</table>

It is a trusted certificate entry, i.e., a trusted CA certificate.

There are 2 entries in the keystore

<table>
<thead>
<tr>
<th>Alias name: c1</th>
<th>Entry type: keyEntry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creation date: Dec 12, 2007</td>
<td></td>
</tr>
<tr>
<td>Entry type: keyEntry</td>
<td></td>
</tr>
<tr>
<td>Certificate chain length: 2</td>
<td></td>
</tr>
<tr>
<td>Certificate[1]:</td>
<td></td>
</tr>
<tr>
<td>Owner: CN=c1, OU=Unknown, O=Bar, L=Unknown, ST=Unknown, C=US</td>
<td></td>
</tr>
<tr>
<td>Issuer: CN=CA, O=Test CA, ST=Some-State, C=US</td>
<td></td>
</tr>
<tr>
<td>Serial number: 4</td>
<td></td>
</tr>
<tr>
<td>Certificate fingerprints:</td>
<td></td>
</tr>
<tr>
<td>Certificate[2]:</td>
<td></td>
</tr>
<tr>
<td>Owner: CN=CA, O=Test CA, ST=Some-State, C=US</td>
<td></td>
</tr>
<tr>
<td>Issuer: CN=CA, O=Test CA, ST=Some-State, C=US</td>
<td></td>
</tr>
<tr>
<td>Serial number: d4bf644c2e6ae6eb694</td>
<td></td>
</tr>
<tr>
<td>Valid from: Sat Dec 08 10:26:14 CST 2007 until: Tue Dec 05 10:26:14 CST 2017</td>
<td></td>
</tr>
<tr>
<td>Certificate fingerprints:</td>
<td></td>
</tr>
</tbody>
</table>

It is a key entry, i.e., a private key along with a certificate.
It means that there are two certificates in the entry
The first certificate is c1's certificate. From the "Issuer" field you can see it is issued by the test CA, so the next certificate is that of the test CA.

There are 2 entries in the keystore

The second certificate is the certificate of the test CA

A certificate chain is also called "certificate path". If the certificate of your test CA was issued by yet another CA, then the certificate path would contain the certificate of that other CA as the last certificate.
Installing Rampart

In order to perform signing or encryption, you need an Axis module called "Rampart". So, go to http://ws.apache.org/axis2/modules to download it. Suppose that it is rampart-1.3.zip. Unzip it into say c:\rampart. Rampart needs another library xalan 2.7.0. If you're using JDK 5 or earlier, you probably has only an old version. So, in that case, download xalan-2.7.0.jar from http://www.apache.org/dist/java-repository/xalan/jars and put it into c:\rampart\lib.

To make rampart available to your web services at runtime, copy all the files shown below:

A .mar file is a module archive. It represents a module in Axis.

The rampart module needs these jar files

To make it available to your client, copy the WrappedService project and paste it as SecureService. Adjust the linked folder. To make the rampart module available to your client code, add the jar files in c:\rampart\lib to the build path of your project and copy rampart-1.3.mar into your project in such a folder structure:

SecureService
src
repository
modules
rampart-1.3.mar

Just like the Axis server which has a repository, your Axis client can also have a repository.

Rename the WSDL to SecureService.wsdl and replace the word "Secure" for "Wrapped" in it. Update the build.xml file:

<project basedir="." default="jar.server">
...

Signing SOAP messages

In order to sign the SOAP messages, modify the WSDL file:
Chapter 9  Signing and encrypting SOAP messages

Saying that the <Body> should be signed is not enough. You still need to specify

```xml
<wsp:Policy wsu:Id="p1">
  <sp:SignedParts>
    <sp:Body />
  </sp:SignedParts>
</wsp:Policy>
```

This is a "policy". A policy specifies non-functional requirements of the web service (e.g., security, quality of service). The syntax of specifying a policy is governed by the WS-Policy standard.

This is a "policy assertion". It requires certain parts of the SOAP message be signed. The parts should be signed are listed here. Here, only the <Body> of the SOAP message should be signed.

Apply the policy "p1" to the SOAP binding of the concat operation. It means the <Body> of all the messages for the concat operation must be signed as long as they're using SOAP over HTTP. Without this the policy would be sitting there idle and would have no effect.

As the <PolicyReference> element belongs to a foreign namespace (wsp), there is no guarantee that the program processing the WSDL file (e.g., <wsdl2code>) understands it. This attribute requires that the program understand it, otherwise it should abort the processing.

If you had multiple operations in the port type and they all required signed messages, you would move the <PolicyReference> to there so that it would apply to the SOAP binding of the SecureService port type.

It belongs to the web service policy namespace

It belongs to the security policy namespace

It belongs to the security policy namespace

Apply the policy "p1" to the SOAP binding of the concat operation. It means the <Body> of all the messages for the concat operation must be signed as long as they're using SOAP over HTTP. Without this the policy would be sitting there idle and would have no effect.

As the <PolicyReference> element belongs to a foreign namespace (wsp), there is no guarantee that the program processing the WSDL file (e.g., <wsdl2code>) understands it. This attribute requires that the program understand it, otherwise it should abort the processing.

If you had multiple operations in the port type and they all required signed messages, you would move the <PolicyReference> to there so that it would apply to the SOAP binding of the SecureService port type.

Saying that the <Body> should be signed is not enough. You still need to specify
that asymmetric encryption should be used and what signature algorithms are supported and etc.:

Why have an extra <Policy> element? For example, the <X509Token> element can be reused in another place (e.g., <RecipientToken> below), then it will be designed as a policy assertion. One assertion cannot directly include another assertion. It has to include a policy first. This way, different policy assertions could be put inside.

This policy assertion states that asymmetric encryption should be used. This assertion and the <SignedParts> assertion are AND'ed together.

The 1st assertion in the asymmetric assertion: What kind of token (certificate here) should be used by the initiator (i.e., the client)? It should use an X509 token, which means an certificate. X509 is the official name.

Always include the token (certificate) in the message to the web service.

X509 certificates have different versions and presentations. Here use v3 and the XML presentation as specified in the web service security (WSS) X509 token profile 1.0.

The 2nd assertion in the asymmetric assertion: What kind of token (certificate here) should be used by the recipient (i.e., the web service)?

Also use X509 v3 certificate for the web service, but do not send its certificate to the client. Instead, send enough information to the client so that the client can retrieve it. How? You'll see later.

The 3rd assertion in the asymmetric assertion. It supports the use of 3DES for encryption and RSA 1.5 algorithm for digital signatures.

In principle you could have multiple elements like this to say that it supports multiple algorithm suites and let the client and the service negotiate to decide which one to use. However, for the moment this negotiation is not supported in Axis. It means what is supported will actually be used. So, do not list multiple alternatives in the policy.

