



Tony Hillerson
Daniel Wanja

Foreword by
Matt Chotin, Senior Product Manager, Adobe Systems, Inc.

Flex on Rails

Building Rich Internet Applications
with Adobe Flex 3 and Rails 2

Developer's Library



Flex on Rails

Copyright © 2009 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.
Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

ISBN-13: 978-0-321-54337-0

ISBN-10: 0-321-54337-8

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

First Printing, December 2008

Trademarks

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Warning and Disclaimer

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Bulk Sales

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Hillerson, Tony.

Flex on Rails : building rich internet applications with
Adobe Flex 3 and Rails 2 / Tony Hillerson, Daniel Wanja.

p. cm.

Includes index.

ISBN 978-0-321-54337-0 (pbk. : alk. paper)

1. Flex (Computer file)
2. Ruby on rails (Electronic resource)
3. Internet programming.
4. Application software—Development.
5. Web site development—Computer programs. I. Wanja, Daniel. II. Title.

QA76.625.H57 2008
006.7'6—dc22

Editor-in-Chief
Karen Gettman

**Senior Acquisitions
Editor**
Chuck Toporek

**Senior
Development
Editor**
Chris Zahn

Managing Editor
John Fuller

**Full-Service
Production
Manager**
Julie B. Nahil

Editorial Assistant
Romny French

Project Editor
diacriTech LLC

Book Designer
Gary Adair

Compositor
diacriTech LLC

Foreword

This is the decade of the web application: business logic and data storage provided on a centralized server or servers while end users interact with the user interface on their own machines. In 2004, Macromedia (now Adobe) took the world of web applications further with the introduction of Flex. Flex is a set of languages, frameworks, and tools for building Rich Internet Applications (RIAs)—the evolution of the web application. By using the ubiquitous Flash Player, installed on over 98 percent of Internet-connected PCs, application developers can deliver a top-notch user experience while still leveraging the compelling distribution and update model of the web application.

The latest version of Flex, Adobe Flex 3, was released in early 2008 to great fanfare. This version added more capabilities to the IDE used by most Flex developers, Adobe Flex Builder, including a profiler, enhanced design view, and refactoring support. We also greatly enhanced the data visualization capabilities of Flex by improving our charting controls and adding hierarchical data grids and simple OLAP components. This release has been our most successful to date, with the number of developers adopting Flex growing by over 130 percent year over year.

One thing that encouraged this kind of adoption is that Flex 3 was released as open source under the Mozilla Public License. As developers investigate the technologies they'll use for future software development, a criterion has come to the forefront to have visibility into future plans and potentially influence them. Adobe has acknowledged this and has been embracing open source and standards heavily over the last few years, including the standardization of PDF, opening of the SWF file format, and, most importantly, the open source release of Flex as well as the data connectivity solution, BlazeDS.

Flex is at its best when displaying and allowing the modification of data, and BlazeDS provides a number of ways for moving that data from server to client and back again. BlazeDS includes the specification to the Action Message Format (AMF), which allows for more efficient data transfer than any other standard technique on the web today. While most of the runtime code for BlazeDS is written in Java, the available specifications allow any back-end technology to leverage AMF for data transfer. And Ruby on Rails is one of the back-ends that supports AMF through a number of solutions, including RubyAMF and WebORB.

At the same time that Flex has been establishing itself as the best way to produce RIAs, Ruby on Rails has been growing quickly as a top choice for building web applications. With the ability to rapidly create databases, value objects, and business logic, Ruby on Rails is a great back-end choice for a Flex front end. The ease of wiring data together with Rails, combined with the compelling visualizations and user interface elements of Flex, make the two a perfect pair.

Flex on Rails is a great resource for those of you looking to use these two great technologies. Tony Hillerson and Daniel Wanja walk through all of the elements you'll need to know to build an application using Flex and Rails. While this book should not generally serve as a pure introduction to either technology, you do not need extensive

experience to benefit from it. The approach is straightforward: get up and running with the two sides talking to each other, then learn the techniques for leveraging the best of both systems. The first section provides broader introductions to areas like data transfer, testing, and using features of the Flex framework and related projects to your advantage. The second section includes examples of some of the more advanced things you'll need to know as you build a robust application. There's no question you're going to be a better Flex and Rails developer when you're done reading this book.

There may be no better time in software development than now. With constant improvements in the tools and technologies for building applications, you've chosen two that are at the forefront of the revolution. I think I speak for all of us at Adobe when I say that we're excited to see what you can do with them.

*Matt Chotin
Senior Product Manager
Adobe Systems, Inc.*

Preface

In 2004 yet another framework for making websites appeared. It was called Ruby on Rails, and when web and enterprise developers who had never heard of Ruby before started to work with it, they discovered that it wasn't like all the others. Why?

Rails offers the promise of fewer configuration files, less boilerplate code, less red tape, and, most of all, the promise of having fun again while programming. Rails was designed to make development more about getting common development tasks done by following conventions, not offering endless flexibility for the remote possibility of connecting up with any number of hypothetical legacy back-ends sometime in the future. By taking away unnecessary choices and offering simple solutions for common problems, Rails lets developers focus on writing applications, and developers have paid Rails back in accolades and adoption.

