3

Flex with RESTful Services

More frequently than not, Rails applications are written using a RESTful approach, which provides a coherent way of organizing your controller actions for applications that serve HTML pages. It also has the added benefit of exposing your application as a service that can be accessed via an API. This capability is important because it enables our Flex application to communicate with the Rails application by passing XML data using the Flex `HTTPService` components. In Chapter 2, Passing Data with XML, we looked at the underlying mechanisms of using the `HTTPService` component and the implications of using XML. In this chapter, we will look at the larger picture of how to access a RESTful Rails application and how to consume nested resources with custom verbs, which are common in RESTful applications. You should still be able to follow this chapter, even if you are not familiar with RESTs. This chapter will guide you through building the "Stock Portfolio" application, which will introduce REST concepts, such as CRUD verbs and nested resources, and use custom REST actions. You will see how to code this application from both a Rails and a Flex perspective.

Let's jump right into it.

Creating the Stock Portfolio Rails Application

The Stock Portfolio application is an online trading application that allows you to buy and sell stock. Of course, this sample application will walk you through what a RESTful Rails application is, even though it doesn't include many aspects that a real-world trading application needs. The data we want to manage is the following: An account holds positions in stock, for example, 50 shares of Google and 20 shares of Adobe. Each position has many movements created when the stock is bought or sold. To get started, let's create a new Rails application:

```
$ rails rails
$ cd rails
```

Now you can create the Account, Position, and Movements "resources" as follows:

```
$ ./script/generate scaffold Account name:string
$ ./script/generate scaffold Position account_id:integer quantity:integer\
 ticker:string name:string
$ ./script/generate scaffold Movement price:float date:datetime\
 quantity:integer position_id:integer operation:string
```

In Rails terms, a resource is data exposed by your Rails application following a convention to access and manipulate the data via HTTP requests. From a code point of view, this translates to a controller that can be invoked to create, read, update, and delete the data, and the controller will access the active record of concern to perform the requested action. To access the controller methods, define in the routes configuration file the exposed resources; this definition will dictate which URL can be used to access these resources. We will do this step by step hereafter. Again, when we mention a resource, think of it as combination of the URLs to manipulate the data, the controller that exposes the data, and the active record used to store the data.

The `script/generate` command is a facility to create the files we need as a starting point. We need to apply several changes to the generated code to get a fully functional application. If you look at the `script/generate` commands above, we specified the Account, Position, and Movement resources, their attributes, and how the resources are linked to each other. The Movement resource has a `position_id` column that links the movements to the positions, and the Position resource has an `account_id` column that links the positions to the accounts. The `script/generate` command does not add the code either to associate the active records or to constrain the controllers. Let's do that now. You can add it to the Account, Position, and Movement active records and add the `has_many` and `belongs_to` associations as follows:

```
class Account < ActiveRecord::Base
  has_many :positions, :dependent => :destroy
end

class Position < ActiveRecord::Base
  belongs_to :account
  has_many :movements, :dependent => :destroy
end

class Movement < ActiveRecord::Base
  belongs_to :position
end
```

This code will give you fully associated active records. Assuming you have some data in your database, you could, for example, find all the movements of the first position of the first account using the following Rails statement:

```
Account.first.positions.first.movements
```

Changing the active records was the easy part. The controllers will require more work because to respect and constrain the resource nesting, we want to ensure that the positions controller only returns positions for the specified account, and the movements controller only returns movements for the specified position. In other words, we want to have movements nested in positions and positions nested in accounts. Change the config/routes.rb file to the following:

```
ActionController::Routing::Routes.draw do |map|
  map.resources :accounts do |account|
    account.resources :positions do |position|
      position.resources :movements
    end
  end
end
```

`Routes` tells our application what URL to accept and how to route the incoming requests to the appropriate controller actions. By replacing three independent routes with nested routes, we indicate that, for example, the positions cannot be accessed outside the scope of an account. What URLs does the route file define now? From the command line, type the following `rake` command to find out:

```
$ rake routes | grep -v -E "(format|new|edit)"
```

The `rake routes` command gives you the list of all URLs as defined by your routes configuration file. We just pipe it into the `grep` command to remove from the list any extra URLs we don't want at this stage. For the account resource, we now have the URLs shown in Table 3.1.