Finally, you still need to say that it supports the Web Service Security (WSS) standard v1.0:
Generate the service stub and client stub. Fill out the code in the implementation class:

```java
public class SecureServiceImpl implements SecureServiceSkeletonInterface {
    public String concat(String s1, String s2) {
        return s1 + s2;
    }
}
```

Create `SecureClient.java` in the client package:

```java
import org.apache.axis2.context.ConfigurationContext;
import org.apache.axis2.context.ConfigurationContextFactory;

public class SecureClient {
    public static void main(String[] args) throws RemoteException {
        ConfigurationContext context = ConfigurationContextFactory
                .createConfigurationContextFromFileSystem("repository");
        SecureServiceStub stub = new SecureServiceStub(context);
        stub._getServiceClient().engageModule("rampart");
        String result = stub.concat("xyz", "111");
        System.out.println(result);
    }
}
```

For rampart to sign the `<Body>`, it needs access to the policy. Fortunately `<wsdl2code>` has extracted the policy information from the WSDL and put it into the Java code generated. What is missing is, what is the alias of the certificate to use, the password, the location of the keystore and etc. All this information can be specified in a Java String or in a text file. Here, let's put it into a text file `rampart-config.xml` in the project root:
Chapter 9  Signing and encrypting SOAP messages

Signing and encrypting SOAP messages

To load the configuration file into rampart, modify the SecureClient.java:

```java
import org.apache.axiom.om.impl.builder.StAXOMBuilder;
import org.apache.axis2.description.PolicyInclude;
import org.apache.neethi.Policy;
import org.apache.neethi.PolicyEngine;

public class SecureClient {
    public static void main(String[] args) throws RemoteException,
            FileNotFoundException, XMLStreamException {
        ConfigurationContext context = ConfigurationContextFactory
          .createConfigurationContextFromFileSystem("repository");
        SecureServiceStub stub = new SecureServiceStub(context);
        stub._getServiceClient().engageModule("rampart");
        StAXOMBuilder builder = new StAXOMBuilder("rampart-config.xml");
        OMElement configElement = builder.getDocumentElement();
        Policy rampartConfig = PolicyEngine.getPolicy(configElement);
        stub._getServiceClient().getAxisService().getPolicyInclude()
          .addPolicyElement(PolicyInclude.SERVICE_POLICY, rampartConfig);
        String result = stub.concat("xyz", "111");
        System.out.println(result);
    }
}
```

Load the rampart-config.xml file and get the `<Policy>` element

Convert the `<Policy>` XML element into a Java object

Add that Policy object to the existing policy. Apply this extra Policy to the whole web service.

Of course you need to create a PasswordCallbackHandler class in the client package:

```java
public class PasswordCallbackHandler implements CallbackHandler {
    public void handle(Callback[] callbacks)
```
Chapter 9  Signing and encrypting SOAP messages

public class SecureClient {
    public static void main(String[] args) throws RemoteException, 
        FileNotFoundException, XMLStreamException {
        ConfigurationContext context = ConfigurationContextFactory 
                .createConfigurationContextFromFileSystem("repository");
        SecureServiceStub stub = new SecureServiceStub(context, 
                "http://localhost:1234/axis2/services/SecureService");
        stub._getServiceClient().engageModule("rampart");
        StAXOMBuilder builder = new StAXOMBuilder("rampart-config.xml");
        OMElement configElement = builder.getDocumentElement();
        Policy rampartConfig = PolicyEngine.getPolicy(configElement);
        stub._getServiceClient().getAxisService().getPolicyInclude() 
                .addPolicyElement(PolicyInclude.SERVICE_POLICY, rampartConfig);
        String result = stub.concat("xyz", "111");
        System.out.println(result);
    }
}

Run it and you will see an error in the console saying the a header was not understood:

Exception in thread "main" org.apache.axis2.AxisFault: Must Understand check failed for
    at org.apache.axis2.util.Utils.getFaultFromMessageContext(Exception
    at org.apache.axis2.description.OutInAxisOperationClient.handleResponse(Impl
    at org.apache.axis2.description.OutInAxisOperationClient.send(OutInAxisOperatio
    at org.apache.axis2.AxisFault:91)
    at com.ctdev.secure.client.SecureServiceStub.concat(SecureServiceStub.java:154)
    at com.ctdev.secure.client.SecureClient.main(SecureClient.java:29)

This is fine as the web service is not yet prepared to handle the digital signature. What is interesting is in the request message as shown in the TCP Monitor:
Chapter 9  Signing and encrypting SOAP messages

Supporting digital signatures in the web service

Ideally, when generating the service stub, `<wsdl2code>` should consult the policy in the WSDL and setup rampart properly. However, the current version of Axis is not doing that. That's why the web service is not understanding the `<Security>` header element. To fix the problem, add the policy to `services.xml`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<serviceGroup>
  <service name="SecureService">
    <messageReceivers>
      <messageReceiver mep="http://www.w3.org/ns/wsd1/in-out"
                       class="com.ttdev.secure.SecureServiceMessageReceiverInOut" />
    </messageReceivers>
    <parameter name="ServiceClass">
      com.ttdev.secure.SecureServiceImpl
    </parameter>
  </service>
</serviceGroup>
```

A `<Security>` element is added. It is a header entry.

The "mustUnderstand" attribute is set to 1, meaning that the receiver (the service) must handle this header, otherwise it must return a SOAP fault (which is the case here).

The signature is signing over this element, i.e., the `<Body>` element.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A token (certificate) is directly included here

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.

A `<Signature>` element represents a digital signature. You don't need to fully understand its details. If later you encrypt the message, there will be an `<EncryptedData>` element as its sibling.

A `<BinarySecurityToken>` element has been added to the `<Security>` header. The token includes the certificate used for the signature.

The signature is signing over the `<Body>` element, which contains the message to be signed.

The signature was created using this token (certificate)

The `<Body>` element is basically unchanged. The only exception is that an id has been added so that the signature can refer to it.
Chapter 9  Signing and encrypting SOAP messages

Then engage the rampart module and add the rampart configuration as a policy assertion:
Next, create PasswordCallbackHandler.java in the com.ttdev.secure package to provide the password to decrypt the private key in the "s1" alias:

```java
public class PasswordCallbackHandler implements CallbackHandler {
    public void handle(Callback[] callbacks) throws IOException, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            WSPasswordCallback pwcb = (WSPasswordCallback) callbacks[i];
            String id = pwcb.getIdentifer();
            if (id.equals("s1")) {
                pwcb.setPassword("s1-pass");
            }
        }
    }
}
```

To get a certificate for the service, open a command prompt and then:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<serviceGroup>
    <service name="SecureService">
        <messageReceivers>
            ...
        </messageReceivers>
        <parameter name="s1"/>
        <operation name="concat"/>
        ...
    </operation>
    <wsp:Policy ref="rampart"/>
        <sp:AsymmetricBinding>
            ...
        </sp:AsymmetricBinding>
        <sp:Wss10>
            ...
        </sp:Wss10>
    </sp:SignedParts>
    <sp:Body />
</sp:SignedParts>
    <RampartConfig
        xmlns="http://ws.apache.org/rampart/policy">
        <user>s1</user>
        <passwordCallbackClass>com.ttdev.secure.PasswordCallbackHandler</passwordCallbackClass>
        <signatureCrypto>
            <crypto provider="org.apache.ws.security.components.crypto.Merlin">
                <property name="org.apache.ws.security.crypto.merlin.keystore.type">JKS</property>
                <property name="org.apache.ws.security.crypto.merlin.file">c:/keys/service.ks</property>
                <property name="org.apache.ws.security.crypto.merlin.keystore.password">service-ks-pass</property>
            </crypto>
        </signatureCrypto>
    </RampartConfig>
</wsp:Policy>
</service>
</serviceGroup>
```

```
<table>
<thead>
<tr>
<th>Alias</th>
<th>Private key</th>
<th>Certificate</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Engage the rampart module. The ordering of the `<module>` element doesn't really matter as long as it is directly in the `<service>` element.

It is used as a policy assertion

You'll create this class later

You'll create this keystore entry later for the web service

The keystore password

You'll create this keystore later

You'll create this keystore later

You'll create this keystore later
Chapter 9  Signing and encrypting SOAP messages