Around the same time as Rails was emerging, Macromedia (later bought by Adobe) was bringing a product codenamed “Royale” to market as Flex—a server-side Flash compiler. Flash had been experiencing a groundswell of developers wanting to build rich interfaces for web applications, not just animations or “Punch the Monkey” ads. Flash developers loved that they could build interfaces that would be extremely hard, if not impossible, to produce in HTML and JavaScript, but the Flash IDE was more suited to timeline animations. Flex changed all that by offering an easy-to-understand XML language for developing Flash interfaces, as well as a component set that made it dead simple to build applications. Flex 2 made things even better by taking the Flex compiler off the server so that there was no requirement of building and deploying Flash movies (SWFs) using the Flex framework and compiler. Flex 3 is one step better by being open source!

The Flex language has come a long way too. Bruce Eckel, the author of such books as *Thinking in C++, Second Edition* (Prentice Hall, 2000) and *Thinking in Java, Fourth Edition* (Prentice Hall, 2006), said it best when he said, “Flex is a DSL for graphics, multimedia, and UIs” (www.artima.com/weblogs/viewpost.jsp?thread=193593). DSL, or Domain Specific Language, is a language with terms that match up well with a certain problem domain, and the term DSL should resonate with Rails developers. Flex, more than HTML with JavaScript, is a language for building rich interfaces quickly and easily.

Flex and Rails developers haven't crossed paths that often, but more and more as the word gets out about each technology, developers want to know what these two are about. Having worked for years with both Flex and Rails, we believe that both have something powerful to offer developers who want to build desktop-like functionality into web apps quickly, in a team environment, with the ability to be agile and react to the ever-changing requirements of building and delivering an application to the web.

Audience for This Book

Chances are you're reading this introduction to figure out if this book is for you. Have a look at these "stories" and see if any of them sound familiar.

Rick—Rails Guy

Background	Worked with Java for years: Struts, EJB, Spring, all that. Then discovered the joy of working with Rails and hasn't looked back.
Overheard	"Edge Rails? Is there any other kind?"
Goal	"HTML/CSS/JavaScript is fine for simple interfaces, and I don't mind how cruffy it is as long as I don't have to deal with it. I'd be interested in finding out how to build richer interfaces, though—interfaces that don't just look better but also give users better tools and an overall better experience."

Jill—Java to Flex Convert

Background	Worked with Swing and understands desktop applications. Got into "The Web" and did lots of J2EE. Loves Flex for the ability to build desktop-like functionality on the web, usually in front of J2EE back-ends.
Overheard	"GridBagLayout? What were they thinking?"
Goal	"I love working with Flex, but I'm getting a little tired of all the work it takes to set up the back-end with Java. Enterprise software is great when I need the flexibility of all that configuration, but what about when I have a straightforward model and I just want it to work? There's got to be an easier way."

Pete—Flash/PHP guy

Background	Has done lots of design work and building of interfaces with Flash. Has built websites with PHP and also integrated PHP back-ends with Flash.
Overheard	"I don't [skip intro], I [make intro]"
Goal	"I know what I can do in Flash, and I know how to feed Flash UIs with data from PHP. I'd really like to see what this Flex thing is all about though, since I'm not always building timeline animations. I've also heard a lot about Rails and how easy it is to get a back-end up and running."

If any of these stories sound a bit like you, then this book is for you. Rails and Flex have both revolutionized the way we develop web applications on both the front- and the back-ends.

Developers who have found Rails and left the world of enterprise framework stacks behind would very rarely willingly go back to the slow development cycles and bloated boilerplate code they had to endure. Flex developers have found the declarative XML language much cleaner and less cruffy than HTML and able to do things like 3D and video that would be impossible without Flex.

Of course, there are the normal disclaimers. Rails isn't for every project and neither is Flex. David Heinemeier Hansson wrote about PHP in a blog post (www.loudthinking.com/posts/23-the-immediacy-of-php):

I've been writing a little bit of PHP again today. That platform has really received an unfair reputation. For the small things I've been using it for lately, it's absolutely perfect. . . .

For the small chores, being quick and effective matters far more than long-term maintenance concerns. Or how pretty the code is. PHP scales down like no other package for the web and it deserves more credit for tackling that scope.

And the same goes for Flex. HTML doesn't need to be compiled and needs no special tools besides the ubiquitous browser to view it, whereas Flex needs a compiler and the Flash Player. However, when you find yourself with a medium to large web project with a database back-end, working with a team on a set of complex forms, rich visual interactions, video integration, 3D features, or a very large set of views, then Flex and Rails make a great choice.

The Flex and Rails story has a lot to do with discovering the integration capabilities and learning the ins and outs of making them talk to each other. We've gone a bit beyond that, though, and tried to assemble enough information about the next steps, common tasks, and how-to's that developers will want to know about sooner or later.

We thought that, for the most part, Flex developers who have never used Rails will want to learn some of the features that they'll run into during integration, but also during daily work. Likewise, developers who are already familiar with Rails will want to know a bit more about how Flex works than just consuming Rails' XML services.

Note

This book is not an introduction to Rails nor to Flex. If you haven't used at least one of these technologies before, you may find yourself a little lost. If you've used or are pretty familiar with one technology and haven't been exposed to the other, then this book is a great companion to reading a primer about the other, either first or at the same time.

What's in This Book?

The book is designed to take you through the process of learning about integrating and beginning to explore Flex and Rails. In Part I, *Flex and Rails Essentials*, we cover the core topics that you need to be familiar with in order to get your Flex and Rails projects going.

In Chapter 1, *Developing with Flex and Rails*, we talk about how to set up your environment. Then in Chapter 2, *Passing Data with XML*, and Chapter 3, *Flex with RESTful Services*, we'll show you how to integrate Flex and Rails using XML, both with a regular and with a RESTful Rails service.