Table 3.1 **URLs for the Account Resource**

HTTP Verb	URL	Controller
GET	/accounts	{:action=>"index", :controller=>"accounts"}
POST	/accounts	{:action=>"create", :controller=>"accounts"}
GET	/accounts/:id	{:action=>"show", :controller=>"accounts"}
PUT	/accounts/:id	{:action=>"update", :controller=>"accounts"}
DELETE	/accounts/:id	{:action=>"destroy", :controller=>"accounts"}

To access the positions, we need to prefix the URL with the account ID that nests the positions (see Table 3.2).

Table 3.2 **Account IDs Added as Prefixes to the URLs**

HTTP Verb	URL	Controller
GET	/accounts/:account_id/positions	{:action=>"index", :controller=>"positions"}
POST	/accounts/:account_id/positions	{:action=>"create", :controller=>"positions"}
GET	/accounts/:account_id/positions/:id	{:action=>"show", :controller=>"positions"}
PUT	/accounts/:account_id/positions/:id	{:action=>"update", :controller=>"positions"}
DELETE	/accounts/:account_id/positions/:id	{:action=>"destroy", :controller=>"positions"}

Finally, we need to prefix the URL with the account and position that nests the movements (see Table 3.3).

Table 3.3 **URL Prefixes to Nest the Movements**

HTTP Verb	URL	Controller
GET	/accounts/:account_id/positions/:position_id/movements	{:action=>"index", :controller=>"movements"}
POST	/accounts/:account_id/positions/:position_id/movements	{:action=>"create", :controller=>"movements"}
GET	/accounts/:account_id/positions/:position_id/movements/:id	{:action=>"show", :controller=>"movements"}
PUT	/accounts/:account_id/positions/:position_id/movements/:id	{:action=>"update", :controller=>"movements"}
DELETE	/accounts/:account_id/positions/:position_id/movements/:id	{:action=>"destroy", :controller=>"movements"}

List all the movements of the first position of the first account, for example, by using the following URL: http://localhost:3000/accounts/1/positions/1/movements.

Defining the routes makes sure the application supports the nested URLs. However, we now need to modify the controllers to enforce implementation of this nesting, so we'll add such constraints to all the controllers. But first, let's remove the HTML support from our controllers because, in our case, we want the Rails application to only serve XML, and we don't need to worry about supporting an HTML user interface. Let's simply remove the `respond_to` block from our controllers and keep the code used in the `format.xml` block. For example, we change the `index` method from the following:

```
class AccountsController < ApplicationController
  def index
    @accounts = Account.find(:all)

    respond_to do |format|
```

```
      format.html # index.html.erb
      format.xml  { render :xml => @accounts }
    end
  end
end
```

to the following:

```
class AccountsController < ApplicationController
  def index
    @accounts = Account.find(:all)
    render :xml => @accounts
  end
end
```

You can effectively consider the `respond_to` as a big switch in all your controller methods that provide support for the different types of invocations, such as rendering either HTML or XML. To constrain the positions controller, we will add `before_filter`, which will find the account from the request parameters and only query the positions of that account. Change the index method from the following implementation:

```
class PositionsController < ApplicationController
  def index
    @positions = Position.find(:all)

    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render :xml => @positions }
    end
  end
end
```

to this one:

```
class PositionsController < ApplicationController
  before_filter :get_account
  def index
    @positions = @account.positions.find(:all)
    render :xml => @positions.to_xml(:dasherize=>false)
  end
  protected
  def get_account
    @account = Account.find(params[:account_id])
  end
end
```

The `Position.find(:all)` was changed to `@account.positions.find(:all)`. This change ensures that only the positions for the specific account instance are returned.

The `before filter` loads that account for each method. We also are modifying the format of the returned XML to use underscores instead of dashes in the XML element names to better accommodate Flex, as explained in Chapter 2. When requesting the http://localhost:3000/accounts/1/positions URL, the controller now returns an XML list of all the positions with an ID of 1 that belong to the account. Now we do the same with the movements controller and scope the movements to a specific account and position, as follows:

```
class MovementsController < ApplicationController
  before_filter :get_account_and_position
  def index
    @movements = @position.movements.find(:all)
    render :xml => @movements.to_xml(:dasherize => false)
  end
  protected
  def get_account_and_position
    @account = Account.find(params[:account_id])
    @position = @account.positions.find(params[:position_id])
  end
end
```

So when requesting the http://localhost:3000/accounts/1/positions/1/movements URL, the controller returns an XML list of all the movements of the given position from the given account. First the account is retrieved, and then the positions from that account are queried, enforcing the scope of both the account and the position. Don't directly query the positions by using `Position.find(params[:position_id])` or a similar statement because the users could tamper with the URL and query the positions of a different account.

