

# 50

## Integrating Flex and Java

This chapter explores more deeply into Flex RPC services, focusing on using Flex's `WebService` class to connect to a SOAP-based Web Service written with Java using the Grails framework. The Grails framework is an agile web development platform very similar to Ruby on Rails, except that it runs on the Java VM, which many enterprises already support in their data centers. Grails is based on the Groovy Java language extensions, which bring a lot of dynamic language syntactical sugar to the Java runtime, increasing developer productivity. With Grails, Groovy, and an XFire plug-in, which exposes Java objects at Web Services, you can create Web Services that are easily accessed from Flex applications in no time.

Building on the previous chapter's `HTTPService` sample, we're going to build this chapter's RPC communication with typed Web Services. While developing custom server messaging formats in pure XML is simple, it offers no runtime type-checking. The more complex the messages are that go back and forth, the more useful it becomes to have Web Services defined using WSDL (Web Services Description Language). In addition, this chapter's sample code will advance the Flex client design by breaking all the service layer calls into their own class, separating the view functionality out — a practice that you'll want to employ as the services you call increase in number and complexity. Plus, it's a stepping stone to using a full-blown Flex framework, like Cairngorm, which is discussed in Part X, "Using Cairngorm."

Also, just because we're exposing SOAP-based services and using the `mx.rpc.soap.WebService` class to access the remote data, it doesn't mean that you couldn't implement a RESTful service in much the same way using Grails and Java, or whatever your preferred backend development platform. Any platform these days can offer up data to Flex using any of your preferred RPC libraries, whether that's `HTTPService`, `WebService`, or `RemoteObject`. Chapter 51 looks at Web Services with .NET.

### Introduction to the Routes Web Service

Web Services written to the WSDL 1.1 specification are supported by Flex's `mx.rpc.soap.WebService` library. With a WSDL-based service, methods (operations) and types (defined by XML Schema 1.0) are defined so that clients consuming the service will know the explicit

## Part VIII: Server Integration

---

contract for how to invoke the service, including the appropriate parameters to pass to the methods, and what the methods will return in a type-safe manner. The more complex the operations, their parameters, and the hierarchy of returned data, the more beneficial it is to use WSDL-based Web Services, which provide the foundation of enterprise service-oriented architecture (SOA).

The developer either designs the WSDL interface first (known as *contract first development*) and then fills in the implementation to meet the contract, or they develop Java code first and then, using some tools, have a WSDL generated based on mapping files and domain object hierarchies (known as *code-first*). This chapter's sample takes the code-first approach to exposing Web Services.

When you want to call the remote operation defined in the WSDL, you create an XML message supporting the SOAP format, which is the XML protocol used to exchange the messages defined in the WSDL over HTTP. The Flex `WebService` library is designed to communicate with remote Web Services and provides serialization support for translating objects from ActionScript into XML for transferring across the wire to the Web Service, and then vice versa for the responses coming back from the server.

Chapter 49 introduced you to working with Web Services from within Flex, as well as providing a very basic introduction to WSDL. If you haven't read that chapter, it's suggested you do so now. After you've set up the server-side code for this chapter, you'll be able to view the complete WSDL of the Routes Web Service.

## Setting Up the Server and Development Environment

Let's get started with setting up the developer and server tools you need to run the demo application.

### Server Software Requirements

The server stack for the Routes Web Service is implemented on open-source technologies. We're going to be using MySQL for the database, Jetty for the application server (because that's what Grails uses by default for a development environment, but feel free to use Tomcat, JBoss, WebSphere, or whatever else as determined by your Java infrastructure, especially if you're looking to release into a production environment), the Grails application framework, and the Groovy dynamic language extensions for the Java language.

### Configuring Java, Grails, Groovy, and MySQL

It's assumed that you have experience working with Java, as this isn't intended as an introduction to the many concepts we'll be glancing at in passing, and that you understand these basic setup directions, since many Java environments follow similar conventions.