Chapter 4, *Using Fluint to Test a Flex with Rails Application*, addresses testing. Chapter 5, *Passing Data with AMF*, discusses how to integrate Flex with RubyAMF, which uses the Flash-native AMF protocol.

Sooner or later you're going to want to learn how to debug your applications on the front- and back-ends to learn about what's going on or going wrong with them, so Chapter 6, *Debugging*, discusses debugging in both environments.

One of the great features of Flex is that it's easy to start making very powerful data visualization features available in your application, so Chapter 7, *Data Visualization*, introduces several pieces of that topic.

Chapter 8, *Flex MVC Frameworks*, talks a bit about a common topic in the Flex community—which frameworks to choose—and discusses two popular ones.

Chapter 9, *Performance and Optimization*, finishes the main section of the book with a discussion of how to get to the bottom of performance problems in both Flex and Rails and some tips about optimizing your applications.

Part II, *Cookbook Recipes*, was a lot of fun to bring to you. It's in the form of a cookbook, which contains many "recipes" or short discussions and how-to's about topics that you'll have questions about at some point. In Chapters 10 through 22, we cover everything from working with common source control systems and authenticating to pushing data and deploying Flex and Rails applications.

- Chapter 10: Source Control Flex and Rails Projects
- Chapter 11: Building Flex with Rake
- Chapter 12: Deploying Flex and Rails Applications
- Chapter 13: Read the Source!
- Chapter 14: Using Observers to Clean up Code
- Chapter 15: Authenticating
- Chapter 16: Reusing Commands with Prana Sequences
- Chapter 17: Hierarchical Data with RubyAMF
- Chapter 18: Advanced Data Grid and Awesome Nested Set
- Chapter 19: Runtime Flex Configuration with Prana
- Chapter 20: Server Push with Juggernaut
- Chapter 21: Communicating between Flex and JavaScript
- Chapter 22: File Upload

Let's Get Started

Now that you've decided to take the first steps to putting Flex on Rails, let's take a look at what you'll need to get your development environment in shape. Start with Chapter 1, *Developing with Flex and Rails*, to make sure you have all the pieces in place.

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
end
```

```

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

The screenshot shows a web application interface with three main panels. The 'Accounts' panel has a table with columns 'Id' and 'Name', containing rows for 'daniel', 'tony', 'chuck', and 'dan'. Below it are 'New', 'Update', and 'Delete' buttons. The 'Positions' panel has a table with columns 'Name' and 'Quantity', listing various stocks like 'GOOGLE', 'SUN MICROSYSTEMS', etc., with their respective quantities. Below it are 'New', 'Buy', and 'Sell' buttons. The 'Movements' panel has a table with columns 'Operation', 'Quantity', and 'Price', showing a 'buy' of 10 shares at 117.29, another 'buy' of 12 shares at 117.61, and a 'sell' of 2 shares at 117.01.

Accounts		Positions		Movements		
Id	Name	Name	Quantity	Operation	Quantity	Price
1	daniel	GOOGLE	80	buy	10	117.29
2	tony	SUN MICROSYSTEMS	100	buy	12	117.61
3	chuck	MICROSOFT CP	100	sell	2	117.01
8	dan	YAHOO INC.	100			
		UBS AG (NEW)	100			
		COCA COLA CO THE	1			
		INTL BUSINESS MAC	20			

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.

Index

Symbols and Numbers

& (Ampersand), MXML declarations and, 48

&& (Logical AND), 48

2D charts, 147

A

Access control, 251

Account resource, RESTful Rails resources, 39-43

Action Message Format. See AMF (Action Message Format)

Actions, RubyAMF, 98

ActionScript

AMF (Action Message Format) and, 85-86

compiling into SWF file, 240

E4X classes supported in ActionScript 3.0, 14

file upload and, 314

Flex performance and, 186

IoC library and, 289

`trace` statement for logging, 109

XML with Flex and, 11

ActionScript 3 (AS3) framework, 159

Active records

associations, 233, 237-238

lifecycle events and, 248-250

loggers and, 108

- Ruby and, 90
- validation (HTTP 400s), 26
- Add button, implementing and testing, 75**
- `addCallback` **method**,
externalInterface, **304–305**
- `AdvancedDataGrid`, **275–288**
 - adding CRUD support to Flex and Rails application, 282–288
 - creating Rails application that returns hierarchy of categories, 276–279
 - data visualization and, 144–147
 - manipulating hierarchical data with, 275, 279–282
 - overview of, 131
- Ajax, 303**
- AMF (Action Message Format)**. *See also* **RubyAMF**
 - benefits of, 86–87
 - Charles debugger and, 128
 - overview of, 85
- AMFPHP, 94**
- `&`, **48**
- Amperсанд (&), MXML declarations and, 48**
- app directory**
 - in contact manager example, 97–98, 100
 - overview of, 7
- Application structure, 6–8**
- Aptana Rad Rails**
 - debugging Rails applications, 116–117
 - installing editors, 5
- Assertion methods**
 - `TestCase` class, 59
 - testing Cairngorm-based applications, 75–78
- associations, active records, 233, 237–238**
- ASUnit, 50**

- attachment_fu plug-in**
 - installing, 315–316
 - using with `FileReference` and `URLLoader`, 313
- Authentication, 251–257**
 - installing `restful_authentication`, 251–257
 - overview of, 251
 - summary, 257
 - user authentication, 251
- awesome_nested_set plug-in**
 - manipulating hierarchical data, 275, 283, 286
 - moving hierarchical data from Rails to Flex, 267–273