Before changing the rest of the methods, let's do some planning and see how we will use all the different controllers. Table 3.4 gives an overview of all the actions for our three controllers.

Table 3.4 **Overview of Actions of the Three Controllers**

Controller Method	Accounts Controller	Positions Controller	Movements Controller
Index	All accounts	All positions for account	All movements for position in account
Show	Not used	Not used	Not used
New	Not used	Not used	Not used
Edit	Not used	Not used	Not used
Create	Creates an account	Buy existing stock	Not used
Update	Updates an account	Not used	Not used
Destroy	Deletes the account	Sell stock	Not used
Customer verbs	None	Buy	None

For our application, several nonrelevant methods don't apply when rendering XML that would apply when supporting an HTML user interface. For example, the controller doesn't need to generate an edit form because the Flex application maps the XML to a form. In the same way, we don't need the `new` action, which returns an empty HTML entry form. Additionally, as in our case, since the `index` method returns all the attributes of each node, we don't really need the `show` method because the client application would already have that data. We don't use the `show`, `new`, and `edit` methods for all three controllers, so we can delete them.

For the positions controller, we won't update a position; we will simply buy new stock and sell existing stock, meaning we are not using the `update` method. We also differentiate buying new stock and buying existing stock, because for existing stock, we know the ID of the position and find the object using an active record search. But, for a new stock position, we pass the stock ticker and we create the new position, which may not save and validate if the ticker is invalid. Therefore, to support these two different usage patterns, we decided to use two different actions: we use the `create` action for existing stock, and we add the custom `buy` verb to the positions controller to buy new stock.

The movements controller doesn't enable any updates since movements are generated when buying and selling positions, so only the `index` method is significant. Providing such a mapping table of the verbs serves as a good overview of the work you will do next. First, you can remove all unused methods. As you already implemented the index methods earlier in the chapter, we are left with seven methods, three for the accounts controller and four for the positions controller. Let's dive into it. For the accounts controller, in the `create`, `update`, and `destroy` methods, we simply remove the `respond_to` blocks and keep only the XML rendering.

```
class AccountsController < ApplicationController
      def create
        @account = Account.new(params[:account])
        if @account.save
          render :xml => @account, :status => :created, :location => @account
        else
          render :xml => @account.errors, :status => :unprocessable_entity
        end
      end

      def update
        @account = Account.find(params[:id])
        if @account.update_attributes(params[:account])
          head :ok
        else
          render :xml => @account.errors, :status => :unprocessable_entity
        end
      end
```

```
    def destroy
      @account = Account.find(params[:id])
      @account.destroy
      head :ok
    end
end
```

We saw earlier that the positions controller `index` method was relying on the `@account` variable set by the `get_account before_filter` to only access positions for the specified account. To enforce the scoping to a given account, the remaining methods of the positions controller will also use the `@account` active record to issue the find instead of directly using the `Position.find` method. Let's go ahead and update the `create` and `destroy` methods and add a `buy` method, as follows:

```
class PositionsController < ApplicationController
 def create
   @position = @account.positions.find(params[:position][:id])
   if @position.buy(params[:position][:quantity].to_i)
     render :xml => @position, :status => :created,
            :location => [@account, @position]
   else
     render :xml => @position.errors, :status => :unprocessable_entity
   end
 end

 def destroy
   @position = @account.positions.find(params[:position][:id])
   if @position.sell(params[:position][:quantity].to_i)
     render :xml => @position, :status => :created,
            :location => [@account, @position]
   else
     render :xml => @position.errors, :status => :unprocessable_entity
   end
 end

  def buy
   @position = @account.buy(params[:position][:ticker],
                            params[:position][:quantity].to_i)
   if @position.errors.empty?
     head :ok
   else
     render :xml => @position.errors, :status => :unprocessable_entity
   end
end
```

For the `buy` method, we simply use the ticker and invoke the `buy` method from the account active record:

```
class Account < ActiveRecord::Base
  has_many :positions, :dependent => :destroy
  def buy(ticker, quantity)
    ticker.upcase!
    position = positions.find_or_initialize_by_ticker(ticker)
    position.buy(quantity)
    position.save
    position
  end
end
```

