Installation, configuration, and deployment

JBoss in Action

SAMPLE CHAPTER

Javid Jamae
Peter Johnson

MANNING

JBoss in Action

Configuring the JBoss Application Server

by Javid Jamae
and Peter Johnson

Chapter 9

brief contents

9

Configuring Web Services

This chapter covers

- Understanding Web Services
- Developing a simple web service
- Developing web service clients
- Exploring JBoss Web Service-specific annotations
- Securing a web service
- Encrypting SOAP messages

It was August of 2000 in Orlando, Florida. I (Peter) recall sitting in a frigid conference room (the air conditioning was on high to combat the sweltering temperature outside) at the Professional Developer's Conference (PDC) when Microsoft rolled out their vision of the future complete with the .NET Framework and a thing called Web Services. At the time, they didn't have Visual Studio completely working with Web Services. When they did roll out the beta version of Visual Studio .NET, you could create a simple echoing web service with a few mouse clicks. The annotation capabilities in the .NET Framework made creating Web Services simple; the tools and the Framework handled the glue code that made it all possible.

The next March I attended JavaOne in San Francisco. Almost every presentation mentioned the new hot topic: Web Services. Many of the presenters pointed out that EJBs, specifically stateless session beans, were a natural fit for Web Services because they already supplied a similar capability within distributed applications.

In addition, Sun Microsystems published a document that stated how a stateless session bean could be converted into a web service endpoint. This process consisted of around a dozen steps, running a wide variety of tools and performing a wide variety of configuration steps, and only worked with Sun's application server. Needless to say, I never got my EJB-based web service working.

Development of Java-based Web Services has come a long way since then. The annotation support introduced in Java SE 5.0, and embraced by a wide variety of Java technologies, makes creating and consuming Web Services in Java as easy as in the .NET Framework.

In this chapter, we describe Web Services and present a simple web service example, showing how to develop and deploy that web service within JBoss AS. We focus on Web Services defined using the Java API for XML-based Web Services (JAX-WS) as delineated in JSR-181, implemented by JBoss Web Services 3.0, and provided in JBoss Application Server 5.0. If you're interested in the J2EE 1.4-compliant Web Services (JSR-109), see http://jbws.dyndns.org/mediawiki/index.php?title=JAX-RPC_User_Guide, where this topic is well documented. After the example, we present various configuration topics such as describing web service annotation, securing a web service, and encrypting web service messages.

If you're already familiar with Web Services or only want to learn how to configure Web Services within JBossWS, you can skip to section 9.3. If you're an administrator, you might want to skip to section 9.4 and get right into the security configuration.

9.1 Understanding Web Services

What is a web service? A cynic might say that it's nothing more than remote method invocation (RMI) performed over HTTP using a text-based (an XML document in this case) transport mechanism. And the cynic would be right. Web Services aren't necessarily a revolution but do represent an evolutionary step towards interoperability of heterogeneous systems.

There are two key concepts to Web Services. First, if two (or more) parties agree on the format for a certain type of data, then they can exchange data. For example, if hospitals and doctors agree on the layout of patient data, then a doctor could easily transfer information about a patient to the hospital where the patient is scheduled for surgery. Various industry groups have defined such data layouts for data of interest to their industries. Using XML as the basic layout for such data has increased the chances that such vertical industry data layouts will be developed and accepted.

Second, this data, which is software readable, can be transmitted over a protocol that can get through corporate firewalls. Performing standard Remote Method Invocation (RMI) between companies isn't usually possible because the firewalls block the ports used for RMI. But HTTP ports 80 and 443 are typically opened in firewalls to allow customers and other users to access a company's web site. Web servers then become responsible, not only for human-generated traffic to service web pages, but also for application-generated traffic in the form of Web Services.

9.1.1 Understanding web service terminology

As with other technologies, Web Services have their own jargon and set of mnemonics that you have to learn. Although we don't provide an exhaustive list, we do want to highlight a few of the terms that you'll encounter in this chapter. Figure 9.1 illustrates some of the relevant terminology for Web Services.

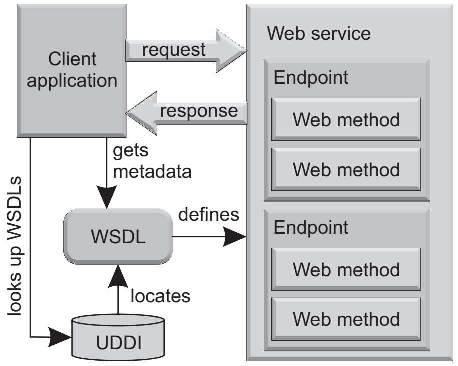


Figure 9.1 This figure illustrates Web Services terminology, showing how the client relates to the server and how Web Services are constructed within the server.

A web service is a collection of *endpoints*. Each endpoint is implemented in Java as a class. An endpoint can contain one or more *web methods*. You can also use an interface to define an endpoint and use a class to implement that endpoint. The endpoint interface is always used on the client side to construct a proxy that can marshal the arguments to the web method and unmarshal the result.

The Web Services Description Language (WSDL) file is an XML document that describes the web service. Although you can create a WSDL from scratch to define your web service and then generate the necessary stubs from it, it's usually easier to define the web service in terms of the endpoint written in Java and generate the WSDL from that. We present both mechanisms in this chapter.

The Universal Description, Discovery, and Integration (UDDI) registry is a mechanism used to publish Web Services. Think of it as a phone book with the WSDL as a phone number. If you know the phone number (WSDL), you can make the call directly. If you don't know the phone number, you can look it up in the phone book (UDDI) and then make the call.

We don't cover UDDI usage in this book because the subject of Web Services is much bigger than what can be covered easily in a single chapter. This chapter presents a simple introduction to Web Services and highlights various configuration topics when using JBossWS.

9.1.2 Understanding SOAP binding styles

SOAP is a protocol that enables the exchange of data between heterogeneous systems. It provides two different SOAP binding styles—document and Remote Procedure Call (RPC)—to pass data to a web method. In the RPC style, clients typically pass numerous parameters to a web method, and those parameters typically use simple data types such as strings and integers. Such web services tend to be chatty, or fine grained, in that the client calls on the service frequently to perform a single task.

The document style of web services tends to be coarse grained; the client packages up all the information into a single object, which is then passed to the web method. The web method has all the necessary information to perform the task. In many cases, document-style calls tend to be asynchronous; the client makes the call and then goes off to do other things. The client either checks later to see if there was a response to the call or registers to be notified when the response comes in. Asynchronous, document-style calls are preferred when using Web Services between companies.

Now that you have a basic understanding of Web Services, let's look at a simple web service, which we use as an example for the rest of the chapter when discussing various configuration topics.

9.2 Developing a web service

The example web service returns the sales tax for a purchase based on the customer's state. You input the two-character postal state code (such as CA for California), and the service returns the sales tax rate. (Don't we wish it were that easy! We don't know about other states, but in California, each county and, sometimes, even each city has its own sales tax rate. We could expand the service to also require the postal ZIP code, which would help pinpoint the exact sales tax rate. But to keep the example simple, we assume that sales tax rates are also simple, with one per state.)

Once we've shown how to code the web service, we then show how to deploy it and how to write clients to access it. Yes, we mean *clients*, as in plural. Because the biggest selling point of Web Services is interoperability among heterogeneous systems, you'll find that people who use technologies other than Java will want to access your web services. Therefore, we show you how to write clients in Java and in C# for the web service.