```
c:\>cd \keys
c:\keys>keytool -genkey -alias s1 -keystore service.ks -keyalg RSA -sigalg SHA1withRSA
Enter keystore password:  service-ks-pass
What is your first and last name?
[Unknown]:  s1
What is the name of your organizational unit?
[Unknown]:
What is the name of your organization?
[Unknown]:  Foo
What is the name of your City or Locality?
[Unknown]:
What is the name of your State or Province?
[Unknown]:
What is the two-letter country code for this unit?
[Unknown]:  US
Is CN=s1, OU=Unknown, O=Foo, L=Unknown, ST=Unknown, C=US correct?
[no]:  yes
Enter key password for <s1>
(RETURN if same as keystore password):  s1-pass

Generate a certificate request for it:

c:\keys>keytool -certreq -alias s1 -keystore service.ks -file s1.csr
Enter keystore password:  service-ks-pass
Enter key password for <s1>s1-pass

Use your test CA to create a certificate for it (remember that "ca-pass" is the password for the CA key):

c:\keys>cd \CA
c:\CA>openssl x509 -CA cacert.pem -CAkey cakey.pem -CAserial serial.txt -req -in c:\keys\s1.csr -out c:\keys\s1.cer -days 1095

Import the certificate of the CA and that for the service into the keystore for the service:

c:\keys>cd \keys
c:\keys>keytool -import -alias testCA -keystore service.ks -file c:\CA\cacert.pem
Enter keystore password:  service-ks-pass
Owner: CN=CA, O=Test CA, ST=Some-State, C=US
Issuer: CN=CA, O=Test CA, ST=Some-State, C=US
Serial number: d4bf64c2e6aeb694
Valid from: Sat Dec 08 10:26:14 CST 2007 until: Tue Dec 05 10:26:14 CST 2017
Certificate fingerprints:
Trust this certificate? [no]:  yes
Certificate was added to keystore

c:\keys>keytool -import -alias s1 -keystore service.ks -file s1.cer
Enter keystore password:  service-ks-pass
Certificate reply was installed in keystore

Do you need to import c1's certificate? No. As the client will include it in the message, you don't need it in the keystore. On the other hand, do you need to import s1's certificate into the keystore for the client? Yes. This is because the web service will not send its certificate to the client, but just the issuer's DN and serial number of the certificate. So the client needs this certificate in its keystore. So, import it:

c:\keys>keytool -import -alias s1 -keystore client.ks -file s1.cer
Enter keystore password:  client-ks-pass
Certificate was added to keystore

Now, run the client again. This time it will work. If you check the SOAP response
That is, it is telling the service that the certificate used to sign the message is issued by CN=CA,O=Test CA,ST=Some-State,C=US and the serial number of the certificate is 5. It is hoping that the client can use this information to locate the certificate and then use the public key in it to verify the signature. For this to work, the client may scan all the certificates in the keystore to try to find it. It

There is no <BinarySecurityToken> here. It means the s1 certificate is not sent.

Use the issuer DN and certificate serial number (5 here) to identify the certificate. It is up to the client to look it up.
means you must import s1’s certificate into the keystore on the client.

To check that the service is really verifying the signature, note messages like below in the console:

```
[INFO] Deploying module: soapmonitor-1.3
[INFO] Script module activated
[INFO] Deploying Web service: BizService
[INFO] Deploying Web service: ImageService
[INFO] Deploying Web service: ManualService
[INFO] Deploying Web service: SecureService
[INFO] Deploying Web service: SimpleService
[INFO] Deploying Web service: version.aar
[INFO] Deploying Web service: WrappedService
[INFO] [SimpleAxisServer] Started
[SimpleAxisServer] Started
[INFO] Listening on port 8080
[INFO] Undeploying Web service: SecureService
[INFO] Undeploying Web service: SecureService
[INFO] Undeploying Web service: SecureService
[INFO] Deploying Web service: SecureService
[INFO] Verification successful for URI "urn:uuid:47794454-49bc-49bb-b436-0580f976807f"
[INFO] Verification successful for URI "urn:uuid:47794454-49bc-49bb-b436-0580f976807f"
```

### Encrypting SOAP messages

At the moment the messages are signed, but they aren't encrypted and thus people on the Internet can see them. If the information is confidential, you should encrypt it. To do that, modify the policy in the WSDL file:

```xml
<wsdl:definitions ...>
  <wsp:Policy wsu:Id="p1">
    <sp:AsymmetricBinding>
      ...
    </sp:AsymmetricBinding>
    <sp:Wss10>
      ...
    </sp:Wss10>
    <sp:SignedParts>
      <sp:Body />  <!-- The <Body> element of the SOAP message should be encrypted -->
    </sp:SignedParts>
    <sp:EncryptedParts>
      <sp:Body />  <!-- The <Body> element of the SOAP message should be encrypted -->
    </sp:EncryptedParts>
  </wsp:Policy>
  ...  
</wsdl:definitions>
```

Generate the service stub and client stub again. Modify rampart-config.xml for the client:
This is a keystore alias. Get the certificate for the alias "s1" from the keystore and use the public key there to encrypt the message. Note that you don't need the password to get the public key.

Specify the cryptographic provider to perform encryption. Here, you still use the Merlin provider (JDK). You also specify its configurations (the path to the keystore and the keystore password). Here, everything is the same as the cryptographic provider for signing.

For the web service, modify services.xml:
Chapter 9  Signing and encrypting SOAP messages

However, there is a problem here. As you're encrypting the response message

<serviceGroup>
  <service name="SecureService">
    ...
    <wsp:Policy ... wsu:Id="p1">
      <sp:AsymmetricBinding>
        ...
      </sp:AsymmetricBinding>
      <sp:Wss10>
        ...
      </sp:Wss10>
      <sp:SignedParts>
        <sp:Body />
      </sp:SignedParts>
    </wsp:Policy>
  </service>
</serviceGroup>

Encrypt the response using c1’s public key

The <Body> element of the SOAP message should be encrypted

Specify the cryptographic provider to perform encryption. It is the same as the one used for signing. It is also identical to the one used by the client except that it uses a different keystore file.
using c1's public key, how can it find out c1's public key? You'll need to put c1's certificate in the keystore for the web service. In addition, this web service can only talk to a single client c1 (see the diagram below). If there is another client c2, it can encrypt the request using s1’s public key, but s1 will encrypt the response using the public key of c1 (NOT c2), making c2 fail to decrypt it:

To solve this problem, rampart supports a special way of operation. If c1 both signs and encrypts the request, it will sign it using its own private key. If it also includes its certificate in the request, then rampart can be instructed to look up this certificate in the request and use it to encrypt the response. Therefore, it will use c1's certificate to encrypt the response. If c2 sends it a request, it will encrypt the response using c2’s certificate:

To enable this operation, put a special value "useRegSigCert" into the <encryptionUser> element:
Now run the client and it should work. To verify that the messages are indeed encrypted, check them out in the TCP Monitor:

```xml
<serviceGroup>
  <service name="SecureService">
    ...
    <wsp:Policy wsu:Id="pl">
      ...
      <RampartConfig xmlns="http://ws.apache.org/rampart/policy">
        <user>s1</user>
        <encryptionUser>useReqSigCert</encryptionUser>
      </RampartConfig>
    </wsp:Policy>
  </service>
</serviceGroup>
```

It stands for "use request signing certificate". That is, use the certificate that signed the request message.
Security issues when performing both signing and encrypting

When you're performing both signing and encryption, there are security issues. For example, if you sign the <Body> and then encrypt it, then the resulting message will be like:
The problem is that, if you run the client multiple times, the digest will be the same. This is the way it should be. Given some particular plain text, anyone can calculate the digest and it should be the same. This means that a hacker could calculate the digest of some common plain text to build a lookup table like:

Plain text

```
<ns1:concat xmlns:ns1="http://ttdev.com/ss">
  <s1>xyz</s1>
  <s2>111</s2>
</ns1:concat>
```

```
<ns1:concat xmlns:ns1="http://ttdev.com/ss">
  <s1>xyz</s1>
  <s2>abc</s2>
</ns1:concat>
```

Then he can capture your message, get the digest and use the lookup table above to recover the plain text, even though you’ve encrypted the content of the <Body> element. It means the digest is actually leaking the plain text.

You may wonder if the hacker can do the same thing using the encrypted content of the <Body> element?
Chapter 9  Signing and encrypting SOAP messages

If you run the client multiple times, you'll see that the encrypted content of the
<Body> element will change every time. This is a basic requirement of
encryption algorithms to prevent such a lookup attack (called "dictionary
attack").

Now the question is how to prevent the digest from leaking information? There
are three alternative solutions.

The first solution is to perform encryption first and then sign on the encrypted
<Body> content. As the encrypted content changes every time, the digest will
change every time. However, this is not a very good solution as digital
signatures should be performed on what is seen by the users (i.e., plain text, not
encrypted text). For the case on hand, as it is the client (not user) signing it, it
may be good enough.

The second solution is to sign and then encrypt and finally also encrypt the
signature. This works for the case on hand. However, if the web service was
supposed to verify the signature but needed to pass the encrypted data to a 3rd
party, then the web service wouldn't have the key to decrypt the signature and
couldn't verify it.

The third solution is to include a random element (usually called "nonce" or
"salt") into the plain text so that the digest changes every time. For example,
you could add a third element to the request:

```xml
<ns1:concat xmlns:ns1="http://ttdev.com/ss">
  <s1>xyz</s1>
  <s2>111</s2>
  <salt>kgyu8FDsfdFafa389r</salt>
</ns1:concat>
```

This is the most flexible solution but it means a lot of extra work on you.
Anyway, in order to implement the first solution (encrypt and then sign), modify
the policy:

```xml
<soapenv:Body ...>
  <xenc:EncryptedData Id="EncDataId-26622782" ...>
    <xenc:EncryptionMethod .../>
    <ds:KeyInfo ...>
      ...
    </ds:KeyInfo>
    <xenc:CipherData>
      dKeF1WLDqSV...
    </xenc:CipherData>
  </xenc:EncryptedData>
</soapenv:Body>
```

The encrypted content of the
<Body> element
To implement the second solution, modify the policy:

```xml
<wsp:Policy wsu:Id="p1">
  <sp:AsymmetricBinding>
    ...
    <sp:EncryptBeforeSigning/>
    ...
  </sp:AsymmetricBinding>
  <sp:Wss10>
    ...
  </sp:Wss10>
  <sp:SignedParts>
    <sp:Body />
  </sp:SignedParts>
  <sp:EncryptedParts>
    <sp:Body />
  </sp:EncryptedParts>
  <sp:EncryptedElements>
    <sp:XPath>
      //*[local-name()='Signature']
    </sp:XPath>
  </sp:EncryptedElements>
</wsp:Policy>
```

Look for any descendant of XML root element (<Envelope> here)

BUG ALERT: Due to a bug in the current version of Rampart, the `<EncryptedElements>` feature is not working.

**Protecting WS-Addressing header elements**

If you’re using WS-Addressing, most likely you’d like to ensure that the WS-Addressing header elements are not tampered with. To do that, you can modify the policy to require signing on header elements:
Chapter 9  Signing and encrypting SOAP messages

To protect all such elements, you may list them one by one:

```xml
<sp:SignedParts>
  <sp:Body />
  <sp:Header Name="To" Namespace="http://www.w3.org/2005/08/addressing"/>
  <sp:Header Name="MessageID" Namespace="http://www.w3.org/2005/08/addressing"/>
  <sp:Header Name="Action" Namespace="http://www.w3.org/2005/08/addressing"/>
  <sp:Header Name="RelatesTo" Namespace="http://www.w3.org/2005/08/addressing"/>
</sp:SignedParts>
</sp:EncryptedParts>
</wsp:Policy>
```

However, this is too much trouble. A better way is not to specify the Name attribute and specify only the Namespace attribute. This way all header elements in the WS-Addressing namespace will be signed:

```xml
<soapenv:Envelope xmlns:wsa="http://www.w3.org/2005/08/addressing" ...>
  <soapenv:Header>
    <wsse:Security>...
    <wsa:To>http://localhost:1234/axis2/services/SecureService</wsa:To>
    <wsa:MessageID>urn:uuid:59F3153E977EDEDE471197688498788</wsa:MessageID>
  </soapenv:Header>
</soapenv:Envelope>
```

To protect all such elements, you may list them one by one:

```xml
<sp:SignedParts>
  <sp:Body />
  <sp:Header Name="To" Namespace="http://www.w3.org/2005/08/addressing"/>
  <sp:Header Name="MessageID" Namespace="http://www.w3.org/2005/08/addressing"/>
  <sp:Header Name="Action" Namespace="http://www.w3.org/2005/08/addressing"/>
  <sp:Header Name="RelatesTo" Namespace="http://www.w3.org/2005/08/addressing"/>
</sp:SignedParts>
</sp:EncryptedParts>
</wsp:Policy>
```

BUG ALERT: Due to a bug in the current version of Rampart, this feature is not working. Therefore, for the moment, you'll have to list them one by one.

Sending login information

Suppose that the web service will perform the requested operation only for selected users only. To do that, you can configure your client to send the user name and password to the web service. Such information is called a Username Token. To require a Username token in the request message, modify the policy:
How to specify the user name? For the moment rampart will always use the <user> configuration (in your rampart-config.xml file):

```xml
<wsp:Policy>
    <wsp:AsymmetricBinding>
        ...
    </wsp:AsymmetricBinding>
    <sp:AsymmetricBinding>
        ...
    </sp:AsymmetricBinding>
    <sp:SignedParts>
        ...
    </sp:SignedParts>
    <sp:EncryptedParts>
        ...
    </sp:EncryptedParts>
    <sp:SignedSupportingTokens>
        <wsp:Policy>
            <sp:UsernameToken sp:IncludeToken="http://docs.oasis-open.org/ws-sx/ws-securitypolicy/200702/IncludeToken/AlwaysToRecipient"/>
        </wsp:Policy>
    </sp:SignedSupportingTokens>
</wsp:Policy>
```

A Username Token is not like the certificate token which is required for signing or encryption. Therefore it is just a supporting token. Here, you also require that it be signed to make sure that it has not been tampered with.