The `Account#buy` method in turn calls the position `buy` method, which in turn creates a movement for the `buy` operation.

```
 class Position < ActiveRecord::Base
  belongs_to :account
  has_many :movements, :dependent => :destroy

 def buy(quantity)
    self.quantity ||= 0
    self.quantity = self.quantity + quantity;
    movements.build(:quantity => quantity, :price => quote.lastTrade,
                    :operation => 'buy')
    save
  end
end
```

Now let's extend the position active record to add a validation that will be triggered when saving the position. The first validation we add is the following:

```
validates_uniqueness_of :ticker, :scope => :account_id
```

This check simply ensures that one account cannot have more than one position with the same name. We verify that the ticker really exists by using the yahoofinance gem. Install it first:

```
$ sudo gem install yahoofinance
```

To make this gem available to our application we can create the following Rails initializer under config/initializers/yahoofinance.rb that requires the gem:

```
require 'yahoofinance'
```

That's it. Now we can write a `before_validation_on_create` handler that will load the given stock information from Yahoo Finance, and then we add a validation for the name of the stock, which is set by the handler only if the stock exists.

```
class Position < ActiveRecord::Base
  validates_uniqueness_of :ticker, :scope => :account_id
  validates_presence_of :name,
                        :message => "Stock not found on Yahoo Finance."
  before_validation_on_create :update_stock_information

      protected

      def quote
        @quote ||= YahooFinance::get_standard_quotes(ticker)[ticker]
      end

      def update_stock_information
          self.name = @quote.name if quote.valid?
        end
      end
end
```

When referring to the `quote` method, the instance variable `@quote` is returned if it exists, or if it doesn't exist, the stock information is retrieved from Yahoo Finance using the class provided by this gem:

```
YahooFinance::get_standard_quotes(ticker)
```

The `get_standard_quotes` method can take one or several comma-separated stock symbols as a parameter, and it returns a hash, with the keys being the ticker and the values being a StandardQuote, a class from the YahooFinance module that contains financial information related to the ticker, such as the name, the last trading price, and so on. If the ticker doesn't exist, then the name of the stock is not set and the save of the position doesn't validate.

The `sell` method of the positions controller is similar to the `buy` method, but less complex. Let's take a look:

```
 class Position < ActiveRecord::Base
  def sell(quantity)
    self.quantity = self.quantity - quantity
    movements.build(:quantity => quantity, :price => quote.lastTrade,
                    :operation => 'sell')
    save
  end
end
```

Similar to the `buy` method, the `sell` method updates the quantity and creates a `sell` movement, recording the price of the stock when the operation occurs. There is one last thing: we need to add the custom `buy` verb to our routes. Do this by adding the `:collection` parameter to the positions resource.

```
ActionController::Routing::Routes.draw do |map|

   map.resources :accounts do |account|
     account.resources :positions, :collection => {:buy => :post} do |position|
```

```
    position.resources :movements
  end
end
```

This indicates that no ID for the position is specified when creating the URL, thus invoking the `buy` verb on the positions collection. The URL would look something like this:

```
/accounts/1/positions/buy
```

If you wanted to add a custom verb that applies not only to the collection of the positions, but also to a specific position, thus requiring the position ID in the URL, you could have used the `:member` parameter to the positions resource.

Our application starts to be functional. By now, you certainly did a migration and started playing with your active records from the console. If not, play around a little, then keep reading because we are about to start the Flex part of our application.

Accessing Our RESTful Application with Flex

In this section, you will build a simple Flex application that will enable a user to create and maintain accounts, to buy and sell stock for those accounts, and to view the movements of a given stock. The application will look like what is shown in Figure 3.1.

The application has three grids. The ones for Accounts and Positions are editable, so they can be used to directly update the name of an account or to enter the stock ticker and quantity to buy. Let's create the application in three steps: first the accounts part, then the positions, and finally, the movements.

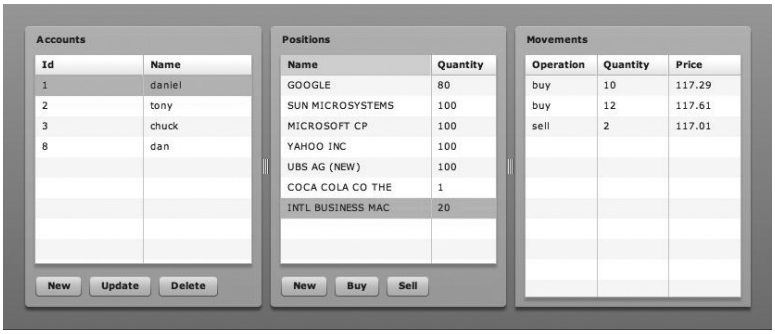

Figure 3.1 The application interface.