9.2.1 Coding the web service

There are two approaches to developing a web service, as follow:

- *The top-down approach*—You first develop the WSDL and use a utility, such as the `wsconsume` utility supplied by JBossWS, to generate the necessary glue code and stubs. You then fill in the code for the business logic in the stub classes. This approach works best when you're collaborating with various other entities to define the Web Services because the WSDL becomes the contract between those involved.
- *The bottom-up approach*—You code the web service first and then generate the WSDL from the web service. You can generate the WSDL using a utility, such as the `wsprovide` utility supplied by JBossWS, or you can package the web service and deploy it. The Web Services deployer will automatically generate the WSDL. This approach works best if you're defining a web service that you'd like others to use and there's no preexisting WSDL.

For this example, we use the bottom-up approach. Once you generate the WSDL, we briefly show how to use the WSDL for the top-down approach.

Listing 9.1 contains the code for the web service.

Listing 9.1 A simple web service

```
package org.jbia.ws;
import java.util.HashMap;
import javax.jws.*;             <-- Imports web service package
@WebService      ❶
```

```
public class SalesTax {
  private HashMap<String, Double> tax;
  public SalesTax() {init();}
  public void init() {
    tax = new HashMap<String,Double>();
    tax.put("CA", 7.75);                       ❷
    tax.put("NH", 0.0);
  }
  @WebMethod      ❸
  public double getRate(String state)   {
    Double rate = tax.get(state);
    if (rate == null) rate = -1.0;              Returns sales
    return rate;                                tax rate
  }
}
```

Notice the annotations, `@WebService` ❶ and `@WebMethod` ❸, which define the web service and what methods it supports. This web service is based on a POJO, and not an EJB, but you could have as easily added these annotations to a stateless session bean. We choose to use a POJO to keep the example simple.

In a real application, the code that initializes the tax rate hash table ❷ would be loaded from a database, but (again, to keep the example simple) we initialize it with a few hard-coded values. Although we could put in values for all 50 states, that would lengthen the example without adding anything to the discussion at hand.

9.2.2 *Packaging the web service*

You need to package the web service as a web application. Before you can do that, you need to create a web.xml file declaring the web service class as a servlet. The web.xml file is shown in listing 9.2.

Listing 9.2 The web.xml for the web service

```
<web-app>
  <servlet>
    <servlet-name>SalesTax</servlet-name>                      Identifies web
    <servlet-class>org.jbia.ws.SalesTax</servlet-class>   <-- service class
  </servlet>
  <servlet-mapping>
    <servlet-name>SalesTax</servlet-name>     Identifies context used
    <url-pattern>/tax</url-pattern>      <--  to access web service
  </servlet-mapping>
</web-app>
```

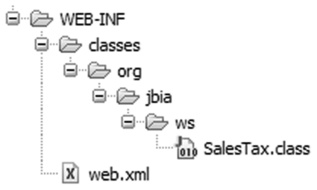


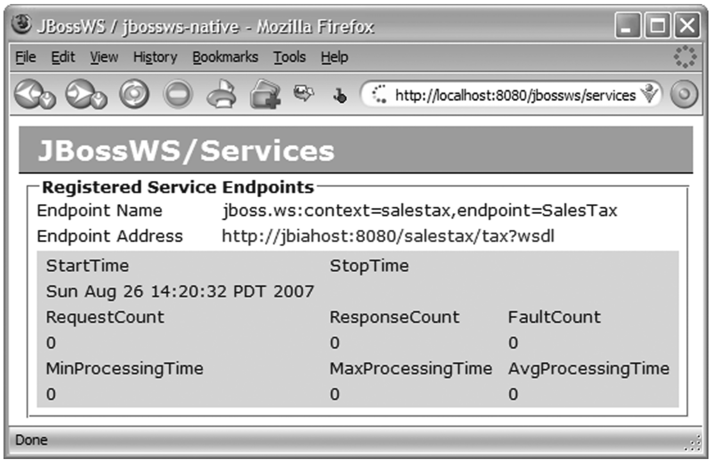
Figure 9.2 The salestax.war file contains only two files: the class file that implements the web service and the standard descriptor file.

You're now ready to package the web service. Create a salestax.war file as indicated in figure 9.2.

Optionally, you could package the web service in a *.wsr file. For example, instead of packaging the example in salestax.war, you could package it in salestax.wsr. What's the difference? From a content point of view, nothing. A *.wsr file has the exact same content as a *.war file. But the deployer deploys *.wsr files after *.war files. If it's important to have a web service deployed after the web applications, name the file *.wsr.

DEPLOYING AND ACCESSING THE WEB SERVICE

Deploying the web service is as easy as deploying any other web application; you copy the WAR file to the deploy directory. The application server creates the WSDL automatically. You can view web services deployed to the application server by going to the URL http://localhost:8080/jbossws/services. Figure 9.3 shows the resulting page with the SalesTax web service displayed.


JBossWS / jbossws-native - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:8080/jbossws/services
Done

JBossWS/Services

Registered Service Endpoints

Endpoint Name jboss.ws:context=salestax,endpoint=SalesTax
Endpoint Address http://jbiahost:8080/salestax/tax?wsdl

StartTime	StopTime	
Sun Aug 26 14:20:32 PDT 2007		
RequestCount	ResponseCount	FaultCount
0	0	0
MinProcessingTime	MaxProcessingTime	AvgProcessingTime
0	0	0

Figure 9.3 This screenshot displays information about the example web service. The Endpoint Address value is a hyperlink to the WSDL for the web service.

Click the URL identified by Endpoint Address to access the WSDL. The WSDL URL is important because you'll need it to create the client application.

9.2.3 Manually generating the WSDL

Instead of letting the application server generate the WSDL, you could generate it manually and include it in your WAR file. Run the `wsprovide` utility as follows:

```
wsprovide -o wsgen -c XXX -w org.jbia.ws.SalesTax
```

The `-o` option indicates that the output goes in the wsgen directory. The `-c` option provides the class path (`XXX` in the example) where you can find the endpoint class, `SalesTax` in this case. The `-w` option indicates to generate a WSDL file.

The generated WSDL contains a placeholder for the web service URL; you must supply the proper URL, as shown in listing 9.3.

Listing 9.3 Excerpt for the WSDL using an updated web service URL

```
...
 <service name='SalesTaxService'>
  <port binding='tns:SalesTaxBinding' name='SalesTaxPort'>
   <soap:address location='http://localhost:8080/salestax/tax'/>
  </port>
 </service>
...
```

Web service URL

The wsprovide utility is only one of many web service utilities provided by the application server. Table 9.1 lists those utilities, without the suffixes .bat and .sh, and describes their purposes. You can find the utilities in the bin directory. For usage details, run the utility passing -h as a parameter. You'll see examples of how to use each of the tools (other than wstools) in this chapter.

Table 9.1 Web service-related scripts

Script name	Purpose
`wsconsume`	Generates stubs or interfaces from a WSDL file. Used in top-down development.
`wsprovide`	Generates a WSDL file from web service classes. Used in bottom-up development.
`wsrunclient`	Runs a web service client and provides the necessary class path for that client.
`wstools`	Script used for JSR-109 Web Services development.

Now that you've generated the WSDL file, let's look at how you create a web service using the top-down approach.

9.2.4 *Developing a web service using the top-down strategy*

To develop a web service using a top-down approach, you need to start with the WSDL file. Then you run wsconsume to generate the class stubs from the WSDL and provide the business logic for the web methods.

To take the WSDL you generated and create the SalesTax class using the top-down approach, you generate the stubs using wsconsume as follows:

```
wsconsume -o stubs -k wsgen/SalesTaxService.wsdl
```

The -o option causes the generated files to be placed in a directory named *stubs*. The -k option indicates that the generated Java source files are to be kept; if this option isn't specified, the source files are removed and only the class files remain. Finally, the WSDL file is the one generated by wsprovide earlier.

When examining the files generated, you'll notice that one of them is called SalesTax.java. This file contains an interface that defines the web service. You need to make a few changes to the original SalesTax class to use this interface, as noted in listing 9.4.

