

CHAPTER 20

Flex Integration with J2EE

IN THIS CHAPTER

- Java and Flex Concepts
- J2EE Technical Architecture
- Flex Technical Architecture
- iteration::two Online Banking Application
- Stateful J2EE Integration

RIA Development with Flex and J2EE

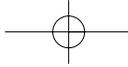
In the previous chapter, we described how Macromedia Flex integrates with business logic exposed as a set of web services in a service-oriented architecture (SOA) environment.

In collaboration with the Flex Web Service proxy, a Rich Internet Application (RIA) delivered with Flex can invoke web services located on a remote machine in a loosely coupled fashion, or it can be used in a more tightly coupled way to bind a rich client interface to middleware exposed through a web service application programming interface (API). For middleware residing in the .NET architecture, this offers a simple way of performing .NET integration, as the following example class shows:

```
using System;
using System.Web.Services;

[WebService(Namespace="http://banking.iterationtwo.com/",Description="Fetch Account
Details for Customers")]
public class AccountDelegate
{
    [WebMethod(Description="Get Accounts for Given Customer Number")]
    public Account[] getAccounts ( String customerID )
    {
        // business logic goes here...
        return accounts;
    }
}
```

In the preceding C# class, the `WebMethod` metadata on the `getAccounts()` method ensures that .NET will expose the method on the `AccountDelegate` class, which itself has been exposed as a `WebService` using the `WebService` metadata.



2 | Chapter 20 FLEX INTEGRATION WITH J2EE

Because Flex 1.0 is a J2EE application residing within a J2EE application server, Flex can couple even more tightly to J2EE middleware. Similar to .NET, we can expose J2EE middleware as web services and use the techniques discussed in the previous chapter; however, Flex also provides the `<mx:RemoteObject>` tag, which offers us a means of directly invoking J2EE business logic via the Flex server-side gateway.

Direct Invocation on .NET Infrastructure

Macromedia announced that the roadmap for Flex will include a native .NET port of Flex. We are assured that the aim of this native port is to ensure a “write once, deploy anywhere” approach with MXML. Consult the documentation for the .NET port of Flex when it’s released, for details of how to perform tight Flex and .NET integration. We’ll endeavor to update the accompanying Web site for the book with additional notes on new product features in future versions of Flex.

In this chapter, we explore the `<mx:RemoteObject>` tag and gain an understanding of how to integrate an RIA front-end developed in Flex with new or existing middleware deployed within a J2EE application server.

Because many of the concepts of Remote object invocation are similar to the web service integration we discussed in the previous chapter, we’ll use this opportunity to consider best practices and candidate architectures for both our client- and server-side development. Although we will introduce these architectural discussions in the context of Remote object invocation, we should be clear that they apply to all service integration with Flex—remote object, web service, and HTTP service.

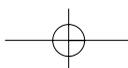
Rather than present a property-by-property discussion of the `<mx:RemoteObject>` tag, our aim in this chapter is to empower J2EE developers to recognize how they can adapt J2EE best practices into RIA development with Flex. We’ll start by presenting some general concepts with regard to Java and Flex integration, and show how these concepts are supported by Remote object behavior. We’ll then suggest some best practices that can be adopted in your own development; first to expose server-side business logic, and then to facilitate invocation of the business logic, and handling results returned by the server in the appropriate context.

Finally, we’ll build a sample application, demonstrating many of our best practices as they apply to the development of an RIA that requires J2EE integration. We’ll build the “account summary and statement” feature of an online banking application, in which account information for a customer is fetched from middleware residing on the J2EE server.

Let’s start with a general look at the concepts involved in tying Java and Flex.

Java and Flex Concepts

Prior to the release of Flex, J2EE developers could tightly integrate their middleware with a Flash-based client by using the Flash Remoting technology. This was introduced by one of the authors in the Macromedia Press title *Reality J2EE—Architecting for Flash MX*. If you’re familiar with the concepts of Flash Remoting, you should be able to quickly grasp the concepts presented in this section.



The manner of invocation of Java methods residing in a J2EE application server is actually very similar in Flex to the handling of web services and REST architectures, using the `WebService` and `HTTPService` tags introduced in Chapter 19, “Flex Integration with Web Services.”

In our experience, however, if you’re an experienced J2EE developer who has been developing Web applications, perhaps by using frameworks such as Jakarta Struts or based around Java Server Pages (JSP) or similar template-driven technologies, there are some important concepts that are worth grasping before developing rich-client interfaces for your existing or new J2EE middleware.

Maintaining State

The biggest leap in understanding to be made is that there’s no longer a need for HTTP session state. Although your requests between rich client and server are ultimately being made over HTTP, HTTP has been reduced to a transport technology instead of a technology that we need to integrate with. J2EE developers are comfortable with the concept of placing attributes and parameters in the HTTP request and response objects, and maintaining state by storing objects in the HTTP session. With an RIA, state can and should be maintained in the client, which is now a stateful client, as opposed to the stateless thin client that HTML technologies offer.

The Flex Presentation Server acts as a mediator between your Flex RIA and your J2EE middleware. Although Flex can broker your requests, and serialize and deserialize your requests over HTTP, this no longer need concern you in your day-to-day development.

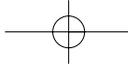
We’ll discuss later how Flex can in fact offer access to the HTTP session and allow the Flex client to treat the HTTP session as “just another object on the server.” However, in our experience, using the HTTP session should be a “bad smell” that identifies a refactoring opportunity to shift some of your application state onto the client.

Asynchronous Request-and-Response Model

With a traditional Web application, the model of request and response is such that the user makes a request to the server (by submitting a form, for instance) and then must wait: while the request is passed to the server, while the server processes the response, while the results are passed back to the server, and until the page the user was viewing is refreshed with the next page in the application. From submitting the request to receiving the response, the application is “blocked.”

With an RIA, the page-based development metaphor is finally laid to rest. An application can make a request to the server when it needs to update information on the server or to fetch data from the server. However, rather than “stall” or “block” the application, this activity is performed behind the scenes, allowing the user to continue interacting with the application. At a future point—most likely immediately as far as the user is concerned—the server will have data in the form of a result to pass back to the RIA client. The Flex gateway notifies the client that data has arrived, allowing the client to kick into life the client-side logic necessary to deal with the results returned from the server.

Enterprise developers entrenched in the JSP mindset—or indeed any developer who has been developing with Web application technologies—need to remember when making a request to the server that the application should carry on with a separate callback method prepared to handle the response at a future point in time. We’ll discuss this topic in more detail later in this chapter.



4 Chapter 20 FLEX INTEGRATION WITH J2EE

Exposing J2EE Classes to an RIA

The Flex Presentation Server incorporates a gateway servlet, which acts in role of “front controller,” brokering all requests between the Flex client and the J2EE middleware, and manages the serialization and deserialization of data in both directions.

With a Flex RIA development, the gateway is deployed within the same application server as your J2EE middleware, which means that the gateway is capable of invoking business methods on behalf of the Flex client. For J2EE developers, this means that there is zero additional work required to prepare middleware for an RIA front-end.

AMF versus SOAP

An RIA client built with Flex makes requests from the client to invoke business methods residing in the J2EE application server. These requests might require the passing of data as parameters to the request, which will start life as types or objects in the client-side ActionScript 2.0 language, but will become the equivalent Java types or objects by the time the Flex gateway invokes the methods on the server-side.

Furthermore, the middleware is likely to return results to the client; these will be Java types or objects passed “over the wire” from server to client. As these objects are passed over the wire, Flex transparently converts them from Java objects into appropriate ActionScript objects. We’ll discuss shortly some design patterns that allows us to maintain a consistent object model across the client and server.

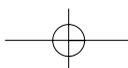
In many enterprises, standards-based support is considered to be of paramount importance; consequently, Flex is capable of wrapping the invocation of methods and the passing of data between client and server as Simple Object Access Protocol (SOAP) calls. As developers, we can use the Remote object features of Flex without having to get our hands dirty with SOAP, but can assure management that all client-server communication is a standards-based SOAP message transfer underneath the covers.

However, there is a performance penalty to be paid when objects are serialized and deserialized into an XML structure because XML is a text-based format. Flash Remoting introduced the Action Media Format (AMF) protocol as a means of passing messages and data over HTTP using a binary protocol proprietary to Flash. Flash Player—the virtual machine that resides in the browser—can make and receive AMF requests, which show significant performance benefits over equivalent SOAP requests.

By default, Flex uses the AMF protocol behind the scenes for all Java method invocations and data transfers using the `<mx:RemoteObject>` tag. We recommend changing these to SOAP if and only if the business insists on standards-based communication, and would otherwise enjoy the performance achieved with the AMF protocol, particularly when passing large data sets.

Interacting with Java Objects

The Flex gateway enables us to invoke methods from a Flex RIA client, upon Java objects that reside in the same J2EE application server as Flex.



Flex can invoke methods on Plain Old Java Objects (POJOs) that live in the Web application class-path, as well as being able to invoke methods on Enterprise JavaBeans (EJBs). Additionally, we can invoke methods on servlets that are deployed within the Web application. Because a JSP is essentially a servlet, we can invoke methods that are exposed on a JSP as well.

Servlet and EJB Integration

In this book, we focus on integration with POJOs, and further advocate that even if you have an EJB tier, you should access these EJBs through a Business Delegate class on the server that is itself a POJO responsible for JNDI lookup and invocation of EJB methods. This will also allow you to use SOAP rather than AMF, if you prefer, to invoke business logic in your EJB tier.

We wouldn't typically advocate integration with a Servlet tier—this most often indicates too fine-grained invocation of methods between client and server, and suggests that some business logic could be moved onto the client. However, if you're implementing an RIA interface on an existing J2EE infrastructure that has business logic contained in your Servlet tier, consult the Flex documentation to see how you can use RemoteObject to invoke methods exposed on Servlets in your app server.

As discussed in the previous section, the AMF protocol is used by default for remote method invocation and data transfer. Using the AMF protocol, we can invoke methods on all the Java objects described previously. However, if we want to use the SOAP protocol for our client-server integration, this will limit us to integrating with POJOs.

In our discussion on technical architecture later in this chapter, we present a strong case for architecting our applications so that we need only invoke methods on POJOs anyway.

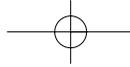
Making Objects Available for Remote Invocation

So how do we make a Java object available to a Flex client? We'll discuss shortly the `<mx:RemoteObject>` tag, which is responsible for invoking business methods on a Java object—however, we must first consider how to make these objects visible to Remote object invocation.

Unsurprisingly, the core requirement for Remote object is that a Java class be available on the class-path of the Flex application. This means that classes will be deployed into the `WEB-INF/classes` directory under the Flex application or contained in JAR files in the Web application's `WEB-INF/lib` directory.

Turning this requirement on its head, we can see that an existing Web application that has the appropriate Flex libraries and configuration files deployed within it, can become the middleware for an RIA without any changes to the application itself.

The astute software engineer might at this stage be concerned about the *entire* application suddenly being available for an RIA client to invoke methods on. Thankfully, Flex not only offers integration with the J2EE security architecture, but honors deployment descriptors that allow us to configure access on a class-by-class basis within the middleware. We'll explore the securing of Java objects as well as web services and HTTP services in the following chapter—for now, we can be confident that security of our middleware is in no way compromised by exposing it to an RIA interface.



6 Chapter 20 FLEX INTEGRATION WITH J2EE

Having made an object or class on the server available to our Flex client, we use the `<mx:RemoteObject>` tag in MXML to name the services on the server that we intend invoking methods on.

Let's now see how to make these method invocations.