Accounts

Accessing a RESTful Rails resource from Flex consists of creating one `HTTPService` component for each of the actions the resource supports. For the account resource, the Rails application exposes the four URLs shown in Table 3.5.

Table 3.5 **The Four URLs Exposed by the Rails App for the Account Resource**

Action	Verb	Url
index	GET	/accounts
create	POST	/accounts
update	PUT	/accounts/:id
delete	DELETE	/accounts/:id

Before we create the service components, let's declare a bindable variable to hold the data the server returns.

```
[Bindable] private var accounts:XML;
```

And let's declare the data grid that will display this list of accounts.

```
<mx:DataGrid  id="accountsGrid"
    dataProvider="{accounts.account}"
    editable="true">
  <mx:columns>
    <mx:DataGridColumn headerText="Id" dataField="id" editable="false"/>
    <mx:DataGridColumn headerText="Name" dataField="name" />
  </mx:columns>
</mx:DataGrid>
```

We make the grid editable except for the first `id` column. We will use this later when adding the create and update functionality. Note also that we set the `dataProvider` to `accounts.account`. The `accounts` variable will contain the following XML, as returned by the Rails accounts controller index action:

```
<?xml version="1.0" encoding="UTF-8"?>
<accounts type="array">
  <account>
    <created-at type="datetime">2008-06-13T13:56:04Z</created-at>
    <id type="integer">1</id>
    <name>daniel</name>
    <updated-at type="datetime">2008-06-13T13:56:04Z</updated-at>
  </account>
  <account>
    <created-at type="datetime">2008-06-13T14:01:41Z</created-at>
    <id type="integer">2</id>
    <name>tony</name>
    <updated-at type="datetime">2008-06-14T02:19:46Z</updated-at>
  </account>
</accounts>
```

And specifying `accounts.account` as the data provider of the grid returns an XMLList containing all the specific accounts that can be loaded directly in the data grid

and referred to directly in the data grid columns. The first column will display the IDs of the accounts and the second column the names.

Before declaring all our services we can define the following constant in ActionScript to be used as the root context for all the URLs.

```
private const CONTEXT_URL:String = "http://localhost:3000";
```

To retrieve the account list, we need to declare the following `index` service:

```
<mx:HTTPService id="accountsIndex" url="{CONTEXT_URL}/accounts"
    resultFormat="e4x"
    result="accounts=event.result as XML"/>
```

In the `result` handler, we assign the result to the `accounts` variable we declared above. Again, we need to specify the `resultFormat` on the `HTTPService` instance, which in the case of `"e4x"` ensures that the returned XML string is transformed to an XML object.

Next we can declare the `create` service, as follows.

```
<mx:HTTPService id="accountsCreate"  url="{CONTEXT_URL}/accounts"
    method="POST" resultFormat="e4x" contentType="application/xml"
    result="accountsIndex.send()"  />
```

We set the HTTP verb using the `method` attribute of the `HTTPService` component to match the verb Rails expects for this operation. We also need to set the `contentType` to `"application/xml"` to enable us to pass XML data to the `send` method and to let Rails know that this is an XML-formatted request. Also notice that we issue an `index` request in the result handler. This request enables a reload of the account list with the latest account information upon a successful `create`, and thus, by reloading the data in the grid, we would now have the ID of the account that was just created.

For the update service, we need to include in the URL which account ID to update. We use the grid-selected item to get that ID. Since the name column is editable, you can click in the grid, change the name, and press the update button (which we will code shortly). Also note that, in this case, we don't reload the index list because we already have the ID and the new name in the list.

```
<mx:HTTPService id="accountsUpdate"
 url="{CONTEXT_URL}/accounts/{accountsGrid.selectedItem.id}?_method=put"
    method="POST" resultFormat="e4x"  contentType="application/xml"  />
```

As explained in the previous chapter, the Flash Player doesn't support either the `delete` or `put` verbs, but Rails can instead take an additional request parameter named `_method`. Also note that we reload the account list after a successful `delete` request, which provides the effect of removing the account from the list. We could have simply removed the account from the account list without doing a server round trip.

```
<mx:HTTPService id="accountsDelete"
    url="{CONTEXT_URL}/accounts/{accountsGrid.selectedItem.id}"
    method="POST" resultFormat="e4x" contentType="application/xml"
    result="accountsIndex.send()" >
  <mx:request>
    <_method>delete</_method>
  </mx:request>
</mx:HTTPService>
```

When the Flex application starts, we want to automatically load the account list. Implement this by adding the `applicationComplete` handler to the application declaration, and issue the send request to the account `index` service.