Listing 9.4 A simple web service with top-down changes

```
package org.jbia.ws;
import java.util.HashMap;
import javax.jws.*;
@WebService(endpointInterface="org.jbia.ws.SalesTax",          ❶
         portName="SalesTaxPort",      ❷
      wsdlLocation="WEB-INF/wsdl/SalesTaxService.wsdl")         ❸
public class SalesTaxImpl implements SalesTax {       ❹
  private HashMap<String, Double> tax;
  public SalesTaxImpl() {...}
  public void init() {...}                         No WebMethod
  public double getRate(String state) {...}    <-  annotation
```

The class must be renamed to prevent the class name from clashing with the interface name, and the class must implement the interface ❹. The `@WebService` annotation must be modified to match the information in the WSDL file, so we add three elements:

- *The `endpointInterface` element* ❶—Identifies the interface that defines the web service. In the earlier bottom-up example, the class defined the web service; therefore, you didn't need this element in that example.
- *The `portName` element* ❷—Identifies the port name. You get this information from the WSDL file. If you don't provide this information, the port name is assumed to be derived from the class name (`SalesTaxImplPort` in this case).
- *The `wsdlLocation` element* ❸—Identifies the location of the WSDL file. You can specify any location within the web application, although a location within META-INF or WEB-INF is generally preferred.

Note that the `@WebMethod` annotation isn't required on the method because that annotation is already on the method in the interface. We don't show the contents of the methods because they haven't changed from the earlier example. Other than these minor changes the class remains the same.

You also have to make one change to the web.xml file, as shown in listing 9.5. Because the servlet must refer to the class and not the interface, you have to change the class name to reference `SalesTaxImpl`.

Listing 9.5 The web.xml file with top-down changes

```
<web-app...>
  <servlet>
    <servlet-name>SalesTax</servlet-name>
    <servlet-class>org.jbia.ws.SalesTaxImpl</servlet-class>     <  The only change
  </servlet>
  ...
</web-app>
```

Now that you have all the files, compile the interface and class, and package them along with the WSDL and web.xml files in a WAR file, as illustrated in figure 9.4. You can then deploy the WAR file and access the web service.

Now that you have your web service defined using two different approaches, let's turn our attention to writing a client to access the service.

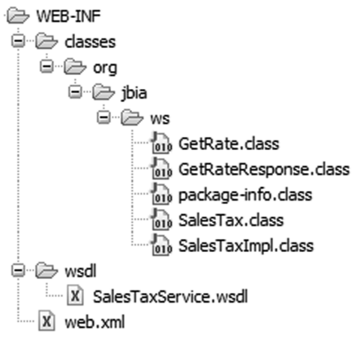


Figure 9.4 The salestax.war file contains more files when you use a top-down approach to construct web services. Compare this list of files to that shown in figure 9.2.

9.2.5 Developing the client

The example client is a simple command-line application that takes a list of state codes on the command line and prints the sales tax rate for each state. First, you generate the stubs for the client from WSDL. Note that this means that the client is coded in a top-down approach. To generate the stubs, make sure that the application server is running, the web service is deployed, and that you can access it from a browser as shown earlier in section 9.2.2. Use the `wsconsume` utility to generate the stub files as follows:

```
wsconsume http://localhost:8080/salestax/tax?wsdl
```

The `wsconsume` utility creates the stub files and compiles them. You'll need to include the generated classes in your class path when you compile the client and the classes in the final JAR file for the client. If you develop the service and the client on the same machine, make sure that the client doesn't have visibility to the files that make up the web service; otherwise, the compiler will get confused. For example, the generated files contain an interface named `org.jbia.ws.SalesTax`, which is the same name as the class that implements the web service if you used a bottom-up approach. If both are available to the compiler or the runtime, the wrong one might be used.

Now that you have the stubs, you can write the client. The code is shown in listing 9.6.

Listing 9.6 The web services client

```
package org.jbia.ws;
public class Client {
 public static void main(String[] args) {
  if (args.length > 0) {
   SalesTaxService svc = new SalesTaxService();       ❶ Creates service
   SalesTax tax = svc.getSalesTaxPort();              ❷ Obtains service endpoint
   for (int i = 0; i < args.length; i++) {
    double rate = tax.getRate(args[i]);               ❸ Invokes service method
    System.out.println("Sales tax for " + args[i] + " is " + rate);
}}}}
```

The first step is to declare the service ❶. Once you have it, you can obtain the service endpoint ❷ and then call the method ❸. As we mentioned earlier, the `SalesTax` item referenced is the interface generated by `wsconsume`, not the class that implements the web service.

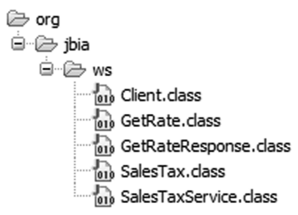
That's all there is to it. Using a web service isn't that much different from using a local library of classes in a JAR file. The secret is that the stubs and the JAX-WS implementation within JBossWS handle all the plumbing code, enabling you to concentrate on the business logic.

PACKAGING AND RUNNING THE CLIENT

Now you're ready to compile and package the client. Remember to include the generated class files in the class path for the compiler and to add them to the JAR file, as shown in figure 9.5.

In the example, you coded the `Client` class. The `wsconsume` utility generated the rest of the class files.

Use the `wsrunclient` script to run the client. This script automatically adds to the class path the JAR files needed to run web service clients.



```
org
  jbia
    ws
      Client.class
      GetRate.class
      GetRateResponse.class
      SalesTax.class
      SalesTaxService.class
```

Figure 9.5 Here are the contents of the client JAR file. Only the Client.class file is hand-coded; the other files are generated by the `wsconsume` utility.

Here's an example of running the client:

```
>wsrunclient -classpath $JBOSS_HOME/client/jbossall-client.jar:./client.jar
➥ org.jbia.ws.Client CA NH TX
Sales tax for CA is 7.75
Sales tax for NH is 0.0
Sales tax for TX is -1.0
```

TIP Did you add logging statements to your client and provide a log4j.properties file, but the expected log file never showed up? Examine the `wsrunclient` script, and you'll see that it sets the `log4j.configuration` system property to wstools-log4j.xml, which you'll not find anywhere. It used to be in the client/jbossws-client.jar file, but now that file no longer appears. If you want to see logging output, remove that reference from the `wsrunclient` script.

Now that you have a Java client for your web service, let's look at writing a C# client.

9.2.6 *Developing a C# client*

The primary motivation behind Web Services is to enable organizations to exchange data among heterogeneous systems. Therefore, we now show how to consume the web service in the .NET Framework using C# and Visual Studio.

In Visual Studio, create a new C# console application project called TaxClient. Once the project is created, add a web reference to the project, as indicated in figure 9.6.

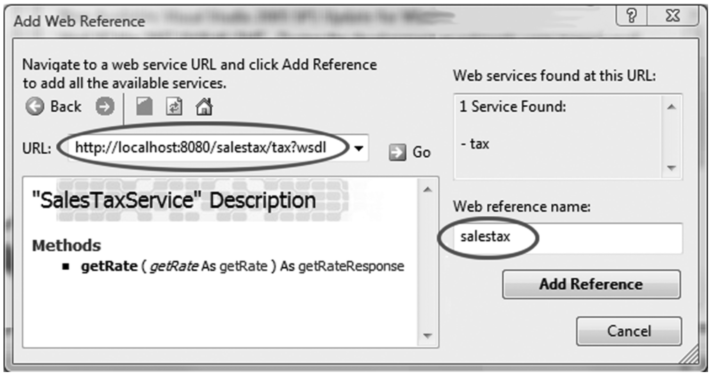

Add Web Reference
Navigate to a web service URL and click Add Reference to add all the available services.
Back
URL: http://localhost:8080/salestax/tax?wsdl
Go
"SalesTaxService" Description
Methods
getRate (getRate As getRate) As getRateResponse
Web services found at this URL:
1 Service Found:
- tax
Web reference name:
salestax
Add Reference
Cancel

Figure 9.6 To add a web service reference to a Visual Studio project, provide the URL for the WSDL file and a name for the Web reference.