Invoking Java Methods with Remote Objects

We'll explore the syntax for the `<mx:RemoteObject>` tag later in this chapter in the context of an online banking example. Right now, you need to get a grasp of the concepts.

Having declared the Remote object in our MXML application on the client, we now have a proxy for the Remote object on the server. Having declared the Remote object proxy, which is a client-side object, we can then call methods on the proxy *as if* they were methods on the client-side. This concept will be comfortable and familiar to developers who have worked with the EJB specification—the `<mx:RemoteObject>` tag is equivalent to creating a home interface that proxies methods on the remote EJB. In the same way that we can invoke methods on the home interface and leave the EJB container to worry about passing data over the wire, invoking the method on the remote EJB and passing data back to the client, we can use the Remote object to pass data over the wire, invoke the remote Java object, and pass data back to the client.

With an EJB, we have to specifically name the methods on the EJB that we want to invoke on our home interface. However, with the Remote object, we have the added luxury of only having to name the class on the server and being able to invoke public methods on it by name. We *can* name the services if we choose to, and doing so affords us the ability to configure how Flex behaves when invoking the method, such as specifying the way multiple calls to the same method should be handled. We'll demonstrate this shortly in our online banking example.

Handling Results from Method Calls

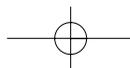
If you're new to RIA development, the following point is worth reiterating—method calls in RIA development with Flex are *asynchronous*. This means that the method call is made, control is returned immediately to the application, and at some future point the client is notified of results being returned from the server.

To handle the notification, when we call a method on Remote object we specify a handler function that's called when the result is passed back from the server. Although we can specify a global result handler that is responsible for handling the results of all method calls on a class, it makes more sense for us to specify result handlers for every method that we call on a Remote object. We'll discuss strategies for handling method results in our online banking application later in this chapter.

In addition to handling results from method calls, we also need to be able to handle errors that occur. Borrowing the language of web service invocation (and to offer a common interface across different service types), Flex offers the concept of fault-handling on Remote object calls.

Passing Data Between a Flex Client and Java Server

We discussed how to declare the classes on the server on which we want to invoke business logic, and how to call methods on these Remote objects and handle the results from these Remote object method calls. All that remains is to understand how data is passed backward and forward over the wire.



Java supports a number of different types and collections, such as Number, Boolean, String, ArrayList, and Date. Similarly, ActionScript 2.0 has its own concept of types, including Number, Boolean, String, Array, and Date. In these base cases, it's easy to see what must be done to pass data across the client/server divide.

Consider a client-side method call on a Remote object responsible for making a reservation. The `makeReservation()` method accepts as arguments `startDate`, `endDate`, `numberOfAdults`, and `numberOfChildren`, which are Date and Number objects, respectively.

The Remote object call is defined as follows:

```
<mx:Remote object id="bookings"
    source="com.iterationtwo.ReservationManager">
  <mx:method name="makeReservation" />
</mx:Remote object>
```

The `com.iterationtwo.ReservationManager` class has been assigned an ID of `bookings`, which is now the local proxy to the Remote class.

When we call the `makeReservation()` method from the client, we can use ActionScript 2.0 as follows:

```
var startDate:Date = bookingForm.startDate.selectedDate;
var endDate:Date = bookingForm.endDate.selectedDate;
var numberOfAdults:Number = bookingForm.adults;
var numberOfChildren:Number = bookingForm.children;
bookings.makeReservation( startDate, endDate,
    numberOfAdults, numberOfChildren );
```

When the method call is made on the Remote object, Flex collects the parameters and serializes them for the remote method call made over HTTP to the Flex gateway on the server.

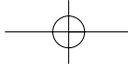
On the server side, the Flex gateway receives the HTTP request, recognizing not only the method call it's being instructed to make, but the data that has been passed. The Flex gateway is then responsible for creating the equivalent request parameters on the server side before invoking the remote method.

The Flex gateway then passes results from the method invocation back to the client over the HTTP connection. On the client side, the results are translated into the equivalent ActionScript 2.0 types, which correspond to what was returned from Java.

What's important to understand here is that Flex insulates us from the details of mapping types between the client and the server; we can invoke methods from within ActionScript, passing ActionScript arguments to the method calls. Flex then invokes the methods on the server using equivalent Java arguments, and returns to the client the results from the Java method call with the results translated once again to appropriate ActionScript objects on the client.

Translation of Custom Objects

In a typical enterprise development, the arguments to business methods and the return types from business methods are less likely to be native attributes such as Strings or Numbers; they are more likely to be complex business objects such as Customer, Account, Product, or CreditCard.



8 Chapter 20 FLEX INTEGRATION WITH J2EE

A number of recognized design patterns, such as the Value Object or Data Transfer Object, are used to represent business objects that are typically transferred between different tiers of a software application. A typical business method on a J2EE middleware might look like this:

```
public AccountVO getShareDealingAccount( CustomerVO customer ) { ... }
```

In the preceding code line, `AccountVO` and `CustomerVO` are both value objects that can contain a number of attributes, some of which can be native types such as `String` or `Number`; others can be other custom objects (a `CustomerVO` can contain an `AddressVO`, for instance, or an `AccountVO` can contain an array of `TransactionVO` objects).

The good news is that the Flex gateway assumes responsibility for passing these custom objects back and forth across the wire as well—if we define these objects on the server and also define the objects on the client, Flex can map between these objects.

In the online banking application example later in this chapter, we'll discuss the design patterns and techniques that enable us to maintain a consistent object model across the client and server, abstracting the developer entirely from the details of mapping objects between the client and server.

J2EE Technical Architecture

Before we embark on the development of a sample online banking application, let's consider some of the design patterns and techniques that are useful in RIA development.

At iteration::two, we embraced and advocated the “Core J2EE Design Patterns” in our server-side architecture, and have extended this experience onto client-side development in `ActionScript 2.0`.

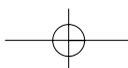
In this section, we offer a brief discussion of the J2EE architectures that are most suited to immediate RIA development, before introducing a number of design patterns that can be implemented in the client-tier to enable the building of robust, scalable, and maintainable RIAs using techniques and patterns that will be familiar to seasoned J2EE developers.

Considerations for J2EE Architecture

The Service Oriented Architecture (SOA) describes a software architecture in which middleware is exposed as business services that can be invoked in some manner by an external client. These business services are typically loosely coupled services that can be aggregated to solve a particular business problem.

SOA has gained widespread popularity due to its suitability as an architecture from which web services can be exposed. Similarly, adopting a SOA facilitates the provision of a rich-client user interface and specifically aids in the provision of an RIA user experience on existing or custom middleware deployed in a J2EE application server.

The technologies that can be employed in a middleware solution are numerous, requiring their own specific domain experience. If we consider the problem of persistence alone, in a J2EE environment we can choose a number of technology offerings.



The JDBC specification allows us to develop code that interrogates a database directly, taking responsibility for sending of SQL queries to a database and parsing of the RecordSets that are returned.

Alternatively, we might elect to use a technology such as Entity EJB, which offers a form of object-to-relational mapping while managing additional features such as security and transactions on our behalf. More recently, lightweight persistence frameworks such as the Hibernate tool are becoming commonplace, which again require specific domain knowledge as to how to build a hibernate solution.

Business Delegates

The Core J2EE Pattern catalog from Sun describes the Business Delegate design pattern as a means of providing a business interface that hides the developer from the implementation details of underlying technologies.

The Business Delegate pattern offers the “point of contact” with the business tier in an application. Whether the underlying technologies comprise JDBC database integration, stateless session bean facades on a tier of entity beans, or a persistence framework such as Hibernate, the end developer doesn’t care. Similarly, if middleware is integrating with a CICS mainframe, an MQ Series message queue using a JMS provider, or web service integration, presenting a Business Delegate class offers a technology-independent means of using these services.

More importantly, the Business Delegate pattern forms a point of integration where all necessary business logic is pulled up, exposing a well-defined API that can be used to employ the underlying services.

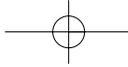
At iteration::two, we recommend pulling all related business logic in our middleware into a Business Delegate class, and configuring the Flex whitelist so that the Delegate class or classes are the only classes exposed to the Flex client.

Flex Whitelist

The Flex whitelist is the means by which you can define which Java classes and web services that Flex applications are allowed to execute through the server-side proxy. The whitelist is covered in more detail in the next chapter.

Though Flex is capable of invoking methods directly on EJBs, we *still* suggest that an EJB—even if a stateless session façade that aggregates EJB access into a business objects—be exposed to an RIA through a plain old Business Delegate Java object.

Although this requirement is by no means strict, we advocate that this architecture allows simple unit testing of our service interface, provides a simple interface to the RIA client that is completely agnostic of server-side implementation, and encourages the “stubbing” of business services during development of new business logic.



Flex Technical Architecture

The discussion that follows is not necessarily particular to J2EE integration, but can apply equally well to the integration of an RIA client using other services such as web services or HTTP services. However, we introduce these concepts now because the decision to integrate with J2EE middleware is typical for the development of an enterprise RIA, in which the number of use cases or tasks is typically high—tens or hundreds, rather than ones or twos.

As the number of tasks scales, so too does the number of methods we want to invoke, as well as the number of different contexts in which we want to invoke these methods. We must consider how best to manage these different services, and how to partition our application architecture so that the invocations of these services are not scattered around our application, but are catalogued in well-defined places in our architecture.

These problems that we want to address are not unique to RIA development, and have been documented and solved by the software community using a collection of different design patterns.

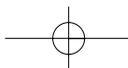
What we describe here are the best-practices advocated at iteration::two. They are not the *best* practices, but simply practices that fit with our understanding of building enterprise RIAs in a predictable and repeatable fashion. You might find best practices that fit within your own development practices—consider the patterns and practices that follow as a starting point for your own development only.

Many of the patterns presented here were already presented by the authors in “ActionScript 2.0 Design Patterns for Rich Internet Applications,” which can be found in the ActionScript 2.0 Dictionary from Macromedia Press. However, Macromedia Flex encourages us to consider alternative implementations of the patterns by using a combination of MXML and ActionScript 2.0.

RIA Service Locator

The problem of looking up remote services and establishing connections to these services is typically solved by the service locator class. Prior to Flex, we recommended a strategy whereby a singleton class provided the capability to look up services by name, returning an instance of a class that could be used as a proxy to the remote service. With Flex, much of the effort required in implementing the Service Locator pattern has been done for us:

- **Named Services**—The `<mx:RemoteObject>` tag allows us to specify not only the location of a class that should be available for integration, but to specify these classes within the `flex-config.xml` deployment descriptor. By using a deployment descriptor, we can name services from configuration files, and look up these services in MXML using these canonical names. This feature will be explained further in the next chapter.
- **Return of a Service Object**—Prior to Flex, our ActionScript 2.0 implementation of the Service Locator was responsible for looking up the Remote object, creating a connection to that object, and returning an instance of a proxy class that could be used to invoke methods on the service. The `<mx:RemoteObject>` tag is an MXML construct that achieves all this without the requirement for ActionScript 2.0 coding.



MXML and the `<mx:RemoteObject>` tag, in collaboration with the `flex-config.xml` file, provides most of the components of a Service Locator implementation. All that we recommend in addition to these features is that the definition of services using the Remote object, web service, and HTTP service tags be provided in a single place in a file called `services.mxml`.