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="vertical"
  applicationComplete="accountsIndex.send()">
```

Now when your application starts, the account index request is made to the server, and the result is assigned to the `accounts` variable, which is bindable, meaning the grid reloads automatically when the assignment occurs. Next we'll add the create, update, and delete functionality for the account list.

What we don't show in these examples is the overall layout of the application. But, in short, we have a horizontal divide box that contains three panels, one for each resource:

```
<mx:HDividedBox>
  <mx:Panel title="Accounts" />
  <mx:Panel title="Positions" />
  <mx:Panel title="Movements" />
</mx:HDividedBox>
```

Each panel contains the grid and a control bar that has several buttons for the accounts and positions panels:

```
<mx:Panel>
  <mx:DataGrid />
   <mx:ControlBar>
      <mx:Button />
      <mx:Button />
   </mx:ControlBar>
</mx:Panel>
```

More frequently than using an editable data grid, your application may have data entry forms that show the detail of the selected record or a new entry form when adding data. This is also how a typical Rails application generates HTML. In our example, we are using the editable grid to achieve the same functionality, but in the end, which approach you use will depend on your application requirements.

To add a new account, we need the `New` button.

```
<mx:Button label="New"
  click="addNewAccount()"
  enabled="{accountsGrid.dataProvider!=null}" />
```

This button invokes the `addNewAccount` method, which simply adds a new XML node to the XML accounts list. There is no server call happening yet.

```
private function addNewAccount():void {
  accounts.appendChild(<account><id></id><name>new name</name></account>);
}
```

The user can then change the name of the account directly in the grid and use the following `Create` button to send the new account information to the Rails application:

```
<mx:Button
  label="{accountsGrid.selectedItem.id==''?'Create':'Update'}"
  click="accountsGrid.selectedItem.id=='' ?
          accountsCreate.send(accountsGrid.selectedItem) :
          accountsUpdate.send(accountsGrid.selectedItem)" />
```

If the selected item has no ID, it's a new record and the label of the button is set to `Create`, and the click action triggers the `accountsCreate` service. If an ID exists, the button transforms into an `Update` button, which uses the `accountsUpdate` service. Now we just need to add the delete functionality. If the selected item has an ID, in other words, if it exists on the server, the click handler invokes the `accountsDelete` service, which deletes the current record. We also support the scenario where you click the new button but want to cancel the creation, and the account list is then just reloaded from the server in this case.

```
<mx:Button label="Delete"
  click="accountsGrid.selectedItem.id=='' ?
      accountsIndex.send() : accountsDelete.send()"
  enabled="{accountsGrid.selectedItem!=null}" />
```

In the end, very little code supports the create, update, and delete functionality. In a real application, you may want to move some of the logic we added directly to the button's click handlers to a controller class, but the underlying principles and calls are the same.

Positions

Implementing the positions functionality is very similar to implementing an account grid and services, with just a few differences. For instance, the position resource is a nested resource, so make sure the URL is set to retrieve the positions for the selected account. Another difference is that the positions resource has the custom `buy` action. And, finally, when a new account is selected in the accounts grid, we want to automatically retrieve the positions. Let's get started.

As for the accounts, we declare a variable to hold the list of positions retrieved from the server:

```
[Bindable] private var positions:XML;
```

Next let's create the grid and bind it to the positions. Again, we use an E4X instruction to get an XMLList containing each position. We also make the grid editable, although the `name` column is only editable when the position has not yet been saved to the server. This can be verified by the fact that no ID is assigned, which enables us to use the name column to specify the ticker to buy.

```
<mx:DataGrid id="positionsGrid" width="100%" height="100%"
       dataProvider="{positions.position}"
       editable="true">
  <mx:columns>
    <mx:DataGridColumn headerText="Name" dataField="name"
      editable="{positionsGrid.selectedItem.id==''}"/>
    <mx:DataGridColumn headerText="Quantity" dataField="quantity"/>
  </mx:columns>
</mx:DataGrid>
```

Now you can add the following buttons to a control bar below the data grid.