B

- Benchmark script, Rails performance scripts, 209**
- Benchmarking, in Rails, 207–208**
- Binary format, AMF, 86**
- `BindingUtils`, Flex observers, **245–248**
- Blocks, Rake, 222**
- Blog entries, Rich Text Editor and, 246**
- Branches, in development, 218**
- Breakpoints, setting, 114**
- Breakpoints tab, Flex debugging, 113**
- Builds, Rake and, 221–225**

C

- Cairngorm, 154–158**
 - controller, 156
 - goals and concepts, 154–155
 - implementing, 169–176
 - model, 154–155
 - origin of, 154
 - reusing commands with Prana sequences, 259–265

- service layer, 157–158
- view, 156
- Cairngorm applications, testing**
 - implementing Cairngorm commands, 70–71
 - implementing `EditView` class, 78–79
 - implementing `ListView` class, 74–78
 - `NoteDelegate`, 63–64
 - `NoteTake` user interface, 60–62
 - overview of, 59–60
 - `ServiceLocator`, 62–63
 - test runner, 66–67
 - `testCreate`, 67
 - `testDelete`, 69–70
 - testing delegate’s CRUD actions, 65
 - testing `EditView` class, 79
 - testing `RetrieveNotesCommand`, 71–72
 - testing `SaveChangeCommand`, 72–74
 - `testList`, 65–66
 - `testUpdate`, 68–69
- call **method**, `externalInterface`, 304–305
- Call stacks, debugging and, 110–111**
- Capfile, 228–229**
- Capistrano gem, 228**
- Capistrano tool, for deploying applications, 227–231**
- Capture Performance Profile button, Flex Builder profiler, 190**
- CDATA constructs, XML, 48**
- Change events, 246**
- `ChangeWatcher`, **Flex observers, 245–248**
- Channels, Juggernaut support for, 297**
- Charles, 127–128**
- Charts**
 - HLOC (High Low Open Close) chart, 149–152
 - overview of, 131, 147
 - Pie charts, 147–148
- Children, attribute, 139**
- `ClassMappings`, **RubyAMF, 88–91**
- Clients, Juggernaut support for, 297**
- Code**
 - cleaning up with observers. *See* `Observers`
 - Flex performance and, 186
 - frameworks and, 153
- Command line debuggers**
 - debugging Flex with `fdb`, 117–123
 - debugging Rails with `ruby-debug`, 123–127
- Commands**
 - atomic, 259–260
 - Cairngorm, 155, 156
 - implementing Cairngorm commands, 70–71
 - Prana `EventSequences`, 261–265
 - PureMVC, 160
- Communication, debugging, 127–128**
- Compiler, Flex, 240–243**
- config directory, 7**
- Configuration**
 - configuring RubyAMF, 87–91
 - “Convention over Configuration” principle, 88
 - runtime configuration of Flex, 289–293
- Console debugger. *See* `fdb`**
- Constructors, implementing PureMVC, 179**
- Contact manager example, RubyAMF, 95–101**
- Content types, account resource and, 41**
- Contexts, Stuff entities, 163**

Controller methods, creating Rails application and, 33–35**Controllers**

- Cairngorm, 156, 172–173
- in contact manager example, 97, 100
- PureMVC, 159–160

Controllers, Rails

- in app directory, 7
- params hash, 89
- Rails fixture controller, 80–81
- RubyAMF, 93

Controls, debugger, 113**“Convention over Configuration” principle, Rails development, 88****Coverflow component (McCune), 224–225****create, read, update, and delete. See CRUD (create, read, update, and delete)****Cross domain policy files, development with Flex and, 8****crossjoin method, OLAP, 137****CRUD (create, read, update, and delete)**

- adding support to Flex and Rails application, 282–288
- implementing CRUD actions, 63–64
- Rails applications, 30
- ServiceLocator supporting, 62–63
- Stuff and, 163
- testing delegate’s CRUD actions, 65

Cygwin, 228

D

Data model complete use case, RubyAMF, 93**Data transfer**

- benefits of AMF, 86
- Data Transfer objects, 94

Data types, mapping between Flex and Rails, 21–24**Data visualization, 131–152**

- Advanced DataGrid and, 144–147
- charts, 147
- Flex mx.olap framework, 135–140
- HLOC (High Low Open Close) chart, 149–152
- OLAP and, 133–135
- OLAPCube applied to Stock Portfolio application, 140–144
- overview of, 131–133
- pie charts, 147–148
- summary, 152

Data warehouses, 134**DataGrid**

- advanced. *See* AdvancedDataGrid
- displaying list of accounts, 40
- function of, 144
- OLAP. *See* OLAPDataGrid
- position resource and, 44

db directory, 7, 96**Debug tab, Flex debugging, 112–113****Debuggers**

- controls, 113
- overview of, 110

Debugging, 105–129

- call stacks and stack frames and, 110–111
- communication, 127–128
- debuggers, 110
- debugging Flex with fdb, 117–123
- debugging Rails with ruby-debug, 123–127
- defined, 105
- Flex and logging, 108–109
- logging and, 106
- overview of, 105–106
- Rails, 116–117
- Rails logging, 106–108

scope and, 110
summary, 129

Debugging Flex
applying, 114–116
Breakpoints tab, 113
Debug tab options, 112–113
Expressions tab, 113–114
overview of, 111
Variables tab option, 113

Deferred execution, of code, 186

Delegates. See *also* `NoteDelegate`
Cairngorm service layer and, 157
implementing Cairngorm and, 171
levels of Flex unit tests, 50
testing delegate's CRUD actions, 65