Notice that the URL used for the WSDL is that same as that used earlier for the `wsconsume` utility. By default, Visual Studio uses the hostname as the web reference name; we changed it to `salestax`.

The C# client does the same thing as the earlier Java client; it accepts state codes on the command line and prints the sales tax rate for each state. The code is given in listing 9.7.

Listing 9.7 The C# Web Services client

```
using System;
using System.Collections.Generic;
using System.Text;
using TaxClient.salestax;      ❶
namespace org.jbia.ws {
 class Client {
  static void Main(string[] args) {
   if (args.Length == 0) {
    Console.WriteLine                                  Prints usage
     ("usage: TaxClient <list-of-states>");      <-   instructions
   } else {
    SalesTaxService svc = new SalesTaxService();     ❷
    for (int i = 0; i < args.Length; i++) {
     getRate rr = new getRate();                 ❸
     rr.arg0 = args[i];
     getRateResponse resp = svc.getRate(rr);     ❹
     double rate = resp.@return;     ❺
     Console.WriteLine("Sales tax for " + args[i] + " is " + rate);
}}}}}
```

The namespace used for the web service is a combination of the name of the project and the name given to the web reference ❶. The usage instructions are slightly different because the project name is used for the program name (for readers unfamiliar with C#, the end result for a compile is an EXE file). The code then gets the web service ❷. Within the `for` loop that iterates through the command line parameters, the code builds the parameter to pass to the web method ❸, calls the web method passing the parameter ❹, and extracts the returned result ❺ before printing out the result.

9.2.7 *Revisiting the SOAP binding styles*

If you did a double take on the code because it looks a little strange, don't worry. It is strange. There are two SOAP binding styles: document and RPC. This code reflects how a C# client is coded if you're using document style. If you're wondering where the SOAP binding style was declared, recall that the web service container now provides reasonable defaults for any options you don't explicitly declare. Because you never stated which binding style to use, the web service container, when it generated the WSDL using the bottom-up approach, used the logical default—document style.

Document style makes perfect sense for the typical web services usage. For example, if the example service were for use in real-world scenarios, you'd probably code it so that it returned a collection of all sales tax rates, instead of a single rate. This way, the client could ask for the rates once when it came up and then cache the rates for repeated use. In that case, the web service would be returning a complex data type. The best way to deal with a complex data type in a heterogeneous environment is to use the document style to return a complex object and let the client extract the data from the complex type using methods or properties to get that data.

If you're dealing with simple types, such as in the example, then you could change the web service to use the RPC-style SOAP binding. Add a `@SOAPBinding` annotation to the SalesTax web service, as shown in listing 9.8.

Listing 9.8 Specifying a different SOAP binding

```
import javax.jws.soap.SOAPBinding;                    | New lines added
@SOAPBinding(style=SOAPBinding.Style.RPC)             |
@WebService()
public class SalesTax {
```

Then you rebuild and redeploy the web service. If you plan on using the Java client we showed you earlier with this web service, run the `wsconsume` utility again to generate updated stubs; fewer classes will be generated, and fewer classes will be in the client's JAR file. No change is necessary to the client source code. It still works.

For the C# client, ask Visual Studio to reload the WSDL by right-clicking the SalesTax entry under Web References within the Solution Explorer panel and selecting the Update Web Reference option. Then change the `else` clause within the client as shown in listing 9.9.

Listing 9.9 Updated C# client for RPC SOAP binding

```
} else {
 SalesTaxService svc = new SalesTaxService();
 for (int i = 0; i < args.Length; i++) {
  double rate = svc.getRate(args[i]);
  Console.WriteLine("Sales tax for " + args[i] + " is " + rate);
 }
```

Now this looks better and more closely matches the Java coding. You can run the client as follows to verify that you can access the web service properly:

```
>taxclient CA NH TX
Sales tax for CA is 7.75
Sales tax for NH is 0
Sales tax for TX is -1
```

There you have it—a Java POJO-based web service with both Java and C# clients.

9.3 Exploring JBossWS annotations

As you saw in the example, much of the configuration of Web Services can be done through annotations. Although we don't explain the annotations defined by JSR-181 (you can learn about them from the JSR-181 specification), we do want to cover the annotations provided by JBossWS itself. There are two such annotations: `@WebContext` and `@EndpointConfig`. A third annotation used with Web Services is the `@SecurityDomain` annotation, which is EJB-related.

9.3.1 Understanding the WebContext annotation

You use the `org.jboss.wsf.spi.annotation.WebContext` annotation to define items normally declared in the web.xml file. These items are identified in table 9.2.

The default column provides the value used if that element isn't specified, not the default value of the element itself; each element typically defaults to an empty string. For example, if you don't provide a `contextRoot` element, its value will be an empty string, but at the time it's used, the Web Services server will choose to use the archive name to build the context root.

Table 9.2 `WebContext` annotation elements

Element name	Default	Description
`contextRoot`	Name of JAR or EAR file	The context used in the URL to access the web service. This option is ignored if the endpoint isn't an EJB.
`virtualHosts`	-none-	Specifies the virtual hosts to which the web service is to be bound. Virtual hosts are defined in the server/xxx/deployer/jbossweb.sar/server.xml file.
`urlPattern`	Name of the class	The name appended to the context root to form the full URL. This option is ignored if the endpoint isn't an EJB.
`authMethod`	-none-	Identifies if the client needs to be authenticated to use the web service. Valid values are `BASIC` and `CLIENT-CERT`. This option is ignored if the endpoint isn't an EJB.
`transportGuarantee`	`NONE`	Indicates the level at which the transport mechanism will guarantee that the transmitted data hasn't been tampered with. The possible values are ▪ *`NONE`*—The data is passed using plain text (not encoded). There's no guarantee that the data hasn't been tampered with. ▪ *`INTEGRAL`*—The transport mechanism guarantees that the data can't be modified while in transit. ▪ *`CONFIDENTIAL`*—The data is encrypted before being transmitted. This also guarantees that the data can't be modified. Usually, any guarantee other than `NONE` causes the data to be sent using SSL. This option is ignored if the endpoint isn't an EJB.
`secureWSDLAccess`	`True`	If the endpoint is secure (authentication is required to access the endpoint), then this indicates if authentication is also required to access the WSDL. This setting is ignored if the endpoint isn't secure.

You might have noticed that most of the annotation elements come into play only if the endpoint is also an EJB. To show how the `WebContext` annotation is used, we also must show how to convert the earlier POJO into an EJB. Let's do that next.

CONVERTING THE ENDPOINT TO AN EJB

Converting the SalesTax POJO web service into an EJB is fairly simple using annotations. Listing 9.10 highlights the changes necessary.

Listing 9.10 Implementing the endpoint as an EJB

```
import org.jboss.wsf.spi.annotation.WebContext;     ❶
import javax.ejb.Stateless;
@Stateless      ❷
@WebContext(contextRoot="/salestax", urlPattern="/tax")      ❸
@SOAPBinding(style=SOAPBinding.Style.RPC)
```

```
@WebService()
public class SalesTax {...}
```

First, the packages that contain the annotations are imported ❶, then the `@Stateless` annotation ❷ declares the class to be a stateless session bean, and finally the `@WebContext` annotation ❸ provides the context information that was supplied as part of the web application when the web service was a POJO.