```
<mx:Button label="New"
  click="addNewPosition()"
  enabled="{positionsGrid.dataProvider!=null}" />

<mx:Button label="{XML(positionsGrid.selectedItem).id==''?'Buy New':'Buy'}"
  click="buyPosition()" />

<mx:Button label="Sell"
  click="sellPosition()"
  enabled="{XML(positionsGrid.selectedItem).id!=''}" />
```

Each of the New, Buy, and Sell buttons will invoke a corresponding function that we will implement just after creating the services. Now, as we explained earlier, you must build the create service to buy an existing stock, the delete service to sell stock, the buy service to buy a new stock, and, of course, the index service to retrieve all the positions. These services will be mapped to the URLs shown in Table 3.6.

Table 3.6 **Service URL Mappings**

Action	Verb	Url
index	GET	/accounts/:account_id/positions
create	POST	/accounts/:account_id/positions
delete	DELETE	/accounts/:account_id/positions/:id
buy	POST	/accounts/:account_id/positions/buy

Also notice that for the nested resource, the prefix is always the same. So let's assume we have selected the account number 2. The prefix for the positions would be `/accounts/2`, and the complete URL to list all the positions for that account would be `/accounts/2/positions`. Another example for the same account, but this time to sell the position with an ID of 3, would go like this: the URL will be `/accounts/2/positions/3`. Flex provides several approaches to assembling the URL with the proper account ID, and here we simply create a string in MXML using the `mx:String` tag and bind that string to the selected item of the accounts data grid.

```
<mx:String
id="positionsPrefix">accounts/{accountsGrid.selectedItem.id}</mx:String>
```

We named this string `positionsPrefix` and will use it in all the URLs for the positions resource. Now each time we issue the `send` command for a service, the selected account ID is automatically part of the URL. Create the `index` service as follows:

```
<mx:HTTPService id="positionsIndex"
    url="{CONTEXT_URL}/{positionsPrefix}/positions"
    resultFormat="e4x"
    result="positions=event.result as XML"/>
```

Note the binding to the `positionsPrefix` string in the URL. The result of this service sets the `positions` variable we declared earlier, and as this variable is bound to `positionsGrid`, the grid reloads automatically.

The create and the buy services are declared with the same parameters, with only the URLs being different. Both are used to buy stock, but in the first situation, you need to pass the ID of the position, and in the second, the ticker of the stock you want to buy. Now this could have been implemented differently, and we could have used only one action and passed different parameters to differentiate the two situations, or we could even have checked whether the ID is a string and assumed you wanted to buy new stock.

```
<mx:HTTPService id="positionsCreate"
    url="{CONTEXT_URL}/{positionsPrefix}/positions"
    method="POST" resultFormat="e4x" contentType="application/xml"
    result="positionsIndex.send()"  />

<mx:HTTPService id="positionsBuy"
    url="{CONTEXT_URL}/{positionsPrefix}/positions/buy"
    method="POST" resultFormat="e4x" contentType="application/xml"
    result="positionsIndex.send()"  />
```

Now the `delete` service is used to sell stock, so it differs from the accounts delete service in the way that we will need to pass additional information parameters to the send call. So you cannot just define the required `_method` request parameter using the `mx:request` attribute of the service declaration, as it would be ignored when issuing the `send` call with parameters. We can, however, stick the `_method` parameter at the end of URL.

```
<mx:HTTPService id="positionsDelete"
  url="{CONTEXT_URL}/{positionsPrefix}/positions/
     {positionsGrid.selectedItem.id}?_method=delete"
    method="POST" resultFormat="e4x" contentType="application/xml"
    result="positionsIndex.send()">
```

You created the New, Buy, and Sell buttons to invoke respectively the `addNewPosition, buyPosition`, and `sellPosition` functions when clicked. Now that we have the services declared, let's code these functions. The `addNewPosition` function simply adds a `<position />` XML node to the `positions` variable which, via data binding, adds a row to the positions grid where you now can enter the ticker name of the stock you desire to buy and specify the quantity.

```
private function addNewPosition():void {
  positions.appendChild(<position><id/><name>enter ticker</name></position>);
}
```

The `buyPosition` function determines if you are buying new or existing stock by checking the ID of the currently selected position. In the case of a new stock, we used the name column to accept the ticker, but the Rails controller expects a `:ticker` parameter; therefore, we copy the XML and add a `ticker` element to it. For existing stock, we send the information entered in the grid.