In our main MXML application, we can then instantiate our services as follows:

```
<i2:services id="services" />
```

The `services.mxml` file shown as follows contains our service definitions:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:mx="http://www.macromedia.com/2003/mxml" height="0"
visibility="false">

  <mx:RemoteObject id="accountDelegate"
    result="event.call.resultHandler( event.result )"
    source="com.iterationtwo.banking.business.AccountDelegate">
    <mx:method name="createAccount" />
  </mx:RemoteObject>

</mx:Canvas>
```

This code then allows us to invoke services as follows:

```
services.accountDelegate.createAccount( customerID, account );
```

This code provides a simple and sufficient implementation of the Service Locator pattern.

RIA Front Controller

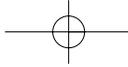
As an RIA scales in complexity, the number of tasks in the application increases accordingly. Furthermore, as an RIA increases in size, the number of ways in which a particular task can be invoked increases—toolbars, menu options, context menus and shortcuts such as double-clicking or dragging and dropping content—all conspire to kick off the same business logic that implements a particular task.

We need a mechanism by which we can dispatch control to appropriate business logic in response to a number of different “requests” in the form of user gestures or changes of application state. Furthermore, we want to centralize the point to which these requests are routed and the dispatch takes place, so that scaling our application is achieved by scaling a single class rather than scattering requests around our application architecture.

This pattern should be familiar to J2EE developers as that solved by the Front Controller pattern. In J2EE, the Front Controller is typically implemented as a single servlet to which all HTTP requests are routed, with a request parameter indicating the nature of the request. For those familiar with the Struts architecture, the Struts Controller is a specific implementation of the Front Controller, with specific Action classes in place to handle use-case-specific business logic.

We achieve a Front Controller implementation in ActionScript 2.0 with a collaboration of two key classes that work in an event-driven manner:

- **Controller**—Configured to listen for specific events and delegate control to specific worker classes when these events occur.



12 | Chapter 20 FLEX INTEGRATION WITH J2EE

- **EventBroadcaster**—Responsible for broadcasting events to the controller in response to user-gestures, changes in application state, and so on.

The `EventBroadcaster` class is implemented as a singleton, with a private constructor and a `getInstance()` method responsible for enforcing singleton access. The class instantiates an `EventDispatcher` class—an ActionScript 2.0 class that decorates a class with the ability to broadcast events to registered listeners. It provides a method, `broadcastEvent()`, that is capable of broadcasting a named event, with some accompanying data. The implementation of the class is shown following:

```
import com.iterationtwo.banking.control.Event;
import mx.events.EventDispatcher;

class com.iterationtwo.banking.control.EventBroadcaster
{
    public static function getInstance()
    {
        if ( eventBroadcaster == undefined )
            eventBroadcaster = new EventBroadcaster();

        return eventBroadcaster;
    }

    private function EventBroadcaster()
    {
        EventDispatcher.initialize( this );
    }

    public function broadcastEvent( eventName:String, eventData:Object )
    {
        var event:Event = new Event();
        event.type = eventName;
        event.data = eventData;

        dispatchEvent( event );
    }

    public var dispatchEvent:Function;
    public var addEventListener:Function;
    public var removeEventListener:Function;

    private static var eventBroadcaster;
}

```

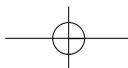
In collaboration with the `EventBroadcaster` class, we have our `Controller` class, which is registered with the `EventBroadcaster` to listen for events that it dispatches and delegate control to appropriate worker or “command” classes as appropriate.

The code for the `Controller` that we’ll use shortly in the online banking example is shown as follows:

```
import com.iterationtwo.banking.control.*;
import com.iterationtwo.banking.commands.*;
import mx.events.*;

class com.iterationtwo.banking.control.BankingController
{
    public function BankingController()
    {
    }
}

```



```
{
    commands = new Array();
    initialiseCommands();
}

private function initialiseCommands()
{
    addCommand( "fetchPortfolio", new FetchPortfolioCommand() );
    addCommand( "displayStatement", new DisplayStatementCommand() );
    addCommand( "displayTransaction", new DisplayTransactionCommand() );
    addCommand( "makePayment", new PaymentCommand() );
}

public function handleEvent( event:Event )
{
    executeCommand( event );
}

private function executeCommand( event:Event )
{
    var command:Command = getCommand( event.type );
    command.execute( event );
}

private function addCommand( commandName:String, commandRef:Command )
{
    commands[ commandName ] = commandRef;
    EventBroadcaster.getInstance().addEventListener( commandName, this );
}

private function getCommand ( commandName ):Command
{
    var command:Command = commands[ commandName ];
    if ( command == null )
        trace( "Command not found for " + commandName );

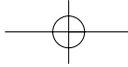
    return command;
}

private var commands:Array;
}
```

Reminder: You Say initialize(), I Say initialise()

We emphasized this point in a previous chapter, but remember that from time to time, our code will contain the UK English spelling of `initialise()`—where *initialize* is used as a Flex keyword, use the U.S. spelling, but be forgiving with our code, in which we often found ourselves reverting to the common UK spelling of the word!

The constructor of our Controller is responsible for creating a list of commands that it's willing to dispatch control to; the `initialiseCommands()` method uses the `addCommand()` helper method to add these commands to a lookup table of event name and command class. In adding the commands like this, the Controller registers itself with the dispatcher to listen for the event name, and provides a mechanism—through the `handleEvent()` method—by which the appropriate command class is executed in response to the event being broadcast.



14 | Chapter 20 FLEX INTEGRATION WITH J2EE

The key thing to notice in our Controller implementation at this stage is that the addition of a new use-case is as simple as adding a new `addCommand()` method call to the `initialiseCommands()` method on the Controller.

Using our Controller in a Flex application is as simple as instantiating the Controller within our MXML application, as follows:

```
<mx:Script>
    controller = new BankingController();
    EventBroadcaster.getInstance().broadcastEvent( "fetchPortfolio" );
</mx:Script>
```

In the preceding example, we also cause a `fetchPortfolio` event to be broadcast, which (as can be seen in our Controller implementation) causes control to pass to the `FetchPortfolioCommand`.

You now see how user gestures, or application states, can cause the broadcasting of an event, and how that event can cause control to pass to an appropriate worker or command class. Let's now take a look at how we implement these use-case-specific command classes.

RIA Command Pattern

It's difficult to introduce the previous Controller pattern without also talking about the Command Pattern, which is a typical collaborator. The Controller is responsible for recognizing events corresponding to user gestures and changes of application state, and then delegating the “work” associated with responding to these requests to worker or command classes.

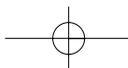
By using the Command Pattern to manage the “work” associated with each request, we achieve a scalable architecture in a simple fashion. As the complexity of an application increases, the number of use-cases also typically increases—each use-case largely corresponds to the creation of a new command class and the registering of an event name with the Controller, which should cause control to pass to this command class.

The Command Pattern

The Command Pattern, often called the “Service to Worker” pattern in J2EE literature (and familiar to many as a “Struts Action” when using the Struts framework), is one of the original behavioral patterns advocated in the “Gang of Four” *Design Patterns* book, by Erich Gamma et al.

The Controller needs to be able to execute an arbitrary number of commands in the same way—we achieve this in ActionScript 2.0 by defining a common class interface that all command classes are required to implement:

```
import com.iterationtwo.banking.control.Event;
interface com.iterationtwo.banking.commands.Command
{
    function execute( event:Event ):Void;
}
```



If you review our Controller implementation, you'll see the following methods:

```
public function handleEvent( event:Event )
{
    executeCommand( event );
}

private function executeCommand( event:Event )
{
    var command:Command = getCommand( event.type );
    command.execute( event );
}
```

When the Controller receives an event, the `handleEvent()` method passes it to the `executeCommand()` helper method. By inspecting the `type` property of the event—which corresponds to the event name—we can fetch the command class responsible for responding to this event. Because every command implements the `Command` interface, we can then invoke the `execute()` method on the command with no further details as to which command we're actually executing.

In the online banking application, which we'll build shortly, we have a `displayTransaction` event that's broadcast when the user selects a transaction on the bank statement. In response to this event, we want to update a read-only form with specific details regarding the transaction. Our code is as follows:

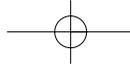
```
class com.iterationtwo.banking.commands.DisplayTransactionCommand
    implements Command
{
    public function DisplayTransactionCommand()
    {
    }

    public function execute( event:Event ):Void
    {
        var transaction:TransactionVO = (TransactionVO) event.data;
        updateTransactionDetails( transaction );
    }
    ...
}
```

The body of the `execute()` method contains the code responsible for updating the display with the details of the particular transaction selected. Ensuring that this command is called when the `displayTransaction` event is broadcast is as simple as adding the following lines to our Controller:

```
private function initialiseCommands()
{
    addCommand( "displayTransaction", new DisplayTransactionCommand() );
}
```

The command class is an incredibly powerful pattern to implement in an RIA architecture—in our experience, it makes the addition of business logic in an enterprise RIA development significantly easier, and it works particularly well with large development teams working on the same project. A typical feature addition requires the addition of new command classes that might or might not require the addition of new business services that the classes can invoke.



16 | Chapter 20 FLEX INTEGRATION WITH J2EE

As a rule, we try to keep our command classes relatively simple—they don't perform any complex business logic of their own; instead, they perform as follows:

1. Fetch data from the user interface.
2. Invoke business services in a particular context by using the data fetched from the user interface.
3. Handle the results of the business service and update the user interface with these results.

Although the command classes can be considered brokers for business logic, they delegate business logic to another class rather than perform any complex business logic on their own. Conveniently, we call this class the Business Delegate class.

Why Delegate?

An important question to ask is this: Why should we delegate business logic, and not perform it within the command class? Typically, business services can be used in a number of different contexts—for instance, in the mail client we built in the previous chapter, the service of sending an email can be used to send a new message, reply to a message, or forward a message.

However, the way we prepare the user interface, what we do after we send our message, might differ in these different contexts. Rather than implement the “send e-mail message” business logic in three different places, we would find ourselves extracting a method to send e-mail out into a common class and using that method in different contexts.

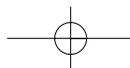
Business Delegate is the result of a common refactoring; extracting business services into a common class, which can be used in several contexts by several command classes. In our experience of building RIAs, this is a refactoring that is inevitably applied and merits consideration at the outset of an RIA technical architecture.

RIA Business Delegate Pattern

The concept of the Business Delegate pattern is likely to be familiar to J2EE developers who know of this pattern from the Core J2EE Patterns catalogue. The Business Delegate represents a clear point of integration between client and server. Where developers are solely responsible for client-side development and integration of the client with the server, the Business Delegate provides the necessary level of abstraction that allows them to use server-side business services without caring about the implementation details or location of these services.

The Business Delegate typically fulfills its role in collaboration with the Service Locator; it uses the Service Locator to locate and look up remote services, such as web services, remote Java objects, or HTTP services. Once located, the Business Delegate invokes these services on behalf of the class that has delegated responsibility to it for business logic invocation.

The Command class is responsible for deciding which business services should be invoked and ensures that the appropriate data is collected to call these services.



Handling Results from Remote Object Calls

Earlier in this chapter, we introduced the concept of asynchronous callbacks—when a Remote object call is made, at some future point a callback method is invoked on the calling object, with the results passed back from the server.

Consider the following Business Delegate:

```
class com.iterationtwo.banking.business.AccountDelegate
{
    public function AccountDelegate()
    {
        this.delegate =
        mx.core.Application.application.services.accountDelegate;
    }

    public function getPortfolio() : Void
    {
        delegate.getAccountPortfolio();
    }
    ...
}
```

In our Business Delegate constructor, we look up our delegate on the Service Locator, which can be found at `mx.core.Application.application.services` and subsequently invoke the `getAccountPortfolio()` method on the Business Delegate.

The `getPortfolio()` method is called on the Business Delegate by using the Remote object declaration found in the services.mxml file:

```
<mx:Remote object id="accountDelegate"
    result="handleResult( event.result )"
    source="com.iterationtwo.banking.business.AccountDelegate">
    <mx:method name="getAccountPortfolio"/>
</mx:Remote object>
```

The `result` property that is assigned on the `<mx:RemoteObject>` tag is used to specify which method should be called with the result when a method is called on the Remote object. Hopefully, you're already seeing problems with this for anything but the simplest of developments.

Understanding Pending Calls

Realistically, what we want to do with the results is likely to differ for each method that we call on a Remote object. It's therefore more sensible that we specify when we *call* a method on a Remote object that we also say what we want to do with the results. We achieve this by using the Pending-Call object that's returned every time we make a Remote object call in ActionScript.

We can rewrite our Delegate method, `getPortfolio()`, as follows:

```
public function getPortfolio() : Void
{
    var pendingCall = delegate.getAccountPortfolio();
}
```

18 | Chapter 20 FLEX INTEGRATION WITH J2EE

In the preceding method, we captured the pending call object that's returned from the method invocation. Due to the time-based "threaded" model that Flash Player operates on, the physical call to the server-side method doesn't occur until the `getPortfolio()` function call completes. Consequently, we can attach objects dynamically at runtime to the Call object that's returned, as follows:

```
public function getPortfolio() : Void
{
    var pendingCall = delegate.getAccountPortfolio();
    pendingCall.resultHandler = mx.utils.Delegate.create(this, myHandler);
}
private function myHandler( result ) : Void
{
    // the results are passed by the server back to this handler
}
```

What we did here is dynamically attach a property to the Call object, `resultHandler`, which we can point at a function in our Delegate that we want to handle the actual result of the server-side call. All that remains is to tell the Remote object in our Service Locator class (`services.mxml`) to call the `resultHandler` function that we attached to our Call, which we do as follows:

```
<mx:Remote object id="accountDelegate"
    result="event.call.resultHandler( event.result )"
    source="com.iterationtwo.banking.business.AccountDelegate">
    <mx:method name="getAccountPortfolio"/>
</mx:Remote object>
```

Business Delegate and Service Locator Collaboration

Let's examine the collaboration between the Business Delegate and Service Locator again. The Business Delegate will look up the Service Locator to locate the Remote object called `accountDelegate` and will then invoke the `getPortfolio()` method on the Remote object. Invoking the method on the Remote object returns an object, which represents the server-side call that *will* be made as soon as the Delegate method is complete.

Dynamically, before the method is complete, we attach a `resultHandler` property to our pending call, which we point at runtime to the function that we want to handle the results of our Remote object call when they're returned from the server.

In our Service Locator, `services.mxml`, we tell the Remote object call that it should inspect the pending call object for a method call when the results come in, and it should pass those results to the `resultHandler` that we attached to the pending call before the server-side method was invoked. This will pass the results back to the `myHandler()` method, which we attached to the `resultHandler` on our call, which is free to deal with the results as we see fit! We made sure to pass in a function Delegate, as described in Chapter 16, "Managing Application Workflow with Events," to make sure that we don't run into scope issues.

This collaboration gives us a scheme whereby the results of a server-side method call can be handled differently for each call we make on the server side. This liberates us to invoke our server-side logic in many different contexts—the same server-side methods can be used in different ways in different aspects of our application.

We're not done yet, though—you'll learn one more trick that really reaps rewards for having a collaboration of command classes, a Business Delegate, and a Service Locator in a client-side architecture.

Handling Results in the Context of the Command Class

Remember that the users of the Business Delegate are the individual command classes that the controller “passes the buck” to, depending on the user-gesture or change of application state that it has recognized by event broadcasting.

In the preceding scheme, we specified a generic fault handler on the Remote object definition in our Service Locator. Within the Business Delegate, we could assign a result handler to handle the results. However, our result handler was a private method, which we had to add to the Business Delegate. Let's consider a typical use-case and see why this is still restrictive.

Consider the `getPortfolio()` method on our previous Business Delegate and the different contexts in which we might choose to call that method:

- On application startup, we can call this method from a command class that is responsible for displaying a summary to a banking customer of all their accounts. After the results are returned, we would then populate the UI with the appropriate summary information, perhaps encouraging the user to drill down into the details of one particular account.
- We might also elect to give the banking customer the ability to view detailed reports about their current net worth. We can call the exact same `getPortfolio()` method on our Business Delegate, which would invoke the same server-side business service and return the same results. This time, however, we want to present these results to the user in a pop-up window as graphs and data prepared in a printed report format.

In both these instances, the decision about what to do with the results is best made in the context of the command being executed. To put it another way, we want our Business Delegate to route the results back to the command that called it, rather than attempt to handle them itself.

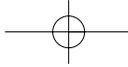
Let's look at how we can do that.

Turning Commands into Responders

First of all, let's change the constructor of our Business Delegate class, so that each time a command class requires to invoke business services, it creates a new Business Delegate (which essentially proxies the business services) and registers itself as the “responder”—the object wanting to handle the results of any asynchronous server-side calls.

Why Responder?

We chose the term “responder” because this was the term first used in Flash Remoting, and one that we discussed in detail in “Reality J2EE—Architecting for Flash MX”. The responder describes the instance of a class that's willing to take responsibility for hosting the method that will handle the results of a server-side method call.



20 Chapter 20 FLEX INTEGRATION WITH J2EE

Our Delegate constructor now looks like this:

```
class com.iterationtwo.banking.business.AccountDelegate
{
    public function AccountDelegate( callingCommand:Responder )
    {
        this.delegate =
        mx.core.Application.application.services.accountDelegate;
        this.responder = callingCommand;
    }
    ...
    private var responder:Responder;
    private var delegate:Object;
}
```

When the Delegate is created, it now has a reference to the command that called it, which we designate as being the responder. As you can see in the constructor for our Delegate, we strongly typed the `callingCommand` as having to implement a Responder interface.

This interface ensures that irrespective of what else the responder does, it provides methods called `onResult()` and `onFault()` that are capable of handling the results or faults that are returned from service calls.

```
interface com.iterationtwo.banking.business.Responder
{
    public function onResult( result ) : Void;
    public function onFault( fault ) : Void;
}
```

Now, when we have a command class that will invoke server-side business logic through the appropriate Business Delegate, we can insist upon our command implementing the responder interface, as well as having the command register itself with the Business Delegate as responsible for handling returned results.

```
class com.iterationtwo.banking.commands.FetchPortfolioCommand
    implements Command, Responder
{
    public var delegate:AccountDelegate;

    public function FetchPortfolioCommand()
    {
        this.delegate = new AccountDelegate( this );
    }

    public function onResult( result ) : Void
    {
        var portfolio:AccountPortfolioVO = (AccountPortfolioVO) result;
        mx.core.Application.application.portfolio = portfolio;
    }

    public function onFault( fault ) : Void
    {
    }
    ...
}
```

The highlighted code shows the changes we made. Our command now implements the `Responder` interface and must therefore provide the `onResult()` and `onFault()` methods that the interface defines. Furthermore, our command passes itself to the `AccountDelegate` as the `Responder` object, so that the `AccountDelegate` knows to pass the results it receives from the server back to our command.

There's one additional caveat concerning the scoping of objects—and that requires us to delve into Event Delegates.

Using the Event Delegate to Pass Scope

In the `Business Delegate` class, you learned how to attach the result handler to be called to the pending call object that's returned from a `Remote` object call. Now that we're passing a reference to our responder (the calling command class) in the `Delegate` constructor, we might be tempted to use the following line method call:

```
public function getPortfolio() : Void
{
    var pendingCall = delegate.getAccountPortfolio();
    pendingCall.resultHandler = this.responder.onResult;
}
```

However, there's a problem with this code that is related to the scoping issues of event handlers that we first highlighted in Chapter 16. When the `onResult()` method is called on the responder, references to `this` within the responder do not necessarily refer to the `Responder` instance itself—it depends entirely on the scoping rules for event handlers.

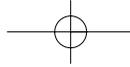
To ensure that `this` in our responder refers to the `Responder` instance, we can use a feature of Flex called the `Event Delegate`. The following implementation of `getPortfolio()` ensures that when the command class handles the result from server-side calls in its `onResult()` method, the command can use the `this` keyword to safely refer to its own scope:

```
public function getPortfolio() : Void
{
    var pendingCall = delegate.getAccountPortfolio();
    pendingCall.resultHandler = mx.utils.Delegate.create( Object(
        this.responder ), this.responder.onResult );
}
```

Casting the Responder as an Object

We used a cast to turn `this.responder` into an `Object`. The implementation of strict-typing in `ActionScript 2.0` currently doesn't recognize that an instance of an object that has been previously defined as an `Interface` type (that is, `Responder`) is a valid type-match in a signature expecting an `Object`. Our workaround is to cast our responder to an `Object` explicitly, which keeps the type-checker happy and allows our code to compile. Expect this workaround to be no longer required in future revisions of `ActionScript 2.0`.

Let's summarize the collaboration of our `Business Delegate` and command classes that allow our commands to act in the role of responders for service results.



22 | Chapter 20 FLEX INTEGRATION WITH J2EE

The command class registers itself with the Delegate as a responder by passing a reference to itself in the Delegate constructor. Furthermore, if the command invokes business services through the Business Delegate, it implements the responder interface and provides appropriate `onResult()` and `onFault()` methods.

Our Delegate now captures the pending call object that's returned when any service calls are made, such as when using the `<mx:RemoteObject>` tag. Knowing that the service call is not actually made until the method executing the call returns, we can attach a `resultHandler` function to the pending call object, which allows us to route results and faults from the service call back to the command class, which can handle them in context. Finally, to ensure that there are no issues with regard to scoping when the responder handles the result or fault events, we use the `mx.utils.Delegate.create()` method to delegate the event handling to our command, rather than assume that the command can handle it without any scoping issues.

Let's now take a look at one final but highly important design pattern—the value object—which we can use to pass data between our application tiers, irrespective of whether they reside on the Flex client or the application server.

RIA Value Objects

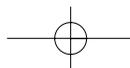
In an enterprise application, several tiers are likely to exist in the software architecture. Architects often talk of a three-tier or n -tier design to describe the layers of architecture that are logically decoupled from each other. In the Microsoft Enterprise Patterns, this architecture is described as the “layered architecture.” In this book, we have been clear about advocating a three-tier client-side architecture of presentation, business, and integration tier.

The different tiers in the application also reflect different levels of abstraction; at the presentation tier, we're concerned only with the visible properties of an object from the perspective of a user. Within the business tier, an object can be furnished with additional information for the purposes of computation or to reflect relationships between business objects. Finally, in the integration tier, data is represented at its lowest level, using language-specific features such as SQL result sets.

Rather than expose developers to the implementation details of the integration tier, we can define data transfer objects (DTOs) that are responsible for passing data between these tiers as business data that reflects the domain objects in our system—customers, bank accounts, employees, flights, reservations, and so on. Because these objects are free of implementation detail and business logic—and instead reflect only the value of data, such as the specific age of a customer or the specific balance of a bank account—we call these objects *value objects*.

The value object pattern is documented in the Core J2EE Pattern catalogue as a pattern that allows the bulk transfer of serialized data. At iteration::two, we embraced this pattern in our own Action-Script 2.0 Pattern catalogue to ensure that we could do the following:

- Encapsulate the implementation of business objects
- Establish a “currency” for data passed between the various tiers on the client
- Establish a common object model on the client and the server



- Enable data to be passed transparently between the client and the server, and treated in the same manner on both sides of the “wire”

In all of our business methods, in the Business Delegate class for instance, we enforce by convention that parameters to method calls and return-types from method calls are scalar types, value objects, or arrays of value objects.

Anatomy of a Value Object

Let’s take a look at a sample value object that represents a customer account in an online banking application:

```
class com.iterationtwo.banking.vo.AccountVO
{
    public function AccountVO()
    {
        _remoteClass = "com.iterationtwo.banking.vo.AccountVO";
    }

    public var _remoteClass : String;

    public var accountNumber:String;
    public var sortCode:String;
    public var title:String;
    public var transactions:Array;
}
```

We are interested initially in the highlighted lines of code; quite simply, they describe the public attributes that we want to expose on our value object to hold specific business data concerning a bank account.

As shown in our declarations of the Transactions attribute, we defined an Array for transactions—this Array can correspond to an Array of other value objects, as you’ll see shortly in our example.

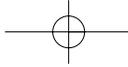
Value Object Accessors

In J2EE, we would likely encapsulate our attributes as private attributes, with public getters and setters, but in ActionScript, we recommend that you consider the following strategy that we adopt at iteration::two.

In the simplest case in which we expect to be only getting or setting attributes, we choose to declare the attributes as we’ve just shown, with public variables. Setting or getting attributes on a value object is then as simple as this:

```
var account:AccountVO = new AccountVO();
account.accountNumber = "00310712";
mx.core.Application.alert( "Account Number is "+account.accountNumber );
```

If the requirement on our value objects becomes more complex; perhaps we can do something a little more clever in setting an attribute, for instance. Then we can encapsulate it by adding getter and setter methods for the attribute.



24 | Chapter 20 FLEX INTEGRATION WITH J2EE

In ActionScript 2.0, we can use a feature known as *implicit getters and setters* and perform the following refactoring:

```
public get accountNumber():String;
public set accountNumber( account:String )
{
    this._accountNumber = "RN/"+account;
}
private var _accountNumber:String;
```

In this refactoring, we change the access of our `accountNumber` attribute from public to private, and rename the attribute. (At iteration::two, our convention is to prefix with an underscore, although many Macromedia classes use underscores in their implementations—and the compiler might warn you if you happen to clash.) We then use the notation for an implicit getter and setter function, with our function containing the same name we had for our public attribute. This then ensures that the following code calls our setter—the implicit getter and setter ensure that a function call is treated as if it were an attribute:

```
account.accountNumber = "00310712";
```

In the preceding example, our motivation for the refactoring is shown as a requirement that account numbers be prefixed with the fixed length string "RN/".

This refactoring allows us to migrate from public attributes to public implicit getters and setter, without affecting any client code (code that performs gets or sets).

We recommend this strategy over coding your own Java Bean style getters and setters.

Saving a Private Variable

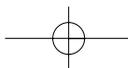
When a value object is serialized from ActionScript and sent over the wire to be deserialized into a Java object, you should be aware that private attributes on the client side are serialized and sent over the wire as well.

A easy mistake is to assume that only public attributes will be passed over the wire, according to the Java Bean serialization model. We recommend that you simply be mindful this issue when storing data in your value objects that you want to pass over the wire. However, if you *really* want to prevent your variables from being serialized, you can add the following code to the value object constructor:

```
class com.iterationtwo.banking.vo.AccountVO
{
    public function AccountVO()
    {
        _remoteClass = "com.iterationtwo.banking.vo.AccountVO";
        _global.ASSetPropFlags( this, "accountNumber", 1 );
    }

    public var _remoteClass : String;
    private var accountNumber : String;
    ...
}
```

The highlighted line of code instructs the ActionScript compiler to mark the `accountNumber` property as hidden.



ASSetPropFlags—Magic Numbers

The number 1 in `ASSetPropFlags` is actually the decimal for the bit-pattern 001, in which the most-significant bit (MSB) of the pattern represents “protect from overwriting,” the middle bit represents “protect from deletion,” and the least-significant bit (LSB) represents the “hidden” property. Thus, a pattern of 101 not only hides the property, but also prevents it from being overwritten. Note that `ASSetPropFlags` is not an officially documented method, so it might go away in future versions of Flex; but it’s safe to use in version 1.0 and is actually used internally in many mx classes.

Mapping Client-Side Value Objects to the Server

Notice that we placed our value objects in the namespace `com.iterationtwo.banking.vo`. At `iteration::two`, our practice is to keep the same package structure on the client as we maintain on the J2EE server—making it simple for developers working on the integration of client and server to find their way around the package structure.

As value objects are passed to arguments on our Business Delegate class and ultimately passed over the wire in Remote object calls, they can be mapped to their server-side equivalents. We say “can be mapped” because it’s up to us to coerce Flex into performing the mapping by somehow tying the client and server-side value objects together:

```
class com.iterationtwo.banking.vo.AccountVO
{
    public function AccountVO()
    {
        _remoteClass = "com.iterationtwo.banking.vo.AccountVO";
    }

    public var _remoteClass : String;

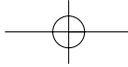
    public var accountNumber:String;
    public var sortCode:String;
    public var title:String;
    public var transactions:Array;
}
```

Let’s look at our initial value object definition again; we highlighted the pertinent sections of code.

`_remoteClass` and `Object.registerClass()`

Implicitly, Flex maps objects between the client and server using a default set of mappings. (You’ll find these mappings in the Flex documentation that ships with the product.)

For a value object that is not a type implicitly handled by Flex, we can specify a `_remoteClass` property on the client-side value object that specifies the name of the equivalent server-side value object. We don’t need to make any changes to the server-side value objects; the knowledge of the mapping is required on the client-side only.



Server-Side VOs: Keeping It Serializable

Here's a common mistake made by J2EE developers who already have a server-side data model comprising a collection of value objects: forgetting to make these value objects implement the `java.io.Serializable` interface. Flex is actually a bit clever about this, and the serializable implementation isn't essential—however, if you build an RIA that may also use Flash Remoting, this is essential.

If a `_remoteClass` attribute is not supplied to point at an appropriate Java class, Flex deserializes the object into a `HashMap` on the server by using the property names in the client-side value objects as `HashMap` keys with appropriate values.

If a client-side value object contains other value objects or even Arrays of value objects as attributes, Flex recursively traverses these during serialization and deserialization, provided that the “dependent value objects” (the child value objects) also have the `_remoteClass` property correctly specified.

Beware of the Circular Reference

When recursively traversing an object graph, the graph must be in the form of a tree in which where there are no circular references—a child containing a reference back to its parent. If the object graph is cyclic (containing circular references), the server will ultimately throw a `StackOverflowException` because the serialization and deserialization get stuck running around the parent-child cycle in the graph.

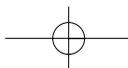
If you're asking Flex to serialize and deserialize object graphs, be careful to ensure that these object graphs are always “non-cyclic;” that is, in tree or list form.

The previous scheme will work irrespective of whether the protocol used for the Remote object is AMF or SOAP encoding.

If you've developed RIAs using J2EE, Flash Remoting, and Flash, you might be familiar with the `Object.registerClass()` scheme for value object mapping, which the authors first discussed in “Reality J2EE—Architecting for Flash MX.” In our client-side value object, we can place the following line of code in our attribute declarations as an alternative to `_remoteClass`:

```
private static var doRegister:Boolean = Object.registerClass(
    "com.iterationtwo.banking.vo.AccountVO", AccountVO );
```

This line of code registers our mapping between the fully qualified server-side value object class name (the first argument) to the constructor of our client-side value object. With this line of code in place, our serialization and deserialization of value objects and value object graphs also works. However, the `Object.registerClass()` scheme works only with AMF encoding;—not with SOAP encoding. We recommend that you stick with the Flex strategy of `_remoteClass` being defined on the client-side value object, so that your value objects can be mapped between the client-side and server-side of your Flex RIA, irrespective of using the AMF or SOAP protocol.



iteration::two Online Banking Application

In this chapter so far, we introduced the `<mx:RemoteObject>` tag as an alternative service to the web service and HTTP service tags. This tag offers a tightly coupled service invocation between a Flex client and J2EE middleware.

More importantly, we used this final discussion of service integration technologies to introduce some design pattern concepts that can be applied from our previous experience of J2EE and RIA development to the architecture and development of Flex applications.

In the pages that follow, we'll put some of this design and architecture into practice with the beginnings of an application for online banking. In the sample application that follows, we'll put in place an architecture of Controller and Event Broadcaster, commands, Business Delegate, and Service Locator to invoke business logic on a J2EE application server.

User Interface

The user interface for our finished application is shown in **Figure 20.1**. The interface is contained within a Panel layout, adding a MenuBar and a two-panel layout to show a summary of the customer's account portfolio, and the detailed transactions in the currently selected account. In the right panel (underneath the bank statement), a number of tabs offer the opportunity to perform common operations on the currently selected account—including paying bills, making account transfers, setting up debits and standing orders, or simply viewing the details of a selected transaction and allowing that transaction to be assigned to a spending category.

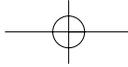
Figure 20.1

The iteration::two online banking RIA demonstration.

The screenshot displays the user interface of the 'iteration::two Online Banking RIA'. The window title is 'Iteration::two Online Banking RIA'. The interface is divided into several sections:

- Menu Bar:** Accounts, Quote, Reports, Help.
- Accounts Summary:**
 - Summary of Your Accounts
 - Print Statement of Account
 - Spending Summary
 - Budget Report
- Account Details:**
 - Account Number: 004
 - Sort Code: 80-61-46
 - Current Cleared Balance: 1250.00
 - Current Overdrafts: 100.00
 - Funds Available: 1350.00
 - i2Bank Savings Account
 - i2Bank High Interest Acco...
- i2Bank Current Account - Statement Details:**

Date	Description	Type	Amount	Balance
Fri Mar 12	Checking Account transaction 0	CDT	0	0
Fri Mar 12	Checking Account transaction 1	CDT	10	10
Fri Mar 12	Checking Account transaction 2	CDT	20	30
Fri Mar 12	Checking Account transaction 3	CDT	30	60
Fri Mar 12	Checking Account transaction 4	CDT	40	100
Fri Mar 12	Checking Account transaction 5	CDT	50	150
Fri Mar 12	Checking Account transaction 6	CDT	60	210
Fri Mar 12	Checking Account transaction 7	CDT	70	280
Fri Mar 12	Checking Account transaction 8	CDT	80	360
Fri Mar 12	Checking Account transaction 9	CDT	90	450
Fri Mar 12	Checking Account transaction 10	CDT	100	550
Fri Mar 12	Checking Account transaction 11	CDT	110	660
Fri Mar 12	Checking Account transaction 12	CDT	120	780
Fri Mar 12	Checking Account transaction 13	CDT	130	910
Fri Mar 12	Checking Account transaction 14	CDT	140	1050
Fri Mar 12	Checking Account transaction 15	CDT	150	1200
Fri Mar 12	Checking Account transaction 16	CDT	160	1360
Fri Mar 12	Checking Account transaction 17	CDT	170	1530
- Transaction Details:**
 - Date: Fri Mar 12 14:38:36 GMT+0000 2004
 - Description: Checking Account transaction 2
 - Amount: 20
 - Category: Entertainment
- Help Text:** Did you know that you can assign a spending category to this transaction, and optionally tell i2Bank to assign all future transactions from Checking Account transaction 2 against this category. By assigning your spending to categories, you can make great use of our reporting features, available from the reporting menu!



28 | Chapter 20 FLEX INTEGRATION WITH J2EE

The user interface comprises the following MXML files and custom components:

- `bankaccounts.mxml`—The main MXML file, which builds the UI, instantiates the Service Locator, and constructs the MenuBar, AccountSummary panel, and AccountDetails panel.
- `appMenubar.mxml`—The application MenuBar.
- `AccountSummary.mxml`—The left panel that displays summary information for each account in the customers portfolio.
- `AccountDetails.mxml`—The right panel that displays the DataGrid of detailed account transactions, as well as the TabNavigator allowing different operations to be performed on the account shown.

The AccountDetails component instantiates a custom component for each of the five operations that appear in the TabNavigator:

`TransactionDetails.mxml`—A form that uses data binding to display the details of the currently selected item in the DataGrid of transactions, contained within the AccountDetails component.

`Transfer.mxml`—A form that enables the user to transfer money between the accounts in the portfolio.

`Payments.mxml`—A form that allows the user to schedule payments to be made from his account to another bank account, such as a bill payment.

`StandingOrder.mxml`—A form that allows the user to instruct a standing order payment to be made from their account, such as a mortgage payment.

`DirectDebit.mxml`—A form that allows the user to set up a debit instruction that allows a utility company for instance, to draw funds directly from the account each month to settle a bill payment.

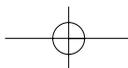
In addition to these user interface components, the project includes two other files:

- `main.css`—A stylesheet responsible for embedding fonts and setting the styles used in the application.
- `services.mxml`—Our MXML Service Locator class that contains the Remote object definitions for the J2EE banking services.

Because the focus of this chapter is on Remote object behavior and how to implement Remote object invocation within a scalable software architecture, we won't list the code in the above files here—the code is available for download from the companion Web site for this book.

J2EE Middleware: Server-Side AccountDelegate and Value Objects

It's not our intention to present the complete source for the J2EE middleware that we're using in our sample application because it also available for download from the companion Web site.



In summary, however, we implemented an `AccountDelegate` Business Delegate on the server that exposes a `getPortfolio()` method on the server and passes back an `AccountPortfolioVO` value object to the client as follows:

```
package com.iterationtwo.banking.business;

import com.iterationtwo.banking.vo.*;

public class AccountDelegate
{
    public AccountDelegate()
    {
    }

    public AccountPortfolioVO getAccountPortfolio()
    {
    }
    ...
}
```

The `AccountPortfolioVO` value object contains an Array of `AccountVO` value objects. The `AccountVO` value object contains miscellaneous attributes that correspond to an account, along with an Array of `TransactionVO` value objects that represent the individual transactions that have been performed on an account.

In a real-world implementation, the `getAccountPortfolio()` method would construct the `AccountPortfolioVO` as the result of a number of CICS mainframe transactions using the Java Connector Architecture (JCA) or perhaps be built as from a call to a stateless session EJB that performs a façade function to several entity EJBs. In our example, to enable the reader to deploy and test the example with minimum fuss, we elected to construct a fake `AccountPortfolioVO` in the body of the Business Delegate method ourselves.

Our value objects reside in the `com.iterationtwo.banking.vo` package; we'll mirror the package structure and class naming for the client side ActionScript 2.0 value objects in the same way as on the server.

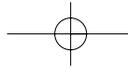
The compiled classes can be packaged in a JAR file and deployed into the `WEB-INF/lib` directory under your Flex Web application context, or the classes themselves can be deployed in a package structure underneath the `WEB-INF/classes` directory on your application server.

Fetching the Portfolio of Accounts

The first task we want to complete is to ensure that on application startup, the summary panel on the left side of the screen displays a summary of the accounts in the user's portfolio.

Using the pattern-based framework described earlier in this chapter, this requires that we do the following:

1. Create the client-side value objects to map to the server-side value objects.
2. Create a Remote object in our Service Locator that will contain our server-side business method `getPortfolio()`.



30 | Chapter 20 FLEX INTEGRATION WITH J2EE

3. Add a business method, `getPortfolio()`, to our Business Delegate to call the remote service.
4. Register a `fetchPortfolio` event with the Front Controller.
5. Create a `FetchPortfolioCommand` command class that will call the Business Delegate to fetch the account portfolio from the server, acting as a responder to handle the results of the Remote object call, so that it can populate the left-hand panel with the account summary.

Let's follow each of these steps in turn.

Creating `AccountPortfolioVO` in ActionScript 2.0

The first thing we need to do is create a client-side value object that will map to the `AccountPortfolioVO` class on our server. It's our convention at `iteration::two` to package and name these classes identically on the server, so developers can quickly begin to treat the classes the same, irrespective of which side of the wire they're working on. Our class definition is as follows:

```
class com.iterationtwo.banking.vo.AccountPortfolioVO
{
    public function AccountPortfolioVO()
    {
        _remoteClass = "com.iterationtwo.banking.vo.AccountPortfolioVO";
    }

    public var accounts:Array;
    public var _remoteClass : String;
}
```

Essentially, the value object sets the `_remoteClass` attribute to map to the server-side value object and declares the `accounts` Array, which will contain the `AccountVO` objects we described earlier in this chapter.

Adding a Remote Object to the Service Locator

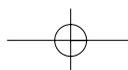
On the server-side, we have an `AccountDelegate` class in the package `com.iterationtwo.banking.business` that contains our `getPortfolio()` method that we'll be using to fetch our portfolio of accounts.

In our architecture discussion, we described how we could add all our services to a single Service Locator implemented in MXML and reference that Service Locator from our delegate whenever we wished to invoke business services.

We add the following implementation of `services.mxml`:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Canvas xmlns:mx="http://www.macromedia.com/2003/mxml" height="0"
visible="false">

<!-- ===== -->
<!-- Remote Java Objects -->
<!-- ===== -->
```



```

<mx:Remote object id="accountDelegate"
    result="event.call.resultHandler( event.result )"
    source="com.iterationtwo.banking.business.AccountDelegate">
  <mx:method name="getAccountPortfolio" />
</mx:Remote object>

</mx:Canvas>

```

Notice that we configured the result callback on our Remote object definition, enabling us to specify the responder as the `resultHandler` attribute on the pending call object returned to our delegate when services are invoked.

Finally, we add the following line to our main application, `bankaccounts.mxaml`, to instantiate our Service Locator:

```
<i2:services id="services" />
```

Now our main application has a lookup facility for services, and we defined a Remote object service for the server-side `AccountDelegate`.

Adding `getPortfolio()` to the Client-Side `AccountDelegate`

Next, we need to implement the `getPortfolio()` method on our client-side business delegate; the method will invoke the `getPortfolio()` method on our server-side delegate, located through our `<i2:services />` component, and will pass the result back to the Responder object that registered itself with the delegate when the delegate was created.

Our method is as follows:

```

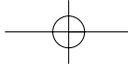
class com.iterationtwo.banking.business.AccountDelegate
{
    public function AccountDelegate( callingCommand:Responder )
    {
        this.delegate =
            mx.core.Application.application.services.accountDelegate;
        this.responder = callingCommand;
    }

    public function getPortfolio() : Void
    {
        var pc = delegate.getAccountPortfolio();
        pc.resultHandler = mx.utils.Delegate.create( Object( this.responder ),
            this.responder.onResult );
    }

    ...
}

```

The constructor for our delegate registers the calling command as the responder, and looks up our server-side delegate through the services component. We then invoke the server-side method in the highlighted line of code and attach a result handler to the pending call that will pass the results of the server-side call back to the command that called the client-side delegate.



32 | Chapter 20 FLEX INTEGRATION WITH J2EE

Registering the Event with the Controller

We're almost ready to create our `FetchPortfolioCommand` class; however, control will pass to that class only in response to an event that has been mapped to the invocation of that command. Using the Controller architecture, we add the following line to the `initialiseCommands()` method of our Controller class, located in `com.iterationtwo.banking.control.BankingController`:

```
private function initialiseCommands()
{
    addCommand( "fetchPortfolio", new FetchPortfolioCommand() );
}
```

This ensures that in response to a `fetchPortfolio` event being broadcast *anywhere* in our application, the `execute()` method on `FetchPortfolioCommand` will be called. We'll leave the reader to take a look at the implementation of the controller from the code on the companion Web site.

Creating the Command Class

The full code for our `FetchPortfolioCommand` is given below:

```
import com.iterationtwo.banking.business.*;
import com.iterationtwo.banking.commands.*;
import com.iterationtwo.banking.vo.*;
import com.iterationtwo.banking.control.*;

class com.iterationtwo.banking.commands.FetchPortfolioCommand
    implements Command, Responder
{
    public var delegate:AccountDelegate;

    public function FetchPortfolioCommand()
    {
        this.delegate = new AccountDelegate( this );
    }

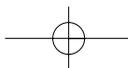
    public function execute( event:Event ):Void
    {
        delegate.getPortfolio();
    }

    public function onResult( result ) : Void
    {
        var portfolio:AccountPortfolioVO = (AccountPortfolioVO) result;
        mx.core.Application.application.portfolio = portfolio;
    }

    public function onFault( fault ) : Void
    {
    }
}
```

The key here is that our command class creates a new instance of the client-side `AccountDelegate` in its constructor, passing itself as the responder argument to the Delegate.

By implementing the `Responder` interface, our `FetchPortfolioCommand` is obliged to provide an `onResult()` and `onFault()` method as the `Responder` interface demands.



In the body of the `execute()` method, our command invokes the `getPortfolio()` command on the `AccountDelegate`, with our `onResult()` method handling the return of the `AccountPortfolioV0`.

In `bankaccounts.mxml`, the application state is maintained as a global variable, `portfolio`, of type `AccountPortfolioV0`, allowing the binding of UI components to the portfolio. By updating the global `portfolio` object from our command, any bindings for areas of the UI responsible for displaying portfolio information will be fired.

Firing an Event on Application Startup

Finally, we add the following code to `bankaccounts.mxml`:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    xmlns:i2="*"
    themeColor="haloSilver"
    initialize="applicationInit()" >

  <mx:Panel title="iteration::two Online Banking RIA"
    marginLeft="0" marginRight="0" marginTop="0" marginBottom="0">

    <mx:Script>
      import mx.containers.TitleWindow;
      import com.iterationtwo.banking.business.*;
      import com.iterationtwo.banking.control.*;
      import com.iterationtwo.banking.vo.*;

      public var controller:BankingController;
      public var portfolio:AccountPortfolioV0;

      public function applicationInit()
      {
        _global.style.modalTransparency = 50;
        controller = new BankingController();
        EventBroadcaster.getInstance().broadcastEvent( "fetchPortfolio" );
      }
    </mx:Script>
```

Instantiating Controller as an MXML Tag

In the body of our `applicationInit()` method, we create an instance of our controller explicitly in `ActionScript`. Because an MXML tag is essentially a class, an alternative is that we CREATE the Controller using an MXML tag, such as `<i2:BankingController id="controller"/>`.

The choice you make is a matter of style—we simply chose to show the `ActionScript` variant in this application.

We specify a startup method, `applicationInit()`, which creates a single instance of our `BankingController` before broadcasting the `fetchPortfolio` event. This means that on startup, our `FetchPortfolioCommand` immediately has its `execute()` method called, which results in the server-side `getPortfolio()` method being called. The `AccountPortfolioV0` returned from that server-side call

34 | Chapter 20 FLEX INTEGRATION WITH J2EE

is mapped into the appropriate `AccountPortfolioVO` value object on the client, including the dependant `AccountVO` and `TransactionVO` objects, and returned to the `FetchPortfolioCommand` `onResult()` handler, where the client-state is updated with the new portfolio.

All that remains is for us to display the summary details of this portfolio in our left panel.

Displaying an Account Summary

The following code demonstrates the implementation of the left panel, which shows a summary of accounts. The custom `AccountSummary` component defines a `portfolio` attribute, which is bound to the global `Portfolio` object during instantiation within `bankaccounts.mxml`. This ensures that every time the `portfolio` is fetched or updated, the `AccountSummary` will update any bindings on its `Portfolio` object.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.macromedia.com/2003/mxml">

  <mx:Script>
  <![CDATA[
    import com.iterationtwo.banking.vo.*;
    import com.iterationtwo.banking.control.*;

    public var portfolio:AccountPortfolioVO;
  ]]>
</mx:Script>

  <mx:Repeater id="list" dataProvider="{portfolio.accounts}">
    <mx:HBox>
      <mx:Label text="{list.currentItem.title}" width="150"/>
    </mx:HBox>
  </mx:Repeater>

</mx:Panel>
```

As can be seen from the highlighted lines of code, we repeat over each of the accounts in the global `Portfolio` object, and for each of those accounts, display the `title` attribute within a label.