1. Download Grails from <http://grails.org>. As of this writing, 1.03 was used for the sample applications. Make sure that you have version 1.4 or newer of the Java SDK, and that `JAVA_HOME` has been set in your environment.
2. Extract Grails into a local directory, henceforth referred to as `grails_home`, and set up an environment variable called `GRAILS_HOME` that points to your `grails_home` directory. On our

Windows machine, we installed Grails at `c:/grails` and opened up the Windows System Control Panel ⇨ System ⇨ Advanced ⇨ Environment Variables ⇨ System Variables to set the `GRAILS_HOME` variable.

3. Modify your `PATH` so that `grails_home/bin` is set in it. This is so all the Grails scripts can be easily found for execution. You can set the `PATH` in Windows by going to Control Panel ⇨ Advanced ⇨ Environmental Variables ⇨ System Variables.
4. Verify that Grails is properly installed by opening a command window and typing **grails**.

The following should be displayed if you've set up Grails properly:

```
Welcome to Grails 1.0.3 - http://grails.org/  
Licensed under Apache Standard License 2.0  
Grails home is set to: c:\grails
```

```
No script name specified. Use 'grails help' for more info or 'grails interactive  
' to enter interactive mode
```

5. Next, get the source code for this chapter's `GrailsServer` project down to your machine (it's found in the Subversion repository). This will become the project folder, henceforth referred to as `project_home`.
6. Next, install the XFire Grails plug-in, which will give our Grails application WSDL Web Service support. Open a command prompt in `project_home` directory. This is very important, as Grails requires you to be in a Grails application directory in order to install the plug-in. Then at the command prompt, type **grails install-plugin xfire**.

The XFire plug-in and all its dependencies (a long list of JAR files) will be installed into the demo application. (Newer versions of the XFire plugin might require upgrading the Grails framework.)

7. Install the appropriate JDBC driver for MySQL to the `project_home/lib` directory. The `mysql-connector-java-5.1.7-bin.jar` can be found at <http://dev.mysql.com/downloads/connector/j/5.1.html>.
8. Start MySQL and create a new database called `fittracker`. (As mentioned, this example uses MySQL by default). When the Grails server application starts, it will build the necessary database table to persist routes, but the `fittracker` database needs to exist first. Additionally, after you've started the Grails application for the first time, you can run the `data.sql` script to populate the database with some default data.

You can support alternative database configurations, however. If you've installed the `fittracker` database to a different server or with different default login or even a different database system (like Oracle, SQL Server, etc.), you can modify the `project_home/grails-app/conf/DataSource.groovy` file to match your own database configuration. Those of you familiar with JDBC will be able to figure out the different settings. By default, the `DataSource.groovy` file that was downloaded is already configured to connect to the database `fittracker` with the default root user. Modify the JDBC driver section and database login to meet your specific deployment settings.

That should be it. Grails automatically installs Groovy, so your environment should be all set there. The XFire plug-in should be setup, as well.

### Running the Server

To run the server environment, make sure that your MySQL service has been started. Then, at a command prompt, navigate to the `project_home` directory where you installed the code of the `fittracker` application and type **grails run-app**.

The Grails application should start and eventually in the output window, if all goes as planned, you should see something similar to the following:

```
2008-11-28 20:22:05.536::INFO: Started SelectChannelConnector@0.0.0.0:8080
Server running. Browse to http://localhost:8080/fittracker_demo
```

Now, in a web browser, to see the generated WSDL from the application, navigate to `http://localhost:8080/fittracker_demo/services/route?wsdl`. If you can see WSDL, you're ready to continue.

Advanced Eclipse users (see the next section, "Client Software You Might Need") can run the Grails project (and enable debugging) by importing the project into an existing Eclipse workspace. However, programming the server code is beyond the scope of this chapter.