Because the endpoint is now an EJB, you package it as an EJB JAR; you no longer need a web.xml file. The complete JAR file contents are given in figure 9.7. If you deploy this JAR file, remember to first undeploy the salestax.war file. Once it's deployed, the Java and C# clients should still work.

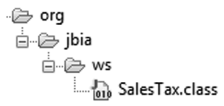


Figure 9.7 Here's the salestax.jar file containing an EJB endpoint. All you need is the class that implements the EJB.

By the way, if you didn't specify a `contextRoot` or `urlPattern` element for the `WebContext` annotation, the URL for the WSDL looks something like http://jbiahost: 8080/salestax/SalesTax?wsdl. The default values for `contextRoot` and `urlPattern` are salestax (the JAR filename) and `SalesTax` (the class name) for this example.

9.3.2 *Understanding the EndpointConfig annotation*

The `org.jboss.ws.annotation.EndpointConfig` annotation is used to identify the configuration to use with the endpoint. Table 9.3 describes the elements that can be set.

Table 9.3 `EndpointConfig` annotation elements

Element name	Default	Description
`configName`	-none-	Identifies the configuration to use.
`configFile`	server/xxx/deploy/jbossws.sar/META-INF/standard-jaxws-endpoint-config.xml	Identifies the file containing the endpoint configurations. This element is ignored if `configName` isn't supplied. If you're using the old JAX-RPC style of web services, a corresponding standard-jaxrpc-endpoint-config.xml configuration file is used instead. The location is relative to the application's location. For example, with the salestax.war file, you could place a handlers.xml file into the WEB-INF directory, in which case the value of `configFile` would be WEB-INF/handlers.xml.

Here's an excerpt from the default JAX-WS configuration file:

```
<jaxws-config ...>
 ...
 <endpoint-config>
  <config-name>Standard WSAddressing Endpoint</config-name>
  <pre-handler-chains>
   <javaee:handler-chain>
```

```
      <javaee:protocol-bindings>##SOAP11_HTTP
      </javaee:protocol-bindings>
      <javaee:handler>
       <javaee:handler-name>WSAddressing Handler</javaee:handler-name>
       <javaee:handler-class>
    org.jboss.ws.extensions.addressing.jaxws.WSAddressingServerHandler
       </javaee:handler-class>
      </javaee:handler>
     </javaee:handler-chain>
    </pre-handler-chains>
   </endpoint-config>
   ...
  </jaxws-config>
```

An endpoint configuration, denoted by the `<endpoint-config>` tag, has a number of attributes, as shown in table 9.4.

Table 9.4 Endpoint configuration attributes

Attribute	Description
`<config-name>`	Identifies the configuration. This name is used in the `configName` element of the `EndpointConfig` annotation.
`<pre-handler-chains>`	Identifies code that will process the message before it's passed to the endpoint. Typical handlers include the following: ▪ *Addressing service handler*—Adds the addressing information to the message context as the value for the `JAXWSAConstants.SERVER_ADDRESSING_PROPERTIES_INBOUND` property ▪ *Security handler*—Handles access control
`<post-handler-chains>`	Identifies code that processes the result after the endpoint has responded to the messages and before the response is returned to the client.
`<feature>`	Identifies particular features to use. You can use this attribute to get the Message Transmission Optimization Mechanism (MTOM) feature, which is used to more efficiently serialize messages containing the MIME types `image/jpeg`, `text/xml`, `application/xml` and `application/octet-stream`. The usage is `<feature>http://org.jboss.ws/mtom</feature>`.
`<property>`	Used to identify name/value pairs of properties.

Let's now turn our attention to securing the web service.

9.4 Securing a web service

Securing a web service includes authorization (and its companion, authentication) and encryption. We look at web service authorization and then venture into encryption.

9.4.1 Authorizing web service access

By default, anyone can call a web service. Although this might be acceptable for a web service accessed only from within a company or for a general-purpose query such as stock quotes, it's probably not the best thing for a web service that, say, obtains someone's medical records.

In this section, we show you how to secure the SalesTax web service. First, we must decide on a security realm, then define some accounts and roles in that realm, and finally use that realm to provide authentication and authorization for the web service.

9.4.2 *Defining the security realm*

An examination of the server/xxx/conf/login-config.xml file shows that a security realm, named `JBossWS`, is used to test security for web services. We use that realm because it's suitable for our purposes. You could easily define a security realm that uses Lightweight Directory Access Protocol (LDAP) or a database to store the authentication information.

The `JBossWS` realm uses the files server/xxx/conf/props/jbossws-users.properties and jbossws-roles.properties to define the accounts and roles. Add a role, `merchant`, and assign an account name and password to each merchant who contracts to use the SalesTax web service. Assuming two merchants have signed up, the jbossws-users.properties would contain the following (although probably with stronger passwords):

```
TJs_Pizza=password1
A1_Auto_Repair=password2
```

And the jbossws-roles.properties file would contain the following:

```
TJs_Pizza=merchant
A1_Auto_Repair=merchant
```

Now that the realm is set up, let's look into securing both the POJO and the EJB Web Services.

SECURING THE POJO WEB SERVICE

Because a POJO web service is packaged in a WAR file and uses the same descriptors. you set access control on the web service the same way as you would for a servlet or JSP. Listing 9.11 highlights the new lines you need to add to the web.xml file.

Listing 9.11 Security-related changes made to the web.xml file

```
<web-app ...>
  ...
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Secure Sales Tax</web-resource-name>     ❶ Context
      <url-pattern>/tax</url-pattern>                                to secure
    <http-method>POST</http-method>          ❷ Secures only
    </web-resource-collection>                  POST requests
    <auth-constraint>
      <role-name>merchant</role-name>         ❸ Authorized role
    </auth-constraint>
  </security-constraint>
  <login-config>
    <auth-method>BASIC</auth-method>          ❹ Uses BASIC
                                                 authentication
    <realm-name>JBossWS</realm-name>
  </login-config>
  <security-role>
    <role-name>merchant</role-name>       ❸ Authorized role
  </security-role>
</web-app>
```

Because the web service uses the /tax context, that's the context that must be secured ❶. This is the same value that would be placed into the `urlPattern` element of the `@WebContext` annotation. The role name, `merchant` ❸, has to match the roles defined in the jbossws-roles.properties file. For the example, we use `BASIC` authentication ❹.

Only POST requests are secured ❷. The client uses POST requests to make the web service calls and a GET request to access the WSDL. The JAX-WS API doesn't provide a mechanism to specify the account name and password when the client obtains the WSDL; securing only POST requests ensures that the client still has access to the WSDL.

You need a jboss-web.xml file to identify the JNDI name for the security realm. Use the existing JBossWS realm, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-web>
  <security-domain>java:/jaas/JBossWS</security-domain>
</jboss-web>
```

Now that you have all the files, you can package them into the WAR file and deploy it. The contents of the WAR file are illustrated in figure 9.8.

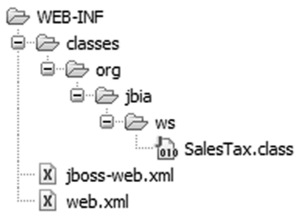


Figure 9.8 Here are the contents of the WAR file for a secured POJO web service. The only additional file, beyond what is listed in figure 9.2, is the jboss-web.xml file.

Now that you have the web service running, you need to modify the client to provide the proper credentials to access the web service. Let's do that next.

MODIFYING THE CLIENT TO ACCESS A SECURE WEB SERVICE

The client needs to supply the username and password when accessing the web service. To keep the changes to the client simple, we hard-code one of the accounts into the client. You need to add several lines right after getting the web services port. Listing 9.12 highlights the new lines in context. (The first and last lines are from the earlier example.)

Listing 9.12 Security-related changes to the client