There can be other types of supporting tokens. Username token is just one possible type.

How to specify the user name? For the moment rampart will always use the <user> configuration (in your rampart-config.xml file):

```xml
            xmlns="http://ws.apache.org/rampart/policy">
    <RampartConfig>
        <user>c1</user>
        <encryptionUser>s1</encryptionUser>
        <passwordCallbackClass>
            com.ttdev.secure.client.PasswordCallbackHandler
        </passwordCallbackClass>
        <signatureCrypto>
            ...
        </signatureCrypto>
        <encryptionCrypto>
            ...
        </encryptionCrypto>
    </RampartConfig>
</wsp:Policy>
```

This is a problem as you probably want the client to allow different users to use it to talk to the web service. In the latest snapshot of rampart these concepts can be separated:
Chapter 9  Signing and encrypting SOAP messages

As that version has not been released yet, here you can only use \texttt{c1} for both purposes. So rampart has the user name, how does it know the password? It can use the password callback. So modify PasswordCallbackHandler.java in the client package:

```java
public class PasswordCallbackHandler implements CallbackHandler {
    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            WSPasswordCallback pwcb = (WSPasswordCallback) callbacks[i];
            String id = pwcb.getIdentifer();
            switch (pwcb.getUsage()) {
                case WSPasswordCallback.SIGNATURE:
                case WSPasswordCallback.DECRYPT:
                    if (id.equals("c1")) {
                        pwcb.setPassword("c1-pass");
                    }
                    break;
                case WSPasswordCallback.USERNAME_TOKEN:
                    if (id.equals("c1")) {
                        pwcb.setPassword("c1-as-user-pass");
                    }
                    break;
            }
        }
    }
}
```

How can the web service verify the password? Again, rampart replies on the password callback to get the correct password for comparison. So, modify PasswordCallbackHandler.java in the \texttt{com.ttdev.secure} package:

```xml
    xmlns="http://ws.apache.org/rampart/policy">
    <RampartConfig>
        <user>u1</user>
        <userCertAlias>c1</userCertAlias>
        <encryptionUser>s1</encryptionUser>
        <passwordCallbackClass>
            com.ttdev.secure.client.PasswordCallbackHandler
        </passwordCallbackClass>
        <signatureCrypto>
            ...
        </signatureCrypto>
        <encryptionCrypto>
            ...
        </encryptionCrypto>
    </RampartConfig>
</wsp:Policy>
```

It is used only as the user name in the Username token

It is used only as the alias for the client certificate

When rampart needs to sign or decrypt, \texttt{c1} is acting as the alias.

When rampart needs to send a Username token, \texttt{c1} is acting as the user name.

Here use a different password to verify that they are different.
Now generate the service stub and client stub again. Run it. You should see the Username token in the TCP Monitor:

```java
public class PasswordCallbackHandler implements CallbackHandler {
    public void handle(Callback[] callbacks)
        throws IOException, UnsupportedCallbackException {
        for (int i = 0; i < callbacks.length; i++) {
            WSPasswordCallback pwcb = (WSPasswordCallback) callbacks[i];
            String id = pwcb.getIdentifier();
            switch (pwcb.getUsage()) {
                case WSPasswordCallback.DECRYPT:  // When rampart needs to sign or decrypt, it needs its own (s1) password.
                    if (id.equals("s1")) {
                        pwcb.setPassword("s1-pass");
                    }
                    break;
                case WSPasswordCallback.SIGNATURE: {  // When rampart needs to verify a Username token, it needs to return the password for the known users (c1).
                    if (id.equals("c1") {  
                        pwcb.setPassword("c1-as-user-pass");
                    }
                    break;
                }
            }
        }
    }
}
```

Now generate the service stub and client stub again. Run it. You should see the Username token in the TCP Monitor:
Chapter 9  Signing and encrypting SOAP messages

If you don't want others to even see the user name of "c1", you can encrypt the Username token. All that is required is to change <SignedSupportingTokens> to <SignedEncryptedSupportingTokens> in the policy.

What if different users have different permissions? You can retrieve the user name in your own code and decide what permissions he has. To do that, you need to understand the data structure created by rampart after processing the request message. There could be multiple rampart module instances running. Each will store its result into an element of a Vector (see the diagram below). Each rampart module instance may perform multiple actions, e.g., verify its signature, verify a Username token or decrypt a message. Therefore, for each action it will create a WSSecurityEngineResult to represent the result of that action. So, for each instance it creates a vector-like structure to store all such results. This is the WSHandlerResult. For example, in the diagram, the first action is SIGN, which means verifying a signature, the result contains the certificate used and etc. The second action is UT, which means verifying a Username token, the result contains the user name:

```xml
<soapenv:Envelope ...>
  <soapenv:Header>
    <wsse:Security ...>
      <xenc:EncryptedKey Id="EncKeyId-29857804">
        ...
      </xenc:EncryptedKey>
      <wsse:UsernameToken xmlns:wsu="..." wsu:Id="UsernameToken-6659511">
        <wsse:Username>c1</wsse:Username>
        <wsse:Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordDigest">
          6GW32nj7XJOsTyIjDZrcQW63X0E=
        </wsse:Password>
      </wsse:UsernameToken>
      <wsse:Nonce>D2oMduF226uzRd4Rs3Bkw==</wsse:Nonce>
      <wsu:Created>2007-12-15T06:16:55.765Z</wsu:Created>
    </wsse:Security>
  </soapenv:Header>
  ...</soapenv:Envelope>
```

For security, the password is not sent as clear text but as a digest. To fight against dictionary attack, a nonce and a time stamp are included when calculating the digest:

```
password + nonce + time stamp
```

In addition, the web service can remember the nonces seen in a short recent period. If the same nonce is used again, it is a replay attack.

If you don't want others to even see the user name of "c1", you can encrypt the Username token. All that is required is to change <SignedSupportingTokens> to <SignedEncryptedSupportingTokens> in the policy.
Now, to retrieve the DN of the user in the back end object, modify SecureServiceImpl.java:

```java
public class SecureServiceImpl implements SecureServiceSkeletonInterface {
    public String concat(String s1, String s2) {
        checkUser();
        return s1 + s2;
    }
    private void checkUser() {
        MessageContext context = MessageContext.getCurrentMessageContext();
        Vector handlersResults = (Vector) context.getProperty(WSHandlerConstants.RECV_RESULTS);
        for (Iterator iter = handlersResults.iterator(); iter.hasNext();)
            WSHandlerResult handlerResult = (WSHandlerResult) iter.next();
        Vector actionsResults = handlerResult.getResults();
        for (Iterator iterator = actionsResults.iterator(); iterator.hasNext();)
            WSSecurityEngineResult actionResult = (WSSecurityEngineResult) iterator.next();
        int action = ((Integer) actionResult.get(WSSecurityEngineResult.TAG_ACTION)).intValue();
        if (action == WSConstants.UT) {
            Principal p = (Principal) actionResult.get(WSSecurityEngineResult.TAG_PRINCIPAL);
            if (p != null) {
                System.out.println("Checking " + p.getName());
                return; //return if the user has the required permission
            }
        }
    }
}
```

Now, run the client and you should see the output in the Tomcat console:
Chapter 9  Signing and encrypting SOAP messages

Modifying services.xml programatically

Currently you're adding the <Policy> and the <module ref="rampart" /> elements to the services.xml file manually. This is no good as it will be overwritten if you run <wsdl2code> again (it is not deleted due to a BUG). A better way is to let Ant modify the services.xml file every time it is generated by <wsdl2code>. To do that, create a file add-policy.xsl in the project root:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Such a file is called an "XSL Transformations (XSLT)". XSL stands for "Extensible Stylesheet Language". What this file does is to copy the services.xml file to the output. In order to add the <Policy> and the <module ref="rampart" /> elements to it, further modify the add-policy.xsl file:
To apply this add-policy.xml file, modify build.xml:
Chapter 9  Signing and encrypting SOAP messages

Now run the build.xml file and the services.xml will be setup properly. Run the client and it should continue to work.