### Client Software You Might Need

When working with Web Services, you may want some additional tools:

- ❑ **Eclipse with the Java development tools** ([www.eclipse.org/jdt](http://www.eclipse.org/jdt)) — If you're running stand-alone Flex Builder or Flex Builder as a plug-in, you are already running Eclipse. However, to run and edit server-side Java code, you'll need to install the Java development tools, as well.
- ❑ **soapUI** ([www.soapui.org](http://www.soapui.org)) — This tool helps you test and consume the Web Services you build. It's quite handy for making sure that your Web Services work first outside the Flex platform, before trying to call them from inside the Flex platform. This is available as either a stand-alone program or an Eclipse plug-in.
- ❑ **Web Services tools** ([www.eclipse.org/webtools/ws](http://www.eclipse.org/webtools/ws)) — This is an Eclipse plug-in with support for creating and designing your WSDL contracts for contract-first development.
- ❑ **Groovy Plugin for Eclipse** (<http://groovy.codehaus.org/Eclipse+Plugin>) — This is an Eclipse plug-in that provides syntax highlighting, code completions, refactoring, and source code formatting for the Groovy language extensions.

## The Grails Routes Web Service

If you've been hearing about Ruby on Rails and are a bit jealous of how simple it is to get up and running to develop your applications, but you're limited to supporting an existing Java infrastructure, Grails and Groovy are worth looking at. Grails is an MVC web development framework that emphasizes convention over configuration. By following certain conventions, you can provide automatic domain model classes to access data in relational database tables, or provide implied MVC coding conventions to your application.

Grails works with existing Java technologies like Spring and Hibernate, so if you're an existing Java developer experienced in these technologies, Grails provides wrapper APIs to quickly get you coding

applications and enables you to configure your custom Java EE stack for the more complex needs of your enterprise.

Our sample `fittracker` Web Services application doesn't use any of the specific Grails MVC infrastructure necessary for typical web applications; however, it does take advantage of Grails Object Relational Mapping (GORM), which makes reading and writing domain objects to a database a breeze, as you'll see below in Listing 50-1. Plus, as you saw in step 6 above, it made installing XFire Web Service support a one-line, 10-second operation. Also, Grails comes with Groovy, so that's configured for you as well.

## Grails Code Supporting the Web Service

Our sample application is going to be performing the same CRUD-based operations on route information as in the previous chapter: `createRoute`, `updateRoute`, `retrieveRoute`, `retrieveRoutes`, and `deleteRoute`. These operations will be available to any application that supports calling Web Services, including our sample Flex application, discussed later in this chapter.

With Grails, you start with a simple domain object written in Groovy and then define a service class and let Grails and XFire work their magic.

### The `Route.groovy` Domain Class

The domain object for the route can be found in `Route.groovy`, and it doesn't look like your everyday Java class:

```
class Route    String name    String description    Float distance    Integer created_by
Date created    ate modified    static optionals = [ "created", "modified" ] }
```

The `Route` class defines some basic properties, including `name`, `description`, `distance`, and so forth, as well as an `optionals` static property, which is used when the `Route` is persisted to the database, informing GORM of the properties that don't require values. The `Route` class is also used to facilitate WSDL generation, where XFire automatically translates (with the help of AEGIS) the POGO (Plain Old Grails Object) into an appropriate WSDL definition that describes how the object will fly across the wire.

The following is the WSDL that's generated to describe the `Route` and that will be referenced in many of the different Web Service operations:

```
<xsd:complexType name="Route">
  <xsd:sequence>
    <xsd:element minOccurs="0" name="created" type="xsd:dateTime"/>
    <xsd:element minOccurs="0" name="created_by" nillable="true" type="xsd:int"/>
    <xsd:element minOccurs="0" name="description" nillable="true" type="xsd:string"/>
    <xsd:element minOccurs="0" name="distance" nillable="true" type="xsd:float"/>
    <xsd:element minOccurs="0" name="id" nillable="true" type="xsd:long"/>
    <xsd:element minOccurs="0" name="modified" type="xsd:dateTime"/>
    <xsd:element minOccurs="0" name="name" nillable="true" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

This basic schema definition will eventually be used by Flex's low-level `SOAPDecoder` class to translate the XML into ActionScript objects based on these properties (we'll talk about this later.)

### **RouteService.Groovy**

The actual service methods are implemented in `RouteService.groovy`. There's one method for each SOAP operation that is being supported by the service. Using Java annotations (more specifically, annotations defined in JSR 181), you can configure a plain class to operate as a Web Service.

To define the `RouteService`, annotate a class with the `@WebService` annotations, passing in both the name and namespace of the Web Service:

```
@WebService(name = "RouteService", targetNamespace =
    "http://www.acme.com/2008/11/RouteService")
class RouteService {
    static expose=['xfire']
}
```

Next, in the `RouteService`, you need to start defining the actual operation that the service will expose. Listing 50-1 shows the `createRoute` function, which you will eventually expose as a SOAP operation to your Flex client.

---

#### **Listing 50-1: The `createRoute` function Web Service Implementation**

```
def Route createRoute(@WebParam(name="route")Route route) throws
    BusinessException {
    if (route.id != null && Route.exists(route.id)){
        throwBusinessException "It appears this route is already exists."
    }

    def r = new Route()
    r.name = route.name
    r.description = route.description
    r.distance = route.distance
    r.created_by = route.created_by
    r.created = new Date()
    r.modified = new Date()
    r.save()

    if (r.id == null){
        throwBusinessException("Route $route.name wasn't saved, try again.")
    }
    return r;
}
```

As you can see, this method takes in a parameter named `route`, which is an instance of the `Route.groovy` domain object defined previously. This `route` was passed into the `RouteService` as XML off the wire, and decoded into a Java object. Using some basic GORM (the `r.save` method), a new object is created and saved into the database.

There's more code for the rest of the `RouteService` implementation code that's included with the sample code for the chapter in the `/GrailsServer` directory.

## Web Service Fault Handling

In the previous chapter, you used custom XML to define server-side business error conditions that you passed back to the client. With SOAP-based Web Services, there's already the concept of SOAP faults that both Flex and WSDL are aware of. SOAP faults are optional elements included with SOAP messages that can communicate to clients when something has gone wrong. And naturally our server-side platform supports sending errors back to the Flex client.

In listing 50-1, the method `createRoute` is defined as potentially throwing a `BusinessException`. And exceptions are raised if either a route is passed in that already has been persisted to the database, in which case the `createRoute` function should have been called, or when the new route failed to persist to the database.

When a `BusinessException` is raised from the server, it will be translated into a SOAP fault, passed across the wire using standard SOAP XML infrastructure, and decoded in the Flex client application as an `mx.rpc.SoapFault`. A custom fault in WSDL is defined as follows:

```
<wsdl:fault name="BusinessException" message="tns:BusinessException" />
```

## Enabling Flex to Receive SOAP Faults

By default, Flex Web Services won't be able to receive SOAP faults because of existing limitations with current web browser technology and the Flash Player plug-in. The response's details with an HTTP status other than 200 aren't passed into the Flash Payer, and consequently the Flex applications. Traditionally, SOAP faults are sent from the server with an HTTP status of 500.

*When connecting a Flex application to a remote Web Service, you might see some mysterious faults being raised in the fault handler of the Web Service. Upon inspecting the fault details of the `faultEvent`, if you're seeing values similar to the following, chances are that Flex isn't properly receiving the SOAP fault from the Web browser. In this case, you should modify the server to return Web Service responses with a status code of 200.*