```
...
SalesTax tax = svc.getSalesTaxPort();
BindingProvider bp = (BindingProvider)tax;          ❶
Map<String, Object> rc = bp.getRequestContext();        ❷
rc.put(BindingProvider.USERNAME_PROPERTY, "TJs_Pizza");     ❸
rc.put(BindingProvider.PASSWORD_PROPERTY, "password1");
for (int i = 0; i < args.length; i++) {
...
```

The object returned by the getXXXPort method is versatile. Besides implementing the web service endpoint, which is SalesTax in this example, that object also implements the javax.xml.ws.BindingProvider interface ❶. This interface owns a Map containing properties used for the request ❷ where you set the username and password ❸.

Now that you have the client updated, compile it and run it as before, using the wsrunclient script. You should once again get the desired sales tax rates. To verify that the authentication is working, you can either scan the server log file looking for entries from org.jboss.security.auth.spi.UsersRolesLoginModule, or you can change the code to provide an invalid username or password—in which case, you should get an HTTP 401 error reported.

Now that the secured POJO version of the web service is running, let's turn our attention to securing the EJB version of the web service.

SECURING THE EJB WEB SERVICE

Use the WebContext annotation to define the security configuration information. Listing 9.13 shows the modified SalesTax EJB web service.

Listing 9.13 Security-related changes to the EJB web service

```
...
@WebService()
@WebContext(contextRoot = "/salestax", urlPattern = "/tax",
        authMethod = "BASIC",        ❶
        secureWSDLAccess = false)      ❷
@SecurityDomain(value = "JBossWS")       ❸
@Stateless
public class SalesTax {...}
```

You only need to change three lines to make the EJB secure. First, the authMethod element for the @WebContext annotation indicates that the BASIC authentication mechanism is used to authenticate the user ❶. This setting corresponds to the <auth-method> tag in the web.xml file for the POJO web service. The secureWSDLAccess element is set to false ❷ so that the client, and others, can access the WSDL without supplying credentials. Finally, the value element of the @SecurityDomain annotation identifies the name of the login module used ❸. This setting corresponds to the <security-domain> tag within the jboss-web.xml file used for the POJO web service, although without the java:/jaas/ prefix. You could also provide the prefix as part of the value element, such as value="java:/jaas/JBossWS", but we recommend that you don't.

Compile the source file and package the class file into salestax.jar as you did earlier. Once you deploy the JAR file (don't forget to undeploy the salestax.war file first if it's still deployed), you should be able to access the WSDL via a browser without having to log in. In addition, you should be able to run the client to access the web service.

9.5 *Encrypting SOAP messages*

For confidential information such as medical records, you'll want to also encrypt the message so that the contents can't be monitored during transport. In this section, we show you how to encrypt the SalesTax web service.

One of the unique aspects of encrypting a web service is that it can be done in two different ways. First, you can use SSL to transport messages using HTTPS. The mechanisms used to set this up are much the same as for using SSL with a web application. You can also use WS-Security; the contents of the message are encrypted by the JAX-WS implementation on both the client and the server. These two methods are illustrated in figure 9.9. In this chapter, we cover WS-Security only, but you can refer to chapter 6 for information on setting up SSL.

The steps to encrypt the messages are to generate the security certificates and to configure the server and client to use those certificates. To make this example complete, we walk you through all the steps to secure the web service, even the steps to generate the certificates.

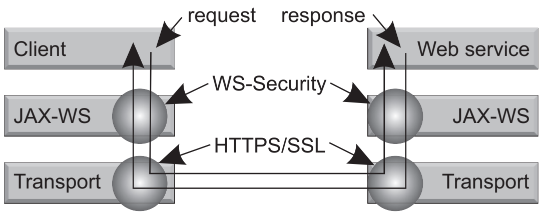


Figure 9.9 Web service requests and responses go though both the JAX-WS and transport layers, so either layer can be used to encrypt and decrypt the requests and responses.

9.5.1 *Generating the certificate*

A web service request and response consists of two messages, each of which has to be encrypted. This is illustrated in figure 9.9. Although you could use the same certificate in both cases, you usually wouldn't want to do so in a production environment because it requires both the server and the client to have the same private key. Usually you want to keep your private key, well, *private*. Therefore, with a single client and a single server you need two certificates so that's what you generate. We discuss how you add more clients after we get the single client example working.

You need two keystores and two truststores. Each keystore contains its own certificate and the public key of the certificate in the other keystore. The truststores contain the public keys of their corresponding certificates. This configuration is illustrated in figure 9.10.

Here are the commands used to set up this configuration:

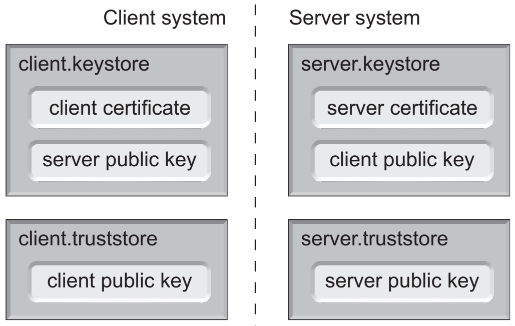


Figure 9.10 Note the relationships among the certificates stored in the keystores and truststores. The sender uses the receiver's public key, which is stored in the keystore, to encrypt the message. The receiver uses its certificate, which contains both its public and private keys, to decrypt the message.

```
keytool -genkey -alias server -keyalg RSA -keystore server.keystore
keytool -genkey -alias client -keyalg RSA -keystore client.keystore
keytool -export -alias server -keystore server.keystore
➥          -file server_pub.key
keytool -export -alias client -keystore client.keystore
```

```
➥          -file client_pub.key
keytool -import -alias client -keystore server.keystore
➥          -file client_pub.key
keytool -import -alias server -keystore client.keystore
➥          -file server_pub.key
keytool -import -alias client -keystore client.truststore
➥          -file client_pub.key
keytool -import -alias server -keystore server.truststore
➥          -file server_pub.key
```

When you're creating the certificates (the first two commands), the keytool command asks for a password for both for the keystore and for the certificate. Remember the passwords you used. You'll need them later.

9.5.2 *Securing the server using WS-Security*

For this example, we use the earlier RPC-style `SalesTax` POJO web service from section 9.2.7. You have to complete two steps: configure the server to use its keystore and truststore and configure the web service to use that configuration.

The jboss-wsse-server.xml file identifies the keystore and the truststore to the server. For a POJO web service, this file is placed into the WEB-INF directory; for an EJB web service, you place it into the META-INF directory. In this file, you also indicate that you want messages to be encrypted. Listing 9.14 shows the contents of the file.

Listing 9.14 Encryption-related security configuration file: jboss-wsse-server.xml

```
<jboss-ws-security
  xmlns="http://www.jboss.com/ws-security/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.jboss.com/ws-security/config
http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">
  <key-store-file>
  ➥WEB-INF/server.keystore</key-store-file>      ❶
  <key-store-type>jks</key-store-type>      ❷
  <key-store-password>password</key-store-password>      ❸
  <trust-store-file>
  ➥WEB-INF/server.truststore</trust-store-file>      ❹
  <trust-store-type>jks</trust-store-type>      ❺
  <trust-store-password>password</trust-store-password>      ❻
  <key-passwords>
    <key-password alias="server" password="serverpwd" />      ❼
  </key-passwords>
  <config>
    <encrypt type="x509v3" alias="client" />      ❽
    <requires>
      <encryption />      ❾
    </requires>
  </config>
</jboss-ws-security>
```

The locations of the keystore ❶ and truststore ❹ files are relative to the base directory of the WAR file. The keystore and truststore use the same password (❸ ❻); you probably want to use stronger passwords. The `<key-store-type>` ❷ and `<trust-store-type>` ❺ default to JKS, so you could leave these tags out. The server key password is provided by the <key-passwords> tag (7) because that password is used to access the server certificate in the keystore. The <encryption/> tag (9) requests that the message be encrypted using the alias provided by the <encrypt> tag (8). The client's public key is used to encrypt the message on the server and is decrypted at the client using the client's private key from the client's keystore. You can also provide <signature/> and <sign> tags to perform authentication.

Add the @EndpointConfig annotation to the SalesTax class to indicate that you want to use WS-Security. Listing 9.15 is an excerpt from the updated SalesTax class, highlighting the added lines.

Listing 9.15 Encryption-related changes to the client