Dependency Injection, IoC (Inversion of Control), 289–293

Deploying applications
Capistrano tool for, 227–231
overview of, 227
summary, 232

Design Patterns: Elements of Reusable Object-Oriented Software, 154

Development, branches and main line, 218

Development use case, RubyAMF, 93

Dimensions, multidimensional databases, 134

Directories
Cairngorm, 157–158
creating structure of Flex project, 52
Flex and Rails, 7–8
Flex SDK files, 239
gems, 236
ignoring files in Subversion, 215–217
PureMVC, 162–163

doc directory, 7

dpUnit. See **Fluint (Flex Unit and Integration)**

Drag-and-drop, moving categories in Flex, 287

“DRY” principle, Rails development, 88

E

E4X
XML in Flex and, 14–17
XML parsing and, 86

Eclipse, 5

ECMAScript for XML specification, 14

Editors, for working with Flex and Rails, 5–6

EditView class
implementing when testing
Cairngorm application, 78–79
`NoteTaker` application, 61–62

Error codes, Flash Player and, 26–28

Error handling, HTTP errors, 25

Event handlers, Flex messaging client, 299

Events/commands, Flex unit tests, 50

EventSequence, Prana, 261–265

Expressions tab, Flex debugging, 113–114

ExternalInterface, 304–309
applying, 305–309
`call` and `addCallback` methods, 304–305

F

Facade, PureMVC
implementing, 177–178
overview of, 159–160

fdb
debugging Flex, 117–123
as Flex console debugger, 109

Fibonacci visualizer example, 195–198

File upload, 313–320

creating Rails application and installing attachment_fu plug-in, 315–316

Flex FileReference class for uploading files, 316–318

Flex URLLoader class for uploading PNG files, 318–320

overview of, 313–315

summary, 320

FileReference class, for uploading files, 316–318

Fixtures, 79–84

Flex TestHelper, 81–82

overview of, 79

Rails fixture controller, 80–81

Flash Player, 41

benefits of AMF and, 86

error codes and, 26–28

file upload and, 314

Flex performance and, 185–186

HTML formatting, 246

security model, 303–304

Flash Player Content Debugger

memory profiling and, 187

overview of, 111

Flash Remoting, AMF, 85

flash.system.system's totalMemory **property, 187**

Flex

accessing RESTful applications with, 39–48

adding CRUD support to Flex applications, 282–288

BindingUtils and ChangeWatcher, 245–248

command line debugger, 117–123

communicating with JavaScript. *See* JavaScript/Flex communication

compiling with Rake, 221–225

deploying applications, 228–231

file uploads, 314

FileReference class for uploading files, 316–318

generated or compiled source, 240–243

logging and, 108–109

mapping data types with Rails, 21–24

moving XML from Flex to Rails, 19–21

MVC frameworks. *See* MVC frameworks

as open source, 233

performance. *See* Performance, Flex project structure, 52

RIAs (rich Internet applications) and, 303

RubyAMF used with, 94–95

runtime configuration with Prana, 289–293

sending hierarchical data from Rails to. *See* Nested sets

sending XML between Flex and Rails, 17–19

sending XML data to/from Rails, 24–25

source code, 238–240

URLLoader class for uploading PNG files, 318–320

XML in, 14–17

Flex-Ajax bridge, for JavaScript/Flex communication, 309–311**Flex Builder, 187**

benefit of using, 3

compiler, 240–243

creating new Flex project, 52

debugging and, 111–112, 115

debugging options, 106

- installing, 5–6
 - profiler, 187–194
 - Rake and, 221
 - reading Flex source, 238
 - Subclipse and, 218
 - trace statements and, 109
 - Flex Builder Professional**
 - advanced data grid component in, 275
 - charts, 147
 - data visualization framework, 131
 - flex mx.olap framework in, 133
 - Flex clients, creating messaging client, 299–301**
 - Flex Compiler, 240–243**
 - Flex Content Debugger, 115**
 - Flex Debugging Perspective, 112**
 - Flex mx.olap framework**
 - in Flex Builder Professional, 133
 - overview of, 135–140
 - Flex SDK**
 - directories, 239
 - installing, 4
 - Flex Unit and Integration. See Fluint (Flex Unit and Integration)**
 - FlexUnit, 50**
 - Fluint (Flex Unit and Integration)**
 - assertion methods, 59
 - compared with other Unit tests, 50–51
 - creating MXML component for invoking Rails application, 53–54
 - creating new Rails project, 57
 - creating structure of Flex project, 52
 - downloading, 52
 - overview of, 49
 - running a test, 57–59
 - setting up Flex test environment, 52–53
 - steps in, 51
 - summary, 83
 - testing Cairngorm-based applications. *See* Cairngorm applications, testing
 - testing queries to Rails server, 54–57
 - Frames, debugging and, 110–111**
 - Framework classes, 239–240**
 - Frameworks, 153. See also MVC frameworks**
-
- ## G
-
- Garbage Collector, Flex memory profiling, 194, 201**
 - gem environment **command, 235**
 - gem which **command, 235**
 - Gems**
 - directories, 236
 - initializing for Rails application, 37–38
 - installing Juggernaut Push Server via, 298
 - ruby-debug/ruby-debug-ide, 5
 - Ruby Gems, 4
 - SQLite Gem, 5
 - Generators, RubyAMF, 91–92**
 - getTimer **method, profiling techniques, 186**
 - Getting Things Done (GTD), 163**
 - Git, 217–219**
 - git-svn, 219**
 - .gitignore files, 217**
 - Graphs, memory usage, 198–199**
 - GTD (Getting Things Done), 163**
-
- ## H
-
- Hierarchical data**
 - AdvancedDataGrid for manipulating, 275
 - creating Rails application that return hierarchy of categories, 276–279