```
fault.code = 'Server.Error.Request'
faultDetail = 'Error: [IOErrorEvent type = 'ioError...truncated for brevity
fault.rootCause = "Error #2032: Stream Error..."type="warning"
```

To enable Flex applications to properly receive SOAP faults, do one of the following:

- ❑ Set `useProxy` to `true` on the `WebService` class, which will use an intermediary server that will translate the HTTP status of 500 into an HTTP status code of 200. Adobe's BlazeDS supports proxying Web Service requests.
- ❑ Change the SOAP responses on the server to return with an HTTP status of 200. This is what we do with the chapter sample application by creating a Servlet filter. See the enclosed `ExceptionHandler.groovy` and `ExceptionHttpServletResponseWrapper.groovy` for implementation details. Note that we've modified the `web.xml` deployment descriptor to assign the `ExceptionHandler` to the `XFireServlet`, which is responsible for the Web Service calls.

For additional details about the Java implementation of the `RouteService` Web Service implementation, browse through the sample code in the `GrailsServer` subdirectory for this chapter's sample code.

## The Flex Sample Application

This chapter's sample code includes a Flex project that runs a sample application used to manage routes via the Routes Web Service that was created earlier. It's the same application as the previous chapter on `HTTPService`, except that the architecture has changed slightly on the client, separating out the service calls into their own file to help the application design. Also, we're using `mx.rpc.soap.WebService` instead of the `HTTPService`. The sample application performs all five of the operations that the Routes Web Service supports: getting a list of routes, and all CRUD operations for a given route.

The Flex application consists of three files:

- ❑ `Chapter_50_Web_Services_with_Java.mxml` — The main MXML file containing the main user interface of the application called
- ❑ `RouteService.as` — An ActionScript file that manages all the conversation with the remote Routes Web Service implemented in Grails earlier
- ❑ `RouteVO.as` — The domain object used to communicate back and forth with the Web Service by taking advantage of Flex's SOAP-specific serialization between ActionScript and SOAP-based Web Services

### The `RouteService.as` Service Class

As opposed to defining all the code in one MXML file, as was done in the previous chapter, here the `RouteService.as` class was created to manage all aspects of communicating with the remote server. In the future, if you want to change the Web Services that you communicate with, or even switch the protocol of communication, you need to modify only the `RouteService` class. Also, in Part X you'll learn about how Cairngorm manages the separation of services from the view code in a Flex application, so consider this a steppingstone to full-blown Cairngorm application.

As described in the following sections, the `RouteService` class consolidates all the remote service access using a lot of specific APIs, including `SchemaTypeRegistry`, `IResponder`, and `AsyncToken`.

### Using `IResponder`

The `mx.rpc.IResponder` interface is used as a contract for communicating success or failure on asynchronous calls. Usually when you create a `WebService` component, you set the `fault` and `result` event handlers to your custom functions. However, you can implement the `IResponder` interface in your MXML component and then pass a reference to the interface around different layers of the Flex application and have the appropriate methods called when your service calls complete.

To do this, you first must implement the `IResponder` interface in MXML. Do this in the main `Application` tag, as follows:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
paddingBottom="0" paddingLeft="0" paddingRight="0" paddingTop="0"
pageTitle="WebService Route Demo"
implements="mx.rpc.IResponder">
```

Next, implement the interface by creating `fault` and `result` methods as defined by the `IResponder` interface. After adding the `implements` directive to the `Application` class, you must implement the `fault` and `result` methods of the interface; otherwise, the compiler will complain.

The following `result` implementation handles all the successful calls to the Web Service:

```
public function result(data:Object):void
{
    var event:ResultEvent = data as ResultEvent;

    // Turn off custom wait, if it was on.
    customWait = false;

    switch(event.token.typeOfRequest)
    {
        case RouteService.DELETE_ROUTE_REQUEST:
            // event.result is of type Route
            retrieveRoutes();
            Alert.show("Route Deleted: " + event.result.id)
            break;
        case RouteService.RETRIEVE_ROUTES_REQUEST:
            routeData = event.result as ArrayCollection;
            break;
        case RouteService.UPDATE_ROUTE_REQUEST:
            retrieveRoutes();
            Alert.show("Route Updated: " + event.result.name)
            clearForm();
            break;
        case RouteService.CREATE_ROUTE_REQUEST:
            retrieveRoutes();
            Alert.show("Route Created: " + event.result.name)
            clearForm();
            break;
        case RouteService.RETRIEVE_ROUTE_REQUEST:
            var route:RouteVO = event.result as RouteVO;
            Alert.show("Retrieved route: " + route.name);
            currentRouteId = route.id;
            txtName.text = route.name;
            txtDescription.text = route.description;
            txtDistance.text = route.distance.toString();
            break;
    }
}
```

The following `fault` handler handles all the failures of the service layer, including the specialized SOAP faults thrown from the Web Service, as well as standard HTTP communication errors, such as those produced when the remote service can't be reached:

```
public function fault(info:Object):void
{
    var event:FaultEvent = info as FaultEvent;

    customWait = false;

    // event.fault can be a SOAPFault
```

## Part VIII: Server Integration

---

```
if (event.fault.faultCode == "Server.Error.Request") {
    Alert.show("Error connecting to server, try again later.");
} else if (event.fault.faultString == "BusinessException"){
    var bf:XML = new XML(event.fault.faultDetail);
    Alert.show(bf.text(), "Oops, an error occurred...");
} else {
    Alert.show(event.fault.faultString, "Unplanned Service Error..." );
}
}
```

The client will make a request to the service layer (`RouteService.as`), passing in a reference to its implemented `IResponder` interface, which in this case is the main MXML file. The service layer will then use the reference to communicate success or fail back to the UI layer.