```
...
import org.jboss.ws.annotation.EndpointConfig;        (1)
...
@EndpointConfig(configName="Standard WSSecurity Endpoint")      (2)
public class SalesTax {...}
```

The import statement imports the annotation class (1), and the configName element identifies the configuration you want to use (2). The valid configurations can be found in the file server/xxx/deploy/jbossws.sar/META-INF/standard-jaxws-endpoint-config.xml. Listing 9.16 is an excerpt from that file, showing the Standard WSSecurity Endpoint configuration.

Listing 9.16 Endpoint-handler configuration file: standard-jaxws-endpoint-config.xml

```
<jaxws-config ...>
 ...
<endpoint-config>                                             (1) Configuration
<config-name>Standard WSSecurity Endpoint</config-name>  <--      name
 <post-handler-chains>
  <javaee:handler-chain>
  <javaee:protocol-bindings>##SOAP11_HTTP</javaee:protocol-bindings>
   <javaee:handler>
    <javaee:handler-name>WSSecurity Handler</javaee:handler-name>
    <javaee:handler-class>
➥org.jboss.ws.extensions.security.jaxws.
➥WSSecurityHandlerServer      <--  WS-Security
    </javaee:handler-class>   (2) handler class
   </javaee:handler>
  </javaee:handler-chain>
 </post-handler-chains>
</endpoint-config>
</jaxws-config>
```

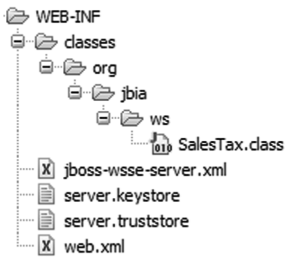
The configuration name given here (1) matches the configuration name used in the EndpointConfig annotation. The WSSecurityHandlerServer class (2) handles the encryption and decryption of the messages.

You can add other handler chains to this configuration and even write your own handler by extending the org.jboss.ws.core.jaxws.handler.GenericSOAPHandler class.

Such a handler has access to and can manipulate the full SOAP message.

Now that you have all the files, you can package them into the salestax.war file, as shown in figure 9.11, and deploy the WAR file. If you have previously deployed the salestax.jar file, remember to undeploy it first.

Note that the standard-jaxws-endpoint-config.xml file isn't included in the WAR file; it's picked up from its default location. If you'd like to place that file into the WAR file, you could provide the location using the `configFile` element on the `@EndpointConfig` annotation. Once the WAR file deploys, you can access the WSDL file through a browser.

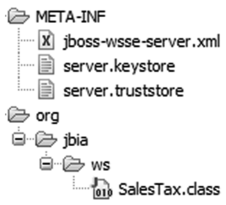


```
WEB-INF
  classes
    org
      jbia
        ws
          SalesTax.class
  jboss-wsse-server.xml
  server.keystore
  server.truststore
  web.xml
```

Figure 9.11 Here are the contents of salestax.war when using WS-Security. The additional files, beyond what you saw in figure 9.2, are the keystore, truststore, and jboss-wsse-server.xml file.

ENCRYPTING AN EJB WEB SERVICE

The steps to encrypting an EJB web service are similar to that of a POJO web service, except that the configuration files and the keystore go into the META-INF directory. You'll also have to change the location of those files in the `<key-store-file>` and `<trust-store-file>` tags in the jboss-wsse-server.xml file. The packaged JAR file is shown in figure 9.12.



```
META-INF
  jboss-wsse-server.xml
  server.keystore
  server.truststore
org
  jbia
    ws
      SalesTax.class
```

Figure 9.12 Here are the contents of salestax.jar, which contains an EJB-based endpoint, when using WS-Security. The keystore, truststore, and jboss-wsse-server.xml file are the additional files, as in the previous figure, but the files are placed into the META-INF directory.

The configuration you have done so far means that the server won't recognize a message unless it's encrypted. You still have to make the changes to get the client to encrypt the message before sending it. Let's look at that next.

9.5.3 *Securing the client using WS-Security*

The client source files don't require any changes to encrypt the message, although be sure to use the earlier client from section 9.2.3 that doesn't perform any login because the server isn't expecting it. The only thing you have to do is configure WS-Security. You use two files to correspond to the two configuration files used for the server.

First, provide the information regarding the keystore and truststore. You can do this by creating a jboss-wsse-client.xml file and placing the necessary information into it, as shown in listing 9.17.

Listing 9.17 Client configuration file: jboss-wsse-client.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss-ws-security
 xmlns="http://www.jboss.com/ws-security/config"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.jboss.com/ws-security/config
    http://www.jboss.com/ws-security/schema/jboss-ws-security_1_0.xsd">
     <key-store-file>
    ➥META-INF/client.keystore</key-store-file>
     <key-store-type>jks</key-store-type>
     <key-store-password>password</key-store-password>
     <trust-store-file>
    ➥META-INF/client.truststore</trust-store-file>
     <trust-store-type>jks</trust-store-type>
     <trust-store-password>password</trust-store-password>
     <key-passwords>
       <key-password alias="server"                  1 Identifies password
                 password="clientpwd" />                for server key
     </key-passwords>
     <config>
       <encrypt type="x509v3" alias="server"/>         Identifies
       <requires>                                      certificate alias
         <encryption/>        Requests message
       </requires>            encryption
     </config>
    </jboss-ws-security>
```

The contents of this file look similar to that used by the server, the only difference being that the keystore and truststore are located in the META-INF directory. The server public key ❶ is used to encrypt the message, which is decrypted at the server using the server's private key.

You can leave out the information about the keystore, truststore, their passwords, and types, and provide that information using the following system properties:

- `org.jboss.ws.wsse.keyStore`
- `org.jboss.ws.wsse.keyStorePassword`
- `org.jboss.ws.wsse.keyStoreType`
- `org.jboss.ws.wsse.trustStore`
- `org.jboss.ws.wsse.trustStorePassword`
- `org.jboss.ws.wsse.trustStoreType`

If you specify this information both in the configuration file and as system properties, the configuration file takes precedence. Additionally, because the same class handles the jboss-wsse-client.xml and jboss-wsse-server.xml files, the system properties could be used for the server also. Because the server might serve multiple Web Services, each with their own WS-Security configuration, it makes sense that the settings in the configuration file take precedence over the system properties.

You have to state that you want to use WS-Security by creating a META-INF/standard-jaxws-client-config.xml file. An example of this file can be found at server/xxx/deploy/jbossws.sar/META-INF/standard-jaxws-client-config.xml. Copy this file to your project and edit it, removing the configurations that you don't want. The only configuration you should leave is `Standard WSSecurity Client`, as shown in listing 9.18.