Implementing a Sliding Panel Effect

Because we have more information than just the title that we might want to display, let's implement a "sliding panel" effect. When we click one of the labels, we want everything below to slide down, revealing a more detailed account summary. Clicking the summary again hides the details, whereas clicking another title hides the original details, and reveals the details for the new account.

We add the following ActionScript 2.0 into our Script block:

```
public function accountSummarySelected( index )
{
  // Resize the summary to display more details
  for ( var i=0; i < portfolio.accounts.length; i++ )
  {
    if ( i == index )
    {
```

```

        slideOpen( i );
    }
    else
    {
        slideClosed( i );
    }
}
}

private function slideOpen( index )
{
    var e = new mx.effects.Resize(detail[index]);
    e.heightTo = 120;
    e.duration = 300;
    e.playEffect();
}

private function slideClosed( index )
{
    var e = new mx.effects.Resize(detail[index]);
    e.heightTo = 0;
    e.duration = 300;
    e.playEffect();
}

```

The private helper methods are set up to slide a container within the repeater called `detail`, between a height of 0 or 300, to reveal the content within. With our `accountSummarySelected()` method ensuring that our selected account summary is slid open and all others are slid closed, we make the following changes to `AccountSummary.mxml`:

```

<mx:Repeater id="list" dataProvider="{portfolio.accounts}">
  <mx:HBox mouseDown="accountSummarySelected(
    event.target.repeaterIndices[0] )" >
    <mx:Label text="{list.currentItem.title}" width="150" />
  </mx:HBox>
  <mx:VBox id="detail"
    marginLeft="10" marginBottom="10" height="0"
    vScrollPolicy="off" hScrollPolicy="off">
    <mx:Label text="Account Number: {list.currentItem.accountNumber}" />
    <mx:Label text="Sort Code: {list.currentItem.sortCode}" />
    <mx:Label text="Current Cleared Balance: 1250.00" />
    <mx:Label text="Current Overdraft: 100.00" />
    <mx:Label text="Funds Available: 1350.00" />
  </mx:VBox>
</mx:Repeater>

```

Now, our `accountSummarySelected()` method is called when the account titles are clicked, revealing the contents of the `VBox` that we've named `detail`. These `VBox` containers are slid open or closed with the `ActionScript` methods we just defined.

In our main `bankaccounts.mxml` file, the following instantiation of the `AccountSummary` component ensures that the account summary reflects the portfolio fetched by our `FetchPortfolioCommand`:

```

<i2:AccountSummary title="Accounts Summary" portfolio="{portfolio}"
  width="350" heightFlex="1" />

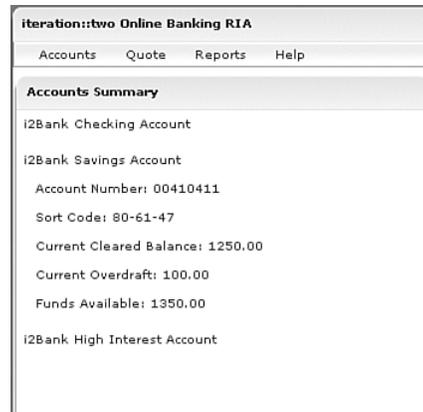
```

Figure 20.2 shows the summary of accounts, with an account having been selected to reveal further details.

36 | Chapter 20 FLEX INTEGRATION WITH J2EE

Figure 20.2

AccountSummary component, with an account selected to reveal further account details.



Displaying Transactions on an Account Statement

When an account is selected in the Accounts Summary panel, we want the detailed list of transactions, or “statement,” to be shown in the DataGrid that’s displayed in the AccountDetails component.

- Using our pattern-based framework, this will require that we do the following:
- Broadcast a `displayStatement` event to our controller with details of the account that has been selected.
- Instruct our Controller to respond to the `displayStatement` event with an appropriate `DisplayStatementCommand` command class.
- Implement the `DisplayStatementCommand` to update our DataGrid with details of the account transactions.

Let’s deal with each of these tasks in turn.

Broadcasting the `displayStatement` Event

We already implemented the `accountSummarySelected()` method in `AccountSummary.mxml`. The following amendment is all that’s necessary to ensure that an appropriately named event is broadcast when a summary has been selected with the mouse:

```
public function accountSummarySelected( index )
{
    // Cause an update of the AccountDetails panel
    var account:AccountVO = portfolio.accounts[ index ];
    EventBroadcaster.getInstance().broadcastEvent( "displayStatement",
    account );
    // Resize the summary to display more details
    ...
}
```

In the preceding example, not only do we broadcast the event, but we attach some data with the event—the value object representing the account that has been selected, which will include the account information and an Array of all the transactions that have been performed on the account, as an Array of `TransactionVO` value objects.

Registering `DisplayStatementCommand` with the Controller

Next, we update our `initialiseCommands()` method on our controller as follows:

```
private function initialiseCommands()
{
    addCommand( "fetchPortfolio", new FetchPortfolioCommand() );
    addCommand( "displayStatement", new DisplayStatementCommand() );
}
```

In only a matter of seconds, we have enabled an entirely new use-case in our application, and provided a scenario—selecting an account in summary view—from which the use-case is kicked into action through a `Command` class!

Implementing `DisplayStatementCommand`

The implementation of our `DisplayStatementCommand` is even simpler; the command is invoked as a result of the `displayStatement` event, which we know broadcasts an `AccountVO` as accompanying data. We pass this `AccountVO` to an `updateStatementDetails()` method, which will encapsulate the setting of the `DataGrid` with all the transactions on the account:

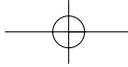
```
import com.iterationtwo.banking.business.*;
import com.iterationtwo.banking.commands.*;
import com.iterationtwo.banking.vo.*;
import com.iterationtwo.banking.control.*;

class com.iterationtwo.banking.commands.DisplayStatementCommand
    implements Command
{
    public function DisplayStatementCommand()
    {
    }

    public function execute( event:Event ):Void
    {
        var account:AccountVO = (AccountVO) event.data;
        updateStatementDetails( account );
    }
}
```

In our `AccountDetails` custom component, the `DataGrid` that renders the transactions has an identifier of `statement`, using a local Array of `TransactionVO` objects as its `dataProvider`, picking attributes from each `TransactionVO` for each column, as the following code shows:

```
<mx:DataGrid id="statement" widthFlex="1" heightFlex="7" >
    <mx:dataProvider>{transactions}</mx:dataProvider>
    <mx:columns>
        <mx:Array>
            <mx:DataGridColumn columnName="date" headerText="Date" />
        </mx:Array>
    </mx:columns>
</mx:DataGrid>
```



38 | Chapter 20 FLEX INTEGRATION WITH J2EE

```
<mx:DataGridColumn columnName="description" headerText="Description" />
<mx:DataGridColumn columnName="type" headerText="Type" />
<mx:DataGridColumn columnName="amount" headerText="Amount" />
<mx:DataGridColumn columnName="balance" headerText="Balance" />
</mx:Array>
</mx:columns>
</mx:DataGrid>
```

The implementation of the private `updateStatementDetails()` method on our `DisplayStatementCommand` is therefore as straightforward as this:

```
private function updateStatementDetails( account:AccountVO ) : Void
{
    mx.core.Application.application.account.transactions=
        account.transactions;
}
```

Again, we can see how a collaboration of Controller and command classes makes it extremely simple to add new functionality to our application without risking consequent effects to existing functionality.

Command classes and Business Delegate methods prove excellent opportunities for the unit testing of ActionScript 2.0 code using the AS2Unit framework.

Displaying Transaction Details

One of the forms displayed in the TabNavigator of the AccountDetails component is the detail of a transaction selected in the Data Grid.

This presents an interesting discussion—should we pull all the details that might need to be displayed by the client over the wire when the application starts up? Or should we operate in a master/detail view, in which clicking a transaction causes further data to be fetched from the server to then display more detail?

In essence, this is a trade-off of startup performance versus runtime performance, which is most likely to be an application-specific decision. One of the great benefits of an RIA is that the master/detail scenario can be handled “under the covers,” so that it’s almost imperceptible to the user. If we fetch details from the server on demand, the user need not suffer a page refresh or long wait for this information to be fetched.

At iteration::two, if we have a data set size that we think *might* be able to be pulled over the wire at startup without penalty to the user, we go with this in the first instance, only optimizing for a “deferred fetching” model when performance demands it. With a Command pattern implementation, this is a relatively simple refactoring that can be performed in a specific command class without affecting any other part of the application architecture.

This is another strong case for adopting a Command pattern-based architecture; the strong encapsulation of business logic affords us the luxury of targeted refactorings for performance and optimization as soon as they become critical.

Back to our example—we must make the following changes to our application:

1. Broadcast an appropriate `displayTransaction` event when a transaction is selected on the `DataGrid`, with the event containing a `TransactionVO` object representing the transaction.
2. Register the `displayTransaction` command with the Controller.
3. Implement `DisplayTransactionCommand`.

Broadcasting the `displayTransaction` Event

Broadcasting the appropriate event is as simple as our previous example. We add the following method to our `AccountDetails` component:

```
public function displayTransaction( event )
{
    var selectedRowIndex = event.itemIndex;
    EventBroadcaster.getInstance().broadcastEvent( "displayTransaction",
                                                    transactions[selectedRowIndex] );
}
```

We attach it to the `cellPress` event on the `DataGrid`:

```
<mx:DataGrid id="statement" widthFlex="1" heightFlex="7"
             cellPress="displayTransaction(event)" >
```

This notifies the Front Controller about our desire to display transaction details on the appropriate tab of our `TabNavigator`.

Registering the `displayTransaction` Event with the Controller

Registering a new event with the Controller is a simple one-line addition with our pattern-based architecture:

```
private function initialiseCommands()
{
    addCommand( "fetchPortfolio", new FetchPortfolioCommand() );
    addCommand( "displayStatement", new DisplayStatementCommand() );
    addCommand( "displayTransaction", new DisplayTransactionCommand() );
}
```

Implementing `DisplayTransactionCommand`

Finally, we need to implement the `DisplayTransactionCommand`, which receives the `TransactionVO` in the event broadcast by the Controller, and updates the user interface accordingly.

Again, we use a private helper method to update the user interface. Extracting this user interface logic into a private method is the first step toward performing the Extract Class refactoring. At iteration::two, we ultimately refactor this code into a View Helper class that's solely responsible for mediating between the command and the view.