displaying in `AdvancedDataGrid`,
279–282

RubyAMF for moving from Rails to
Flex. *See* Nested sets

HLOC (High Low Open Close) chart,
149–152

HTML

Flash Player and, 246
methods for supporting HTML user
interface, 35

html-template directory, 8

HTTP, form-based uploads supported,
313–314

HTTP status codes, 25–28

Flash Player restrictions, 26–28
HTTP 200s (success indicator), 25–26
HTTP 400s (active record validation),
26
HTTP 500s (error indicator), 26
testing queries to Rails server, 56–57

HTTPService

asynchronous calls and, 55
controlling URL at runtime, 293
creating for each action resource
supports, 39
Flex connecting to Rails applications
via, 11
sending XML between Flex and
Rails, 17–21

I

Images, deploying Flex applications and,
228

index service, for retrieving account list, 41

Installing

editors, 5
Flex Builder, 5–6
Flex SDK, 4
RubyAMF, 87

Instant Messaging, 295

IoC (Inversion of Control), Prana, 289–293

J

J2EE, integrating Flex application with, 94

Java, AMF and, 85

JavaScript/Flex communication, 303–311

building example for, 304
`ExternalInterface`, 304–309
Flash Player security model and,
303–304
Flex-Ajax bridge, 309–311
overview of, 303
summary, 309–311

JavaScript libraries, 305–307

JSON (JavaScript Object Notation),
297–298, 300

Juggernaut Push Technology, 295–301

creating Flex messaging client,
299–301
creating Rails messaging application,
298–299
overview of, 295
push technology and, 295
Rails server sending messages to Flex
application in real time, 295–298
summary, 301

JUnit framework, 50

L

Lazily loading, 261

lib directory, 8

Libraries

JavaScript, 305–307
Prana, 290
RSLs (Runtime Shared Libraries),
228
Ruby Standard Library, 106

Lifecycle events, active records and, 248–250

Linux computers

- Capistrano and, 228
- Git and, 218–219
- Rails installation on, 4
- Ruby location on, 235

ListView class

- implementing when testing
 - Cairngorm application, 74–78
 - NoteTaker application, 60–61

Live Objects view, Flex memory profiling, 199–200, 202

LiveCycle Data Services, 94

log directory, 7

Logger class, Ruby Standard Library, 106

Logging

- Flex and, 108–109
- overview of, 106
- performance management and, 206–207
- Rails, 106–108

Logical AND (&&), 48

M

Mac computers

- Capistrano and, 228
- Git and, 219
- Rails installation on, 4
- Ruby installation on, 4
- Ruby location on, 235
- Subversion for, 218

Main line, in development, 218

Mapping data types, between Flex and Rails, 21–24

Measures, multidimensional databases, 134

Mediators, PureMVC, 161, 179, 181–182

Members, attribute, 139

Memory profiling, Flex, 194–206

- analyzing using memory snapshots, 201–206
- enabling memory profiling, 198
- experimenting with different
 - Fibonacci sequence lengths, 199–200
- Fibonacci visualizer example, 195–198
- Garbage Collector and, 194, 201
- Live Objects view, 199–200, 202
- memory snapshots and, 201–202
- memory usage graph, 198–199
- overview of, 194

Memory snapshots, Flex memory profiling, 201–206

Memory usage graph, Flex memory profiling, 198–199

Memory usage, profiling techniques, 186

Method local, scope and, 110

ModelLocator, 154–155

Models

- in app directory, 7
- Cairngorm, 154–155
- PureMVC, 160–161

Mongrel, 4–5

move_to_child_of method, 276–277

Multidimensional databases, 134

Multipart requests, 318

MVC frameworks, 153–183. See also

Cairngorm

- Cairngorm, high level view, 154–158
- Cairngorm implementation, 169–176
- creating own, 154
- Flex as MVC application, 49
- PureMVC, high level view, 159–163
- PureMVC implementation, 176–182
- Stuff, 163–164
- summary, 183
- what is a framework, 153

MXML

- ampersand (&) and, 48
- compiling, 9
- compiling into SWF file, 240
- creating MXML component for
 - invoking Rails application, 53–54

N

Nested resources, 43**Nested sets, 267–273**

- awesome_nested_set plug-in, 267–273
- overview of, 267
- summary, 273
- in XML format, 277–278

New Relic RPM (Rails Performance Management), 211–212**NoteDelegate**

- implementing CRUD actions, 63–64
- testCreate, 67
- testDelete, 69–70
- TestNoteDelegate, 65–66
- testUpdate, 68–69

NotesResource class

- creating MXML component for
 - invoking Rails application, 53–54
- TestNotesResource test case, 57–59

NoteTaker application

- overview of, 51
- test runner, 66–67
- user interface, 60–62

O

Observers, 245–250

- BindingUtils and ChangeWatcher
 - in Flex, 245–248
- BlogPost and, 249–250

- overview of, 245
- summary, 250

OLAP (Online Analytical Processing)

- applying to Stock Portfolio
 - application, 132–133
- defined, 133
- Flex mx.olap framework, 135–140
- OLAPCube applied to Stock Portfolio
 - application, 140–144
- overview of, 131, 133–135