For example, the call to get a route looks like this:

```
private function retrieveRoute(routeId:Number):void
{
    startWaiting(StringUtil.substitute("Retrieving Route {0}...", routeId) );
    var rs:RouteService = new RouteService(this);
    rs.retrieveRoute(routeId);
}
```

The `startWaiting` method displays a custom wait message while the remote service is being called. The important thing to notice here is how this is being passed into the constructor of `RouteService`. Since `RouteService` is expecting the interface `IResponder`, it's casting the entire MXML component so that it'll have access to only the `IResponder` implementation, which it'll use when making the actual service call.

The service call to retrieve a route is implemented in the `RouteService.as` class, as follows:

```
public function retrieveRoute(routeId:Number):void{
    var ws:WebService = getWebService();
    var token:AsyncToken = ws.retrieveRoute(routeId);
    token.addResponder(_responder);
    token.typeOfRequest = RouteService.GET_ROUTE_REQUEST;
}
```

First, an instance of `WebService` is created. There's a helper function called `getWebService` that returns a cached instance of a `WebService` or a created one if this is the first time the `WebService` is being called. Next, you call the Web Service operation, `getRoute` (which was defined on the Java service layer and communicated to the client through WSDL).

Optionally, when calling `WebService` operations, a handle to an `AsyncToken` is returned. An `AsyncToken` is created and associated with individual calls to the remote server, and can be used to maintain state information about a particular Web Service call. `AsyncToken` also has a very handy `addResponder` method that takes an `IResponder` as a parameter. And guess what? That reference to `IResponder` that you passed into the constructor of the `RouteService` object is now passed into the `AsyncToken` where, upon completion of the service call, the appropriate `fault` or `result` method of the `IResponder` will be called, which in this case is implemented on the main view of the application.

Finally, to the token, you set the `typeOfRequest` to a static variable on the `RouteService`. This way, when the result handler is called, the `typeOfRequest` can be inspected and the appropriate action in the view of the application can take place, whether it's populating a `DataGrid` with a list of routes or displaying an `Alert` box when a route has been deleted.

This completes how the `IResponder` interface facilitates separation and communication between the layers of a Flex application.

## Working with Custom Server-Side Errors

Unlike working with custom `HTTPService`, where it's up to the server-side developer to define how errors in business logic are returned to the client (usually via custom XML), `WebService` depends on SOAP faults. On the server Grails application, a Java-based, custom `BusinessException` fault will be raised and propagated down to the client as a Flex `SOAPFault`.

Whenever a Groovy/Java exception is raised from the server, using a regular Java exception syntax:

```
throw new BusinessException("BusinessException", "This is an error!")
```

it will be translated into a Web Service fault response:

```
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>BusinessException</faultstring>
      <detail>
        <BusinessException>This is an error.</BusinessException>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

which will then call into the fault handler of the Flex application and be handled according to the appropriate condition:

```
// Truncated code from the fault handler
else if (event.fault.faultString == "BusinessException"){
    var faultDeail:XML = new XML(event.fault.faultDetail);
    Alert.show(faultDeail.text(), "Oops, an error occurred...");
```

The fault handler of the main application tests to see if a custom `faultString` returned from the server is equal to a `BusinessException`. When it is, extract the specific error message from the `faultDetail` using e4X XML processing, and then display the error with an `Alert` box to the user of the application.

## Serialization with SchemaTypeManager

When working with SOAP-based Web Services, Flex supports automatically deserializing XML off the wire into actual `ActionScript` objects. Objects can be translated into either generic objects or specific

## Part VIII: Server Integration

ActionScript objects that you define. This example uses the `Route.VO` object for both inbound and outbound communication with the Web Service.

When calling an operation on a remote service, pass in an instantiated `RouteVO` object, just as the `createRoute` function does:

```
public function createRoute(route:RouteVO):void
{
    var ws:WebService = getWebService();
    Operation(ws.createRoute).encoder.strictNullability = true;
    var token:AsyncToken = ws.createRoute(route);
    token.addResponder(_responder);
    token.typeOfRequest = RouteService.CREATE_ROUTE_REQUEST;
}
```

A route that was created in the UI and set to values of `TextInput` controls is passed into `createRoute`. The `WebService` is created, and the `SoapEncoder`'s `strictNullability` is set to `true` (more about this in a minute), and then the call to the Web Service operation `createRoute` is made, passing in the ActionScript `RouteVO`.

The `RouteVO` is translated into the XML schema type, `Route`, which is defined in the Routes Web Service WSDL. Flex manages this with the help of the `SchemaTypeRegistry`, which maps ActionScript object to definitions in the WSDL. For example, in the `RouteService.as` file, this mapping is done every time a `RouteService` is created for an operation to be called:

```
SchemaTypeRegistry.getInstance().registerClass(new QName("http://DefaultNamespace",
"Route"), RouteVO);
```

The `SchemaTypeRegistry` is a singleton that you configure for the application, and it is used by Flex's `SoapEncoder` and `SoapDecoder` when calling operations and when receiving responses from the server. In this case, you have to map the schema type (`Route`) to the ActionScript type, `RouteVO`. Pay special attention to the `QName`; since the Routes Web Service WSDL defines a `targetNamespace` of `http://DefaultNamespace`, you must respect that here; otherwise, the `SchemaTypeRegistry` will neither find the definition of `Route`, nor be able to translate it. (This target namespace — `http://DefaultNamespace` — is a poor choice. Never do this in a production application. All the WSDL operations, however, are declared with an appropriate namespace.)

When results are returned from the Web Service and processed in the `result` handler, you can access them as ActionScript objects, for example, when a route is retrieved from the server from a call to `getRoute`:

```
var route:RouteVO = event.result as RouteVO;
```

Now, if you didn't tell the `SchemaTypeRegistry` about the mapping between `RouteVO` and the XML schema `Route`, don't worry, you'd still get results. The route would just be returned from the server and deserialized into Flex as a generic `ObjectProxy`, making it available to your ActionScript code in a manner similar to that of a regular ActionScript object:

```
var route:RouteVO = event.result as ObjectProxy;
```

## Mapping ActionScript types to XML Schema types

`SoapEncoder` and `SoapDecoder` facilitate conversion between ActionScript and XML schema types. Although you might not need to know how things are mapped for basic services, the more complex the service, the more familiar you might want to be with how ActionScript objects are converted to and from SOAP messages. The Adobe documentation outlines all the supported mappings between ActionScript and WSDL, but the following table lists a few of the more commonly used mappings.

XML Schema Type	Decoding XML to ActionScript	Encoding Actionscript to XML
xsd:anyType or xsd:anySimpleType	String/Boolean/Number	Object
xsd:Boolean	Boolean	Boolean/Number/Object
xsd:byte		Number/String
xsd:date or xsd:dateTime	Date	Date/Number/String
xsd:decimal, xsd:double, xsd:float, xsd:int, xsd:integer, xsd:long, xsd:unsignedLong, xsd:short	Number	Number/String
xsd:string	String	Object

Notice that when encoding ActionScript into XML, most values can be converted into Strings. However, the inverse isn't necessarily true when decoding from XML into ActionScript, as is the case with most numeric values.

*Although WSDL 1.1 is supported, not every attribute is — for example, `simpleType::restrictions`. Be sure to read the Adobe Flex 3 documentation on Web Services, which you can find by searching for “Web Services” at: <http://livedocs.adobe.com/flex/3/html/index.html>. There’s a comprehensive list of what isn’t supported, as well as a complete table of ActionScript to WSDL serialization mapping.*

### Strict Nillability

In the preceding Web Service operation, the `strictNillability` property is set to `true`. With this property set, the `SOAPEncoder` will not include XML elements for properties that aren't set on the `RouteVO`. For example, in the WSDL both `created` and `modified` aren't defined as nillable, meaning that they are expected in the SOAP request:

```
<xsd:element minOccurs="0" name="created" type="xsd:dateTime"/>
<xsd:element minOccurs="0" name="created_by" nillable="true" type="xsd:int"/>
<xsd:element minOccurs="0" name="modified" type="xsd:dateTime"/>
... more properties are defined
```

Because the `RouteVO` is defined as having these properties (mostly for the sake when reading values passed back from the Web Service), these properties will be added when the `SOAPEncoder` persists the `RouteVO` into the SOAP format for transport over the wire. The trouble is that the values in this particular

## Part VIII: Server Integration

instance of `RouteVO` are null, because the server-side implementation is going to create a modified value, and the created value isn't being passed back up because it's not changed. Since the `SOAPEncoder` is using the WSDL definition received from the server as a template on how it should encode the object, it's going to complain when trying to encode the `RouteVO` into XML for passing up passing null values for required fields.

To get around this, set the `strictNillability` property to true for the operation:

```
Operation(ws.createRoute).encoder.strictNillability = true;
```

Now when encoding the `RouteVO`, the null properties will be left off the SOAP request (no modified or created elements will be created based on the `RouteVO`'s values), and since the WSDL defines the `minOccurs` as 0, it's perfectly valid. Here's what the Web Service requests for `updateRoute` looks like when passing over the network:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <tns:updateRoute xmlns:tns="http://www.acme.com/2008/11/RouteService">
      <tns:route>
        <tns:created_by>2</tns:created_by>
        <tns:description>44444</tns:description>
        <tns:distance>25</tns:distance>
        <tns:id>2</tns:id>
        <tns:name>Updated Route Name</tns:name>
      </tns:route>
    </tns:updateRoute>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

With the `encoder.strictNillability` property, you can control how your objects are encoded and passed up to the Web Service.

### Calling Web Services from Other Servers

The Flash Player has built-in security protection that will keep you from directly calling Web Services from domains other than the domain that served up the Flex application. To get around this, either you create a `crossdomain.xml` file and place it in the root of the remote server or you use a proxy server, as explained previously when working with HTTP status codes.

The easiest thing, although it's not very controlled, is to create a `crossdomain.xml` file at the root of the domain that allows total access to any remote Flash/Flex application:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM "http://www.macromedia.com/xml/dtds/
  cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="*" />
</cross-domain-policy>
```

However, if you want a little more control, you can open only Web Services to be delivered to specific domains by limiting which HTTP headers have access through a `crossdomain.xml` file:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM "http://www.macromedia.com/xml/dtds/
  cross-domain-policy.dtd">
<cross-domain-policy>
  <allow-access-from domain="*" />
  <allow-http-request-headers-from domain="*" headers="SOAPAction"/>
</cross-domain-policy>
```

Notice that the preceding filters allowed requests based on headers containing `SOAPAction`. Both of the preceding sample `crossdomain.xml` files are allowing any domain access, but you can just as easily limit which domains delivering your Flex applications are allowed access by setting the `allow-access-from` element's `domain` attribute:

```
<allow-access-from domain="www.alloweddomain.com" />
```

Or, if you want to allow any subdomain for a domain, such as `www.alloweddomain.com` and `blog.alloweddomain.com`, then use a wildcard:

```
<allow-access-from domain="*.alloweddomain.com" />
```

## Summary

This chapter built on the basics of RPC communication, working with SOAP-based Web Services delivered via the Java platform. Be sure to explore the sample code, as there's a lot of code to help facilitate creating your own CRUD-based Web Service consumption for your own Flex applications.

The key to working with Web Services in Flex is to develop and test the server-side independently with tools like `soapUI`, and then integrate the Web Service into your Flex application.

Feel free to mix and match your favorite RPC APIs with the ways you develop your server applications. The sample code from this chapter isn't limited to being used only as a `WebService`. A simple conversion would make it work with an `HTTPService` or `RemoteObject`, too.

Coming up in Chapter 51, "Web Services with .NET and Flex," we'll create a similar Fit Tracker application that connects to a backend written in .NET, using Web Services for its RPC strategy.