Summary

WS-Policy allows you to specify non-functional requirements such as security on web services. You include a policy in the WSDL file and the generated client stub will use it. For the web service, you still need to include it into the services.xml file.

To sign or encrypt a message, specify in the policy the configuration settings such as algorithms to use, whether to include the certificate (token) and how (direct include or issuer DN plus serial number and etc.). You also specify which parts should be signed and which parts should be encrypted.

The Rampart module implements the WS-Security standard and can be used to satisfy security requirements expressed in policies. It gets information from the policy. In addition, you also need to provide further configurations to it using an XML file or a string. Such configurations include the user name alias, password callback class, what cryptographic provider to use (e.g., JDK), the location of the keystore and the keystore password.

When performing both signing and encrypting, to fight against dictionary attacks, you should encrypt the signature, encrypt before signing or include a nonce into the digest.
To send authentication information, you can use a Username token. This is also specified in a policy. Your password callback class should provide the password. The Username token should be signed and probably also be encrypted. You can retrieve the user name in your web service to perform authorization.

To modify XML file using Ant, you can use XSLT. This allows you to modify the services.xml file programmatically.
Chapter 10

Integrating Your Web Services with Tomcat and Spring
What's in this chapter?

In this chapter you'll learn how to run the Axis server inside Tomcat and let your web service invoke business logic in Spring beans.

Axis server as a mini-web server

Up until now you've been running the Axis server as a separate process listening for SOAP messages in HTTP requests on port 8080. Essentially it is acting as a mini-web server. If you're already running a web server such as Tomcat, you probably want to run the Axis server as a web application in Tomcat.

Installing Tomcat

If you already have Tomcat installed, skip to the next section. Otherwise, go to http://tomcat.apache.org to download a binary package of Tomcat. Download the zip version instead of the Windows exe version. Suppose that it is apache-tomcat-6.0.13.zip. Unzip it into a folder say c:\tomcat. Note that Tomcat 6.x works with JDK 5 or above.

Before you can run it, make sure the environment variable JAVA_HOME is defined to point to your JDK folder (e.g., C:\Program Files\Java\jdk1.5.0_02):
If you don’t have it, define it now. Now, open a command prompt, change to c:\tomcat\bin and then run startup.bat. If it is working, you should see:
Open a browser and go to http://localhost:8080 and you should see:

Let's shut it down by changing to c:\tomcat\bin and running shutdown.bat.
Running the Axis server inside Tomcat

Next, go to http://ws.apache.org/axis2 to download the "WAR (Web Archive) Distribution" (e.g. axis2-1.3-war.zip). There are just a handful of files in the zip file. Unzip it and put the files into c:\axis. The only important file there is the axis2-1.3.war file. To install it into Tomcat, copy it into c:\tomcat\webapps. Then start Tomcat by running startup.bat. You should see:

```
INFO: Deploying web application archive axis2.war
[INFO] Deploying module: addressing-1.3
[INFO] Deploying module: script-1.3
[INFO] Deploying module: metadataExchange-1.3
[INFO] Deploying module: ping-1.3
[INFO] Deploying module: soapmonitor-1.3
[INFO] script module activated
[INFO] Deploying Web service: version-1.3.aar
INFO: Starting Coyote HTTP/1.1 on http-8080
INFO: JK: ajp13 listening on /0.0.0.0:8089
INFO: Jk running ID=0 time=0/32 config=null
INFO: Server startup in 35270 ms
```

To further check that the Axis server is running, go to http://localhost:8080/axis2 in a browser. You should see:
Check `c:\tomcat\webapps`, you should see that there is an axis2 folder created with the following structure:
To deploy the web services you developed in the previous chapters, just copy their folders over:

- **services.xml**
- **SimpleService.wsdl**
- **SecureService**
- **addressing-1.3.mar**
- **rampart-1.3.mar**
- **modules**
- **lib**
  - ***.jar**
  - **.mar files**

A .mar file for each module

Jar files needed by Axis itself, the modules or your web services

To deploy a web service, put such a folder here

Configuration file for axis
Restart Tomcat for the changes to take effect. Run a client such as the SecureClient and it should continue to work.

**Invoking Spring beans from your web service**

Up until now all your web services perform very simple operations such as concatenating two strings. In practice, they should really invoke business logic such as placing an order for some goods. Typically such business logic may have been implemented as Spring beans. Next, let’s work on one such example.

In Eclipse copy the WrappedService project and paste it as SpringService. Link the "out" folder to C:\tomcat\webapps\axis2\WEB-INF\services\SpringService. Rename WrappedService.wsdl to SpringService.wsdl and changes the word "Wrapped" to "Spring" in the file. Then modify build.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project basedir="." default="jar.server">
...
<property name="name" value="SpringService" />
...
<target name="generate-service">
<wsdl2code
  wsdlfilename="${name}.wsdl"
  serverside="true"
  generatedservicexml="true"
  skipbuildxml="true"
  serversideinterface="true"
  namespace=packages="http://ttdev.com/ss=com.ttdev.spring"
  targetsourcefolderlocation="src"
```
Run it to generate the stubs.

To setup Spring, go to http://www.springframework.org to download it. Suppose that the file is spring-framework-2.0.6-with-dependencies.zip. Unzip it into say c:\spring-framework. To make the Spring classes available to your application, copy the following jar files into c:\tomcat\webapps\axis2\WEB-INF\lib:

```
   c:\
   spring-framework
   |___ dist
   |     |___ spring.jar
   |___ lib
      |___ cglib
      |     |___ cglib-nodep-2.1_3.jar
      |___ jakarta-commons
         |___ commons-logging.jar
```

You'll also need to access the Spring classes in Eclipse, so add spring.jar to the build path of your project in Eclipse. Then modify c:\tomcat\webapps\axis2\WEB-INF\web.xml as shown below. You add a <listener> element. When Tomcat notes that there is a <listener> element, when it is starting the Axis server (as a web application), it will create a listener object of the specified class (here, the ContextLoaderListener class provided by Spring) and call it. The ContextLoaderListener will initialize the Spring framework, or rather, it will create a Spring application context which is basically a collection of Spring beans. As the listener is loading the context, that's why it is called ContextLoaderListener:
Chapter 10 Integrating Your Web Services with Tomcat and Spring

When Spring is creating the application context, it will try to read a configuration file WEB-INF/applicationContext.xml to find out what beans are available. So, create that file now:

```xml
<?xml version="1.0" encoding="UTF-8"?>
    <bean id="concatBean" class="com.ttdev.spring.middletier.ConcatService"/>
    <bean id="appContextHolder" class="org.apache.axis2.extensions.spring.receivers.ApplicationContextHolder"/>
</beans>
```

Define a bean named "concatBean"

When Spring is creating the application context, it will try to read a configuration file WEB-INF/applicationContext.xml to find out what beans are available. So, create that file now:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>Apache-Axis2</display-name>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
    <servlet>...
    </servlet>
</web-app>
```

1: Look, I need to create a listener of this class.