OLAPCube

- applied to Stock Portfolio application, 140–144
- creating, 135–137
- overview of, 133

OLAPDataGrid

- overview of, 133
- sending OLAPQuery to, 135
- standard data grid compared with, 137

OLAPQuery

- creating, 135–137
- overview of, 133

OLAPResult, 137**OmniFocus, GTD model, 163****Online Analytical Processing. See OLAP (Online Analytical Processing)****Open source, 233–234****Optimization. See Performance**

P

params hash, Rails controllers, 89**Parsing**

- AMF, 86
- XML, 48

PeopleController, sending XML from Flex to Rails, 19–21**Performance, Flex, 185–206**

- Flex Builder's profiler, 187–194

- memory profiling, 194–206
- overview of, 185–186
- profiling techniques, 186–187
- summary, 212
- Performance, Rails, 206–212**
 - benchmark script, 209
 - benchmarking, 207–208
 - overview of, 206
 - performance scripts, 208
 - profiler script, 208–209
 - reading log files, 206–207
 - request script, 209–210
 - RPM (Rails Performance Management), 211–212
 - summary, 212
- Pie charts, 147–148**
- PNG files, Flex `URLLoader` class for uploading, 318–320**
- Policies, cross domain, 8**
- Positions, Stock Portfolio application example, 43–47**
- Prana**
 - `EventSequence`, 261–265
 - reusing Cairngorm commands, 259–261
 - runtime configuration of Flex, 289–293
 - sequences, 259
 - summary, 265
- prana-main.swc, 289–293**
- Production ready use case, RubyAMF, 93**
- Profiler, Flex Builder, 187–194**
 - capturing profiles, 190
 - code for performance profiling example, 187–188
 - determining problem areas, 192–194
 - overview of, 187
 - profiling an application, 189–190

- resetting to clear unneeded information, 192
 - types of data collected, 191–192
- Profiler script, Rails, 208–209**
- Profiling techniques, Flex, 186–187**
- Projects directory, Flex SDK files, 239**
- Prototype JavaScript library, 305–307**
- Proxies, in PureMVC model, 160–161, 181**
- public directory, 7**
- PureMVC, 159–163**
 - controller, 159–160
 - Fluint for testing and, 49
 - implementing, 176–182
 - model, 160–161
 - overview of, 159
 - service layer, 161–163
 - view, 161
- Push servers, 298**
- Push technology, 295. See also Juggernaut Push Technology**

R

- RadRails, 5**
- Rails**
 - accessing RESTful applications with Flex, 39–48
 - associations, 238
 - built-in testing framework, 49
 - command line debugger, 123–127
 - creating application, 275–276
 - creating MXML component for invoking Rails application, 53–54
 - creating new application, 29–39
 - creating new project, 57
 - CRUD support added to application, 282–288
 - debugging, 116–117

- deploying applications with Capistrano, 227–231
 - file upload and, 313–314
 - fixture controller, 80–81
 - fixtures and, 79
 - installing, 4
 - lifecycle events and, 248–250
 - logging, 106–108
 - mapping data types to Flex, 21–24
 - moving hierarchical data from Rails to Flex. *See* Nested sets
 - as open source, 233–244
 - performance. *See* Performance, Rails
 - sending XML between Flex and Rails, 17–19
 - sending XML data to/from Flex, 19–21, 24–25
 - source code, 235–238
 - XML in, 11–14
- Rails Performance Management (RPM), 211–212**
- Rake, 221–225**
 - overview of, 221–222
 - Rakefile, 222–225
 - summary, 225
- `rake routes` **command, 31**
- Rakefile, 222–225**
- Records, active record validation (HTTP 400s), 26**
- Relational databases, multidimensional databases compared with, 134–135**
- Remote calls, Fluint for testing, 55**
- Remote Method Invocation (RMI), 85**
- Remote Procedure Calls (RPCs), 85**
- `RemoteObject` **class**
 - `AsyncToken` and, 172
 - authentication and, 256
 - debugging and, 128
 - properties, 95
- Remoting, 94. *See also* RubyAMF**
- Request script, Rails, 209–210**
- Resources**
 - accessing, 30–32
 - nesting, 43
- Responders, Cairngorm, 156, 171**
- RESTful services, 29–48**
 - implementing RESTful controllers, 93
 - Rails applications and, 11
 - RubyAMF and, 101–103
 - Stock Portfolio example. *See* Stock Portfolio, RESTful application example
- `restful_authentication` **plug-in, 251–257**
- `RetrieveNotesCommand` **class, 71–72**
- RIAs (rich Internet applications), 303**
- Rich Internet applications (RIAs), 303**
- Rich Text Editor, 246**
- RMI (Remote Method Invocation), 85**
- `Routes` **command, routing requests, 31–32**
- RPCs (Remote Procedure Calls), 85**
- RPM (Rails Performance Management), 211–212**
- RSLs (Runtime Shared Libraries), 228**
- Ruby**
 - locations, 235
 - software needed to work with Flex and Rails, 3–4
- `ruby-debug`
 - debugging Rails, 123–127
 - installing, 5
- `ruby-debug-ide`
 - debugging Rails applications, 116–117
 - installing, 5
- Ruby Gems, 4**

`ruby-prof` gem, 208

Ruby Standard Library, 106

RubyAMF

- authentication and, 252–256
- configuring, 87–91
- example, 95–101
- Flex with, 94–95
- generators, 91–92
- installing, 87
- nested set for moving hierarchical data. *See* Nested sets
- overview of, 87
- RESTful services and, 101–103
- summary, 103
- workflow, 92–93