```
private function buyPosition():void {
  if (positionsGrid.selectedItem.id=='') {
    var newStock:XML = (positionsGrid.selectedItem as XML).copy();
    newStock.ticker = <ticker>{newStock.name.toString()}</ticker>;
    positionsBuy.send(newStock)
  } else {
    positionsCreate.send(positionsGrid.selectedItem)
  }
}
```

To sell a position, we pass the `id` and `ticker` to the `send` call.

```
private function sellPosition():void {
  var position:XML =
  <position>
    <id>{positionsGrid.selectedItem.id.toString()}</id>
    <ticker>{positionsGrid.selectedItem.ticker.toString()}</ticker>
    <quantity>{positionsGrid.selectedItem.quantity.toString()}</quantity>
  </position>
  positionsDelete.send(position);
}
```

Next, we want the positions grid to reload when the account is changed. We can do this by adding a change handler to the accounts data grid. Here you may just want to clear out the `positions` and `movements` variables before performing the call to avoid

showing for the duration of the remote call the positions of the previously selected account. Then, if it's not a new account, you can invoke `send` on the `positionsIndex`.

```
<mx:DataGrid  id="accountsGrid" width="100%" height="100%"
    dataProvider="{accounts.account}"
    change="positions=movements=null;
        if (event.target.selectedItem.id!='') positionsIndex.send()"
    editable="true">
```

Movements

The movements are much simpler than accounts and positions, since we just want to display the list of movements for the selected position of the selected account. So, for the `index` service, you can simply bind the selected account and selected position to the URL as follows:

```
<mx:HTTPService id="movementsIndex"
   url="{CONTEXT_URL}/accounts/
        {accountsGrid.selectedItem.id}/positions/
        {positionsGrid.selectedItem.id}/movements"
   resultFormat="e4x"
   result="movements=event.result as XML"/>
```

You also need to declare the following variable to keep the results

```
[Bindable] private var movements:XML;
```

Then you just need to declare the following grid, which is bound to the `movements` variable. This time, we don't specify that the grid is editable because the movements are read only.

```
<mx:DataGrid id="movementsGrid" width="100%" height="100%"
       dataProvider="{movements.movement}">
  <mx:columns>
    <mx:DataGridColumn headerText="Operation" dataField="operation"/>
    <mx:DataGridColumn headerText="Quantity" dataField="quantity"/>
    <mx:DataGridColumn headerText="Price" dataField="price"/>
  </mx:columns>
</mx:DataGrid>
```

When selecting a new account, we simply want to clear out the movements grid. So we need to add this to the accounts grid change handler:

```
<mx:DataGrid  id="accountsGrid" width="100%" height="100%"
    dataProvider="{accounts.account}"
    change="positions=movements=null; if (event.target.selectedItem.id!='')
positionsIndex.send()"
    editable="true">
```

And, finally, when selecting a new position, we want to request the movements for that position. We can achieve this with the following change handler on the positions grid. Again, we clear the movements before issuing the call:

```
<mx:DataGrid id="positionsGrid" width="100%" height="100%"
       dataProvider="{positions.position}"
       change="movements=null;
             if (accountsGrid.selectedItem.id!='' &amp;&amp;
                 positionsGrid.selectedItem.id!='') movementsIndex.send()"
       editable="true">
```

Note that when placing code directly in the MXML declarations, you cannot use the ampersand character directly due to the parsing rules of XML on which MXML is based; otherwise, you get a compiler error. So, in the above code, we need to replace the logical AND expression `&&` with its XML-friendly equivalent: `&&`. This issue is avoided when writing code inside an `mx:Script` tag due to the use of the XML CDATA construct, which indicates to the XML parser that anything between the CDATA start indicator and end indicator is not XML, therefore allowing the use of the ampersand character.

Et voila! You can now create and maintain accounts, buy and sell stock, and see the movements for these positions. In this Flex application, we directly linked the services to the data grid and directly added logic in the different event handlers, and this makes explaining the code easier as you have everything in one file. This is definitely not a best practice, and for a real application, you may want to consider using an MVC framework to decouple the code and make the application more modular.

Summary

We covered lots of ground in this chapter. You created a RESTful Rails application with three nested resources and you created a Flex application enabling you to maintain these resources. In the end, all this functionality doesn't require much code. Hopefully, this guidance will provide a good starting point for creating your own RESTful Flex with Rails applications.