2: Start axis2

3: Create & call ContextLoader Listener

4: Initialize Spring

When Spring is creating the application context, it will try to read a configuration file WEB-INF/applicationContext.xml to find out what beans are available. So, create that file now:

```xml
<?xml version="1.0" encoding="UTF-8"?>
    <bean id="concatBean" class="com.ttdev.spring.middletier.ConcatService"/>
    <bean id="appContextHolder" class="org.apache.axis2.extensions.spring.receivers.ApplicationContextHolder"/>
</beans>
```

Define a bean named "concatBean"

```
<?xml version="1.0" encoding="UTF-8"?>
    <bean id="concatBean" class="com.ttdev.spring.middletier.ConcatService"/>
    <bean id="appContextHolder" class="org.apache.axis2.extensions.spring.receivers.ApplicationContextHolder"/>
</beans>
```

Define another bean. It will get access to the application context and allow others to access it.

Next, create ConcatService.java in the com.ttdev.spring.middletier package:
package com.ttdev.spring.middletier;

public class ConcatService {
    public String concat(String s1, String s2) {
        return s1+s2;
    }
}

To make the classes in this middletier package available to Spring, they should be packed into a jar file and then copied into WEB-INF/lib. To do that, right click the middletier package and choose Export, then choose Java | JAR file:

Select
Export resources into a JAR file on the local file system.

Select an export destination:

General   JAR file
EJB
JSF
Java
Javatox
Mylyn
Plug-in Development
Team
Web
Web Services

Enter the destination path as shown below:
Chapter 10 Integrating Your Web Services with Tomcat and Spring

Click Finish to create the jar file.

To invoke the concatBean in your web service code, create SpringServiceImpl.java:

```java
public class SpringServiceImpl implements SpringServiceSkeletonInterface {
    public String concat(String s1, String s2) {
        ApplicationContext context = ApplicationContextHolder.getContext();
        ConcatService bean = (ConcatService) context.getBean("concatBean");
        return bean.concat(s1, s2);
    }
}
```

Get the application context

Get the "concatBean"

Call the business logic

Now, restart Tomcat for the changes to take effect. To test it, create a SpringClient in the client package:

```java
public class SpringClient {
    public static void main(String[] args) throws RemoteException {
```
SpringServiceStub stub = new SpringServiceStub();
String result = stub.concat("xyz", "abc");
System.out.println(result);
}
)
Run it and it should work.

Summary

The Axis server can be run inside Tomcat as a web application. This is most useful when you are already running Tomcat. In particular, when you're using Spring beans to implement your business logic, in order for your web service code to access the Spring beans, you have to run the Axis server inside Tomcat.

To access a Spring bean from your web service, the key is to gain access to the Spring application context. This can be done through a special application context holder bean.
References


• W3C. URIs, URLs, and URNs: Clarifications and Recommendations 1.0. http://www.w3.org/TR/2001/NOTE-uri-clarification-20010921.
• W3C. Web Services Addressing 1.0 – Core. http://www.w3.org/TR/ws-addr-core.
• W3C. Web Services Addressing 1.0 – Core. http://www.w3.org/TR/ws-addr-core.
• W3C. XSL Transformations (XSLT). http://www.w3.org/TR/xslt.
Alphabetical Index