Runtime Shared Libraries (RSLs), 228

S

`SaveChangeCommand` class, **Cairngorm, 72–74**

SCM (source control management)

- benefits of, 218
- Git, 217–219
- goals and solutions, 215
- ignoring files in Subversion, 215–217
- Subversion, 218

Scope, debugging and, 110–111

script directory, 8

`script/generate` command

- creating Rails application, 30
- RubyAMF generators, 91–92

Scripts, loading test data with, 276–279

Scripts, Rails performance, 208

- benchmark script, 209
- profiler script, 208–209
- request script, 209–210

Security Sandbox, Flash Player, 303–304

Sequences, Prana EventSequences, 261–265

Server errors (HTTP 500s), 26

Server push. *See* **Juggernaut Push Technology**

Server round-trip, Cairngorm commands and, 70–71

Servers, Rails

- creating Rails messaging application, 298–299
- push technology. *See also* **Juggernaut Push Technology**
- running, 9
- testing queries to, 54–57

Service layer

- Cairngorm, 157–158
- PureMVC, 161–163

`ServiceLocator`

- implementing Cairngorm and, 171
- overview of, 157
- testing Cairngorm-based applications, 62–63

`SessionsController`,
`restful_authentication` **plug-in, 253–254**

Sets, nesting. *See* **Nested sets**

SOS, third party loggers for Flex, 109

Source code, 233–243

- benefits of open source and, 233–234
- Flex, 238–240
- generated Flex source, 240–243
- overview of, 233
- Rails, 235–238
- summary, 243

Source control management. *See* **SCM (source control management)**

SQL

- nested sets and, 267
- RPM (Rails Performance Management) and, 212

SQLite 3, 3**SQLite Gem, 5****src directory, 8, 98–102****Stacks, debugging and, 110–111****Star schema, table organization, 135****Stock Portfolio, RESTful application example**

- account resource, 39–43

- creating, 29–39

- movements resource, 47–48

- OLAPCube applied to, 140–144

- positions resource, 43–47

Stuff, 163–169

- controllers, 166–168

- example web application, 164

- GTD model and, 163

- models, 166

- organization of web application, 165

Subclipse, 218**Subversion**

- ignoring files in, 215–217

- overview of, 218

svn. See **Subversion**

SWF files

- accessing with Rails server to avoid security issues, 304

- compiling, 240

- deploying Flex applications and, 228–229

- SCM (source control management) and, 218

SWFObject, JavaScript library, 305–307

T

Tasks, Rake, 222–223**Tasks, Stuff entities, 163****test directory, 8****Test environment, setting up, 52–53****TestCase class**

- assertion methods, 59

- asyncResponder method, 55–56

- setting up Flex test environment, 52

- unit testing and, 50

testCreate, testing Cairngorm applications, 67**testDelete, testing Cairngorm applications, 69–70****TestHelper, testing Flex fixtures, 81–82****Testing Cairngorm-based applications. See Cairngorm applications, testing****Testing Flex with Rails application. See Fluint (Flex Unit and Integration)****testList, testing Cairngorm applications, 65–66****TestNotesResource test case**

- adding tests to, 54–57

- setting up Flex test environment, 52–53

TestRunner class

- setting up Flex test environment, 52

- testing Cairngorm-based applications, 66–67

- unit testing and, 50

TestSuite class, unit testing and, 50**testUpdate, testing Cairngorm applications, 68–69****TextMate**

- debugging Rails applications, 117

- installing, 6

Things, GTD model, 163**ThinkingRock, GTD model, 163****ThunderBolt, third party loggers for Flex, 109****tmp directory, 8****trace statement, for logging, 109****Transfer Objects, integrating remote services, 94**

U

Unit tests, 50–51. *See also* **Fluint (Flex Unit and Integration)**

Update services, account resource and, 41

URLoader class, for uploading PNG files, 318–320

URLs

- accessing resources, 30–32
- for account resource, 40
- controlling `HTTPService` URL at runtime, 293
- position resource and, 44–45

User authentication, 251

“User gestures”, Cairngorm, 155

User interface

- methods for supporting HTML user interface, 35
- `NoteTake`, 60–62

UserController, `restful_authentication` plug-in, 253

V

Validation methods, Rails applications, 37–38

Value Objects (VOs)

- Cairngorm, 155
- remote services, 94–95

Variables tab, Flex debugging, 113

vendor directory, 8

View helpers, in app directory, 7

Views

- in app directory, 7
- Cairngorm, 156
- levels of Flex unit tests, 50
- PureMVC, 161

VOs (Value Objects)

- Cairngorm, 155
- remote services, 94–95

W

Watch expressions, 113–114

Weak references, Garbage Collector and, 194

Web proxy debugger (Charles), 127–128

Web servers, Mongrel vs. WEBrick, 4–5

WEBrick, 4

Windows computers

- Git and, 219
- Ruby installation on, 4
- Ruby location on, 235
- Subversion for, 218

Workflow, RubyAMF, 92–93

X

XML, 11–28

- cross domain policy files and, 8
- error handling, 25
- Flash Player restrictions, 26–28
- HTTP status codes, 25–26
- mapping data types between Flex and Rails, 21–24
- moving XML to Flex, 17–19
- moving XML to Rails, 19–21
- nested sets and, 277–278
- overview of, 11
- parsing rules, 48
- sending data between Flex and Rails, 24–25
- summary, 28
- test format of, 86
- using in Flex, 14–17
- using in Rails, 11–14