Our `DisplayTransactionCommand` ends up as follows:

```
class com.iterationtwo.banking.commands.DisplayTransactionCommand
    implements Command
{
```

40 Chapter 20 FLEX INTEGRATION WITH J2EE

```

public function DisplayTransactionCommand()
{
}

public function execute( event:Event ):Void
{
    var transaction:TransactionVO = (TransactionVO) event.data;
    updateDetails( transaction );
}

private function updateDetails( transaction:TransactionVO ):Void
{
    var detailForm = mx.core.Application.application.account.tabs.details;
detailForm.date.text = transaction.date;
detailForm.description.text = transaction.description;
detailForm.amount.text = transaction.amount;
}
}

```

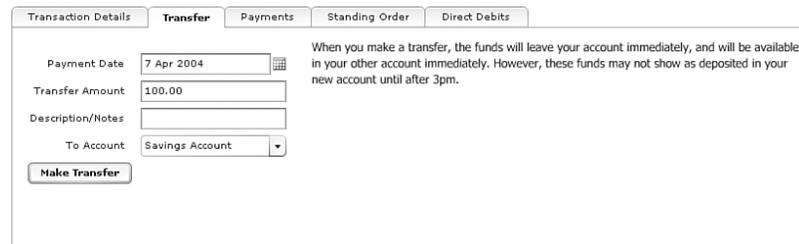
The highlighted code in the helper method is simply responsible for setting the appropriate form fields, located on the Details tab in the AccountDetails panel (referenced as `account.tab.details` from the top-level application), by picking out the appropriate fields from the `TransactionVO` that was broadcast with the event.

Transferring Between Accounts

In this final addition to our sample application, we'll commence the implementation of the Transfer tab on our AccountDetails panel. The transfer panel is shown in **Figure 20.3**.

Figure 20.3

The Transfer panel that allows transfer of money into another account.



To support the transfer of money, we register a new event, `makeTransfer`. In our Controller, we tie that event to `TransferCommand`, which is implemented as follows:

```

class com.iterationtwo.banking.commands.TransferCommand
    implements Command, Responder
{
    public var delegate:AccountDelegate;

    public function TransferCommand()
    {
        this.delegate = new AccountDelegate( this );
    }

    public function execute( event:Event ):Void
    {

```

```

        mx.core.Application.alert( "Transferring Money between Accounts" );
        delegate.makeTransfer();
    }

    public function onResult( result ):Void
    {
    }

    public function onFault( fault ) : Void
    {
    }
}

```

There is something different about our implementation of `TransferCommand`, however—we haven't shown the fetching of the form elements from the view, and instead just seem to invoke the `makeTransfer()` method on the Remote object.

Argument Binding

Argument binding is a quick and simple means of binding form inputs to the arguments for a Remote object invocation, or equally for binding Remote object results to the user interface. When we want to use our Remote object calls in a number of different contexts, argument binding can prove quite limiting. However, for single use-cases, such as only ever wanting to make transfers from our single form on the Transfer tab of the AccountDetails screen, we can define our Remote object method as follows:

```

<mx:Remote object id="accountDelegate"
    result="event.call.resultHandler( event.result )"
    source="com.iterationtwo.banking.business.AccountDelegate">
  <mx:method name="getAccountPortfolio"/>
  <mx:method name="makeTransfer">
    <mx:arguments>
      <paymentDate>
        {mx.core.Application.application.account.tabs.transfer.paymentDate}
      </paymentDate>
      <amount>
        {mx.core.Application.application.account.tabs.transfer.amount}
      </amount>
      <notes>
        {mx.core.Application.application.account.tabs.transfer.notes}
      </notes>
      <toAccount>
        {mx.core.Application.application.account.tabs.transfer.toAccount}
      </toAccount>
    </mx:arguments>
  </mx:method>
</mx:Remote object>

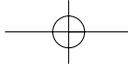
```

Using argument binding, we can specify the arguments to our Java method in order by using any names we like. The preceding highlighted method declaration is equivalent to calling a method on our J2EE delegate as follows:

```

public void makeTransfer( Date paymentDate, double amount,
    String notes, int accountId );

```



42 | Chapter 20 FLEX INTEGRATION WITH J2EE

One of the advantages of argument binding is that we can easily apply validators in our services.mxml to validate the bindings that we're making. For instance, if we want to ensure that only real numbers are provided for the amount of money to transfer, we can add the following validator to services.mxml:

```
<mx:NumberValidator
    field=" mx.core.Application.application.account.tabs.transfer.amount"
    domain="real"/>
```

At iteration::two, we're more likely to adopt a method of explicit parameter passing, calling our service methods from within our ActionScript command classes.

Specifying Concurrency

Finally, we add the following property to our method definition in services.mxml:

```
<mx:method name="makeTransfer" concurrency="last">
```

As with web services, we can specify a concurrency policy on individual method calls; in this example, the policy of `last` indicates that a new request to the `makeTransfer()` method should cancel any existing requests. Alternatively, we can provide a policy of `multiple` to allow all transfers to be processed (the default behavior) or a policy of `single` to ensure that only a single request is allowed.

Stateful J2EE Integration

We kept the following discussion to be last in this chapter because it's our experience that the biggest hurdle for J2EE developers when working with RIA architectures for the first time is the process of shifting state toward the client.

Many developers recognize the benefits that can be achieved with a rich-client architecture and further recognize Flex as being able to provide the rich and interactive controls with which they can deliver these experiences. However, it's easy to get stuck in the "JSP mindset" and attempt to refactor toward a rich-client architecture by replacing each JSP in the application with an equivalent MXML file. Although this is certainly possible, the real rich-client rewards are to be reaped when a collection of interacting JSPs are brought together under a single MXML application, replacing the form submissions required to navigate between these JSPs as asynchronous Remote object (or indeed web service or HTTP service) calls.

There might still be valid reasons why a rich-client architecture requires that some concept of state be shared between the client and the server; let's consider how this can be achieved with Flex.

Stateful Classes

By default, when we make a remote call on a J2EE class using the Remote object, a new object is created for each method call, as opposed to using the same object for consecutive method calls.

The `type` attribute on the Remote object can be used to specify a stateful class; the only restriction on what constitutes a stateful class is that it must reside on the server classpath and have a no-arguments constructor.

Consider a class on the server that's responsible for processing merchant card transactions; it might be necessary to maintain state during several transactions, requiring that we have a stateful class.

We can declare this Remote object in our Service Locator, `services.mxml`, as follows:

```
<mx:Remote object type="stateful-class" id="merchantPayment"
    source="com.iterationtwo.commerce.business.CardPayments" />
```

If not specified, the type would have defaulted to stateless class, which has been implicit in our previous examples.

Cross-Platform Compatibility and Performance

The implementation of stateful classes is achieved in Flex by using J2EE server sessions to maintain the state of stateful Java objects. This raises two important considerations for enterprise development: cross-platform compatibility and performance.

Session cookies are used to maintain the state; for clients that have cookie-handling disabled, the `response.encodeURL()` method can be used to append the `jsessionid` to the URL. Irrespective of which method is used, you should be aware that although Flash Player on Windows, Linux, and Unix supports server sessions, the stand-alone Macintosh Flash Player (the Flash Player that resides outside the browser, not the browser plug-in) does not have support for J2EE server sessions.

Finally, storing objects in the session has implications for memory performance because stateful session objects are stored in the J2EE session. As with traditional J2EE development, care should be taken with regard to the size of objects that are kept in the J2EE session—use the `stateless-class` type for Remote object access wherever possible.

Gaining Access to J2EE Sessions

Flex preconfigures a session servlet in the default `web.xml` file, which makes available a server-side Java object that can be used to access and modify the J2EE session for an application.

By placing one of the following Remote object definitions in our Service Locator at `services.mxml`, we make available to our application a service that can get and set objects in the J2EE session, as well as remove objects from the session:

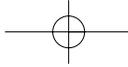
```
<mx:Remote object source="servlet" id="sessionObject" />
<mx:Remote object source="@ContextRoot()" id="sessionObject"/>
```

As with any other `<mx:RemoteObject>` tag, the session service can use result and fault event handlers to handle requests to fetch items from the session.

The `web.xml` descriptor defines the name of the session servlet as `servlet`; therefore, we use one of the following examples to work with the J2EE session:

```
sessionObject.session( "set", "customerID", 700 );
sessionObject.session( "get", "customerID" );
sessionObject.session( "remove", "customerID" );
```

These three examples demonstrate how the session service is used to get, set, or remove items in the J2EE session from a Flex client.



44 | Chapter 20 FLEX INTEGRATION WITH J2EE

As a best-practice, we recommend that access to the J2EE session be encapsulated in a client-side `SessionDelegate` Business Delegate in the same way that we recommend interaction with other server-side Java objects. Furthermore, because setting and getting items in the J2EE session is a relatively expensive operation compared with setting state on the client, we advocate using value objects in their role as “bulk accessors” to get and set a number of related properties.

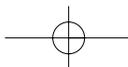
Summary

The actual mechanics of locating a Java class residing in a J2EE application server, remotely invoking business methods on that class, and returning these results to a Flex client are surprisingly simple—thanks to the power and flexibility of the `<mx:RemoteObject>` tag. In this chapter, we exposed the different ways in which the `<mx:RemoteObject>` tag can be used and explored the degree of configurability we can have in what objects we can call and how to call them.

More importantly, however, this chapter has served as a stepping stone for the development of enterprise-scale RIAs. We delved into the topic of design patterns for RIA development in more detail than anywhere else in this book, presenting design patterns that are likely to be familiar to J2EE developers and perhaps to a lesser extent (because we’ve drawn our Pattern catalogue from the knowledge of the Core J2EE Pattern catalogue), to .NET developers.

We showed how a rich collaboration of design patterns enables us to build an application framework that readily accepts the addition of new functionality and encourages refactoring by encapsulating behavior at appropriate levels of abstraction. One of the key benefits that we haven’t emphasized so far is how such an architecture enables the collaborative development of enterprise RIAs on large multidisciplinary teams by providing a clear and common vocabulary for solving business problems, and a clear and common understanding of where in the software architecture a particular new requirement should be accommodated.

We introduced the Service Locator pattern and recommended its implementation in MXML. This is a departure from the recommendation we would have made before the arrival of Flex, but the elegance of the `<mx:WebService>`, `<mx:HTTPService>`, and `<mx:RemoteObject>` tags, in addition to the concept of named services, supports the rapid development and ongoing maintenance of service-based architectures. In collaboration with the Service Locator, we presented the Business Delegate as a class to which all server-side business processing should be delegated, and have seen how the command class interacts with the Business Delegate class. We explored how to architect an interaction between the numerous command classes in our architecture and the lesser number of Business Delegate classes by using Event Delegates and Responder objects to ensure that asynchronous results are routed back to the command class that requested the invocation of a particular business service. Finally, we introduced a Controller and Event Dispatcher pairing that allows control of the application to pass to a command class as the result of form submissions, the selection of items from DataGrids or Menus, double-clicks, or drag-and-drop operations.



In our sample online banking application, we pulled together these design techniques and started to think about how an application comprising several MXML files, custom components, and collaborating ActionScript 2.0 framework can interact with custom middleware deployed on a J2EE application server. Although our banking application is in no way intended to be complete, it does serve to demonstrate how quickly new features can be added to an RIA, even when compared with Web application development using languages such as JSP.

Finally, we considered the means by which we can maintain state between a Flex RIA and a J2EE server, although we left this discussion until last in the hope that the developer reading this section recognizes how so much of an application state can gravitate naturally toward the client in an RIA environment.

In the chapter that follows, we'll tie together our understanding of web services, HTTP services, and remote Java services by exploring the important topic of security. By understanding how you can lock down what's exposed to a client on the server, as well as understanding how you can continue to secure communications over secure HTTP if required, you understand the most important final piece of the RIA development puzzle.

You have now gained a strong understanding of how to develop rich-client experiences with the alternate presentation tier offered by Flex and learned how ActionScript 2.0 and MXML can work together to enable the development of a client-side business tier. These last few chapters have strengthened your arsenal of RIA development techniques with the important aspects of integration with existing and custom infrastructure—we're now a chapter away from walking through a complete Flex application—from concept to delivery!