3DES.................................................................................................................. 170
Ant......................................................................................................................... 70
    Property........................................................................................................ 70
    <move>................................................................................................. 195
    <replaceregexp>.................................................................................. 74
    <wsdl2code>....................................................................................... 70
    <xslt>................................................................................................. 195
Asymmetric encryption...................................................................................... 151
Asynchronous processing in the service.......................................................... 140
AXIOM................................................................................................................ 79
Axis....................................................................................................................... 28
    Installing................................................................................................. 28
Axis server........................................................................................................ 28
    Running in Tomcat.............................................................................. 203
    Setting up............................................................................................... 28
Axis2.xml.......................................................................................................... 64, 143, 204
Base64............................................................................................................. 124
Binding.............................................................................................................. 19
Build.xml.......................................................................................................... 69
CA...................................................................................................................... 155
Callback............................................................................................................ 172
CallbackHandler.............................................................................................. 172
Certificate......................................................................................................... 155
Certificate authority......................................................................................... 155
Certificate chain.............................................................................................. 166
Certificate path................................................................................................. 166
Client stub......................................................................................................... 55
    Specifying an explicit endpoint......................................................... 83
    Specifying the repository location................................................. 171
CN..................................................................................................................... 155
Code Generator Wizard.................................................................................... 30
ConfigurationContext....................................................................................... 171
DataHandler...................................................................................................... 127
Debugging a web service.................................................................................. 66
Deploying a web service................................................................................... 66
    When Axis2 is running in standalone mode........................................ 53
    When Axis2 is running in Tomcat..................................................... 205
Dictionary attack............................................................................................. 186
Digital signature.............................................................................................. 153
Distinguished name......................................................................................... 155
DN..................................................................................................................... 155
Document style............................................................................................... 16
DSA.................................................................................................................... 158
Eclipse.............................................................................................................. 28
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linking a folder to outside</td>
<td>62</td>
</tr>
<tr>
<td>Relinking a folder to another location</td>
<td>88</td>
</tr>
<tr>
<td>Encoded</td>
<td>117</td>
</tr>
<tr>
<td>Encrypting SOAP messages</td>
<td>179</td>
</tr>
<tr>
<td>Encryption key</td>
<td>156</td>
</tr>
<tr>
<td>Endpoint</td>
<td>10</td>
</tr>
<tr>
<td>Endpoint provided by Axis</td>
<td>32</td>
</tr>
<tr>
<td>Extensible Stylesheet Language</td>
<td>194</td>
</tr>
<tr>
<td>Fault</td>
<td>108</td>
</tr>
<tr>
<td>Fault message</td>
<td>108</td>
</tr>
<tr>
<td>Getting the wsdl of a web service</td>
<td>29</td>
</tr>
<tr>
<td>Hot deployment</td>
<td>59, 65</td>
</tr>
<tr>
<td>Hot update</td>
<td>64</td>
</tr>
<tr>
<td>IANA</td>
<td>23</td>
</tr>
<tr>
<td>Input message</td>
<td>12</td>
</tr>
<tr>
<td>International Assigned Numbers Association</td>
<td>23</td>
</tr>
<tr>
<td>Interoperability</td>
<td>129</td>
</tr>
<tr>
<td>With .NET</td>
<td>93, 130</td>
</tr>
<tr>
<td>Keystore</td>
<td>157</td>
</tr>
<tr>
<td>Alias</td>
<td>157</td>
</tr>
<tr>
<td>Alias password</td>
<td>158</td>
</tr>
<tr>
<td>Generating a certificate request</td>
<td>160</td>
</tr>
<tr>
<td>Generating a key pair</td>
<td>157</td>
</tr>
<tr>
<td>Importing a certificate</td>
<td>164</td>
</tr>
<tr>
<td>Password</td>
<td>158</td>
</tr>
<tr>
<td>Keytool</td>
<td>157</td>
</tr>
<tr>
<td>Listing the web services</td>
<td>54</td>
</tr>
<tr>
<td>Literal</td>
<td>34, 116</td>
</tr>
<tr>
<td>Local name</td>
<td>11</td>
</tr>
<tr>
<td>MD5</td>
<td>158</td>
</tr>
<tr>
<td>Message digest</td>
<td>152</td>
</tr>
<tr>
<td>Message receiver</td>
<td>139</td>
</tr>
<tr>
<td>Message transmission optimization mechanism</td>
<td>127</td>
</tr>
<tr>
<td>MIME message</td>
<td>124</td>
</tr>
<tr>
<td>Module</td>
<td>141</td>
</tr>
<tr>
<td>Deploying into Axis2</td>
<td>167</td>
</tr>
<tr>
<td>Deploying into the client</td>
<td>167</td>
</tr>
<tr>
<td>Engaging</td>
<td>141</td>
</tr>
<tr>
<td>Module archive</td>
<td>167</td>
</tr>
<tr>
<td>MTOM</td>
<td>127</td>
</tr>
<tr>
<td>Enabling in the client</td>
<td>127</td>
</tr>
<tr>
<td>Enabling in the service</td>
<td>129</td>
</tr>
<tr>
<td>MustUnderstand</td>
<td>173</td>
</tr>
<tr>
<td>Namespace</td>
<td>11</td>
</tr>
<tr>
<td>Namespace prefix</td>
<td>12</td>
</tr>
<tr>
<td>Nonce</td>
<td>186</td>
</tr>
<tr>
<td>OMEElement</td>
<td>78</td>
</tr>
</tbody>
</table>
Setting up a CA.................................................................161
SHA1..........................................................158
Signing.........................................................153
Signing and encryption...............................184
Signing SOAP messages..............................168
Simple Object Access Protocol....................19
SOAP..............................................................19
SOAP action..................................................34, 144
SOAP body element........................................34
SOAP envelope.............................................34
SOAP fault..................................................109
SOAP header element....................................34
SOAP message format...................................34
Spring.............................................................
  Application context......................................207
  ApplicationContext.xml..............................208
  ContextLoaderListener..............................207
  Invoking Spring beans from your web service.206
Symmetric encryption......................................150
TCP Monitor..................................................79
Time out problem with HTTP.......................132
Tomcat.........................................................200
Undeploying a web service...........................58
Uniform Resource Identifier.........................23
Uniform Resource Name...............................23
Unwrapping..................................................91
URI...............................................................23
URL.............................................................23
URN.............................................................23
  Namespace identifier..................................24
  Namespace specific string........................24
  NID...........................................................24
  NSS..........................................................24
Username Token...............................................188
Using a callback............................................142
Using a separate listener...............................142
Web service..................................................10
Web Service Security.......................................170
Web Services Description Language..............25
Wrapping.....................................................91
WS-Addressing..............................................
  <Action>..................................................144
  <MessageID>..............................................134, 144
  <RelatesTo>................................................134
  <ReplyTo>..................................................133, 144
  <To>.........................................................144
WS-I.............................................................17
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>WS-Policy</td>
<td>168</td>
</tr>
<tr>
<td>&lt;AsymmetricBinding&gt;</td>
<td>170</td>
</tr>
<tr>
<td>&lt;EncryptBeforeSigning&gt;</td>
<td>186</td>
</tr>
<tr>
<td>&lt;EncryptedElements&gt;</td>
<td>187</td>
</tr>
<tr>
<td>&lt;EncryptedParts&gt;</td>
<td>179</td>
</tr>
<tr>
<td>&lt;InitiatorToken&gt;</td>
<td>170</td>
</tr>
<tr>
<td>&lt;MustSupportRefEmbeddedToken&gt;</td>
<td>170</td>
</tr>
<tr>
<td>&lt;MustSupportRefIssuerSerial&gt;</td>
<td>170</td>
</tr>
<tr>
<td>&lt;Policy&gt;</td>
<td>168</td>
</tr>
<tr>
<td>&lt;RecipientToken&gt;</td>
<td>170</td>
</tr>
<tr>
<td>&lt;SignedEncryptedSupportingTokens&gt;</td>
<td>192</td>
</tr>
<tr>
<td>&lt;SignedParts&gt;</td>
<td>168</td>
</tr>
<tr>
<td>For SOAP header element</td>
<td>187</td>
</tr>
<tr>
<td>&lt;SignedSupportingTokens&gt;</td>
<td>188</td>
</tr>
<tr>
<td>&lt;UsernameToken&gt;</td>
<td>188</td>
</tr>
<tr>
<td>&lt;Wss10&gt;</td>
<td>170</td>
</tr>
<tr>
<td>&lt;X509&gt;</td>
<td>170</td>
</tr>
<tr>
<td>&lt;X509Token&gt;</td>
<td>170</td>
</tr>
<tr>
<td>WS-Security</td>
<td>173</td>
</tr>
<tr>
<td>&lt;BinarySecurityToken&gt;</td>
<td>173</td>
</tr>
<tr>
<td>&lt;EncryptedData&gt;</td>
<td>173, 183</td>
</tr>
<tr>
<td>&lt;EncryptedKey&gt;</td>
<td>183</td>
</tr>
<tr>
<td>&lt;Security&gt;</td>
<td>173</td>
</tr>
<tr>
<td>&lt;Signature&gt;</td>
<td>173</td>
</tr>
<tr>
<td>WSDL</td>
<td>25, 32</td>
</tr>
<tr>
<td>Target namespace</td>
<td>32</td>
</tr>
<tr>
<td>Validating a WSDL file</td>
<td>45</td>
</tr>
<tr>
<td>WSE</td>
<td>130</td>
</tr>
<tr>
<td>WSPasswordCallback</td>
<td>172</td>
</tr>
<tr>
<td>WSS</td>
<td>170</td>
</tr>
<tr>
<td>X509</td>
<td>170</td>
</tr>
<tr>
<td>XML catalog</td>
<td>118</td>
</tr>
<tr>
<td>XML-binary optimized packaging</td>
<td>124</td>
</tr>
<tr>
<td>Xop</td>
<td>124</td>
</tr>
<tr>
<td>XPath</td>
<td>187</td>
</tr>
<tr>
<td>Xsd</td>
<td>14</td>
</tr>
<tr>
<td>XSL</td>
<td>194</td>
</tr>
<tr>
<td>XSL Transformations</td>
<td>194</td>
</tr>
<tr>
<td>XSLT</td>
<td>194</td>
</tr>
<tr>
<td>.mar file</td>
<td>167</td>
</tr>
<tr>
<td>.NET</td>
<td>93</td>
</tr>
<tr>
<td>Interoperability</td>
<td>93</td>
</tr>
</tbody>
</table>