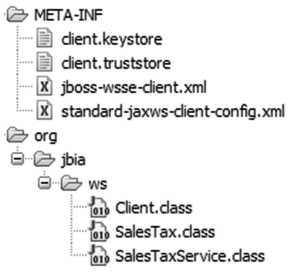
Listing 9.18 Client configuration file: standard-jaxws-client-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jaxws-config xmlns="urn:jboss:jaxws-config:2.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:javaee="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="urn:jboss:jaxws-config:2.0 jaxws-config_2_0.xsd"
>
<client-config>
 <config-name>Standard WSSecurity Client</config-name>        <— Configuration name
 <post-handler-chains>
  <javaee:handler-chain>
  <javaee:protocol-bindings>##SOAP11_HTTP</javaee:protocol-bindings>
   <javaee:handler>
<javaee:handler-name>WSSecurityHandlerOutbound</javaee:handler-name>
    <javaee:handler-class>
 ➥org.jboss.ws.extensions.security.jaxws.         (1) WS-Security handler class
 ➥WSSecurityHandlerClient
    </javaee:handler-class>
   </javaee:handler>
  </javaee:handler-chain>
 </post-handler-chains>
</client-config>
</jaxws-config>
```

The `WSSecurityHandlerClient` (1) is the client-side handler that corresponds to the `WSSecurityHandlerServer` server-side handler. Both of these classes defer to the `WSSecurityHandler` class to handle the messages.

All that's left to do is package the files into a JAR file as illustrated in figure 9.13. The classes are the same as from the earlier example; only the files in META-INF are new.

Once you have the JAR file, you can run the client, once again using `wsrunclient`. It should work. You can verify that the messages are encrypted by turning on message tracing. Uncomment the Enable JBossWS message tracing entry in the jboss-log4j.xml file before starting the application server. Then look for the org.jboss.ws.core.MessageTrace entries in the server.log file.



```
META-INF
  client.keystore
  client.truststore
  jboss-wsse-client.xml
  standard-jaxws-client-config.xml
org
  jbia
    ws
      Client.class
      SalesTax.class
      SalesTaxService.class
```

Figure 9.13 Here are the contents of the client.jar file when using WS-Security. All the classes in the META-INF directory are new.

9.5.4 *Signing the messages using WS-Security*

WS-Security provides a mechanism to sign a message, providing an alternate means of authenticating the user. To illustrate how this works, we modify the example that encrypts messages.

For signing a message, the sender uses his or her private key, and the receiver uses the sender's public key to verify the sender's identity. This means that both the client's public key and the server's public key must be in the server's truststore. This configuration is illustrated in figure 9.14.

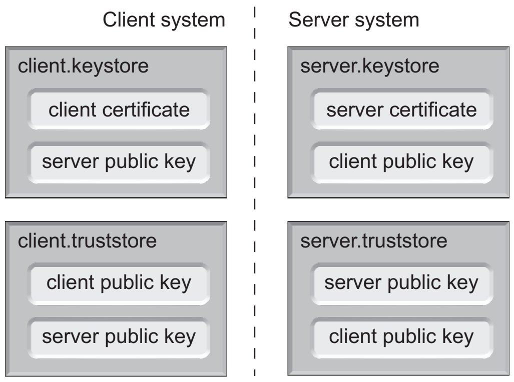


Figure 9.14 Here are the relationships among the keystores and truststores for signing messages. The only difference between this and figure 9.10 is that the other system's public key has been added to the truststore.

Assuming that the keystores and truststores are already set up for encryption, here are the additional commands used to create this configuration:

```
keytool -import -alias server -keystore client.truststore
➥           -file server_pub.key
keytool -import -alias client -keystore server.truststore
➥           -file client_pub.key
```

Once the keys are set up, you must modify the configuration files to use the keys to sign the messages. Listing 9.19 shows an excerpt from the updated jboss-wsse-server.xml file.

Listing 9.19 WS-Security configuration file, jboss-wsse-server.xml, changes

```
<jboss-ws-security ...>
  ...
  <config>
    <sign type="x509v3" alias="server" />      ❶ Identifies certificate used to sign message
    <encrypt type="x509v3" alias="client" />
    <requires>
      <signature />      Requires message to be signed
      <encryption />
    </requires>
  </config>
</jboss-ws-security>
```

The server key is used to sign messages sent by the server ❶. The keystore and truststore-related settings are the same as for the earlier encryption example; only the two lines identified were added.

The changes to the jboss-wsse-client.xml file are similar, as shown in listing 9.20.

Listing 9.20 WS-Security configuration file, jboss-wsse-client.xml, changes

```
<jboss-ws-security ...>
  ...
  <config>
    <sign type="x509v3" alias="client" />      ❶ Identifies certificate used to sign message
    <encrypt type="x509v3" alias="server" />
```

```
      <requires>
        <signature />       <-- Requires message
        <encryption />          to be signed
      </requires>
    </config>
  </jboss-ws-security>
```

In this case the client key is used to sign the messages ❶.

Package up the server and deploy it, package up the client, and then run the client. The messages are now signed. You can verify this by looking at the SOAP messages in the server.log file (after turning on message tracing as indicated at the end of section 9.5.3); you'll see a `<ds:Signature>` entry has been added to the message.

9.6 Summary

This chapter introduced you to Web Services, including its terminology and how that terminology applied to the web service architecture. You learned terms such as endpoints, WSDL, and UDDI. You examined the different SOAP binding styles and should now know the difference between the RPC and document styles.

You built a simple web service, which you then used to examine various configuration topics. You learned how to package and deploy both POJO and EJB-style Web Services. You created Web Services using both top-down and bottom-up approaches. You learned how to use the `wsconsume`, `wsprovide`, and `wsrunclient` utilities. You explored various annotations and configuration files that you can use to configure your web service.

You created both Java and C# clients to access the web service. Working with the C# client led to a more in-depth discussion and understanding of the SOAP binding styles.

You learned how to secure your web service, using both mechanisms to secure web applications and WS-Security. You learned how to use WS-Security to encrypt a message and to sign a message, providing an alternative to standard web application authentication and authorization.

9.7 References

JSR-181, Web Services Metadata for the Java Platform, specification—http://jcp.org/en/jsr/detail?id=181

JSR-224, JAX-WS 2.0, specification—http://jcp.org/en/jsr/detail?id=224

JBossWS User Guide—http://jbws.dyndns.org/mediawiki/index.php?title=JBossWS

JAX-RPC User Guide—http://jbws.dyndns.org/mediawiki/index.php?title=JAX-RPC_User_Guide

JAVA / SOFTWARE DEVELOPMENT

JBoss in Action

Javid Jamae and Peter Johnson

The JBoss 5 Application Server improves performance of Java EE apps. It also boosts developer productivity. It has a cutting-edge layered architecture and modular organization that make it easier to implement best-practice solutions in critical areas like security, transactions, persistence, monitoring, and resource management. JBoss 5 is open source, compliant with the JCP spec, and integrates smoothly with the rest of the Red Hat/JBoss stack, including Seam, Hibernate, JBoss Portal, and Microcontainer.

JBoss in Action is a complete guide to JBoss 5 Application Server, from installation and configuration to production deployment. The book focuses on those things that separate JBoss from other Java EE servers, leading you through component containers, such as the JBoss Web Server, the EJB3 server, and JBoss Messaging. It gives you detailed insight into security, performance, and clustering.

This book is written for Java EE developers, as well as administrators maintaining the JBoss Application Server.

What's Inside

- Full coverage of new JBoss 5.x features
- Focus on what's special in JBoss—not generic Java EE
- No trees wasted: concise and hands-on!

About the Authors

Javid Jamae is a Java programmer, consultant, and trainer. He is also a certified JBoss and Hibernate instructor. **Peter Johnson** is a Java EE architect, a JBoss committer, and a speaker about Java performance at various industry conferences.

For online access to the authors, code samples, and a free ebook for owners of this book, go to www.manning.com/JBOSSinAction

Free ebook
SEE INSERT

"Clear and informative—I highly recommend it."
—Matthew McCullough
Denver Open Source Users Group

"For all professional JBoss users, a must-have book."
—Mario Cartia, Sicily Java Users Group Manager

ISBN-13: 978-1933988023
ISBN-10: 1933988029
54999
9 781933 988023

MANNING US/CAN $49.99 [INCLUDING EBOOK]