

Enterprise Flexible Rails

MEAP

Unedited Draft

Peter Armstrong
Dima Berastau

 MANNING



Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=440>



**MEAP Edition
Manning Early Access Program**

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=440>

Table of Contents

Chapter 1 - Why Ruboss?

Chapter 2 - Developing using Ruboss and XML over HTTP

Chapter 3 - Beyond the Generators

Chapter 4 - Hello, Enterprise: Ruboss with BlazeDS and AMF

Chapter 5 - Testing with Ruboss

Chapter 6 - Ruboss on Merb

Chapter 7 - Ruboss on Google App Engine

Chapter 8 - Ruboss on AIR

Appendix A - The Ruboss API Reference

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=440>

1

Why Ruboss?

Adobe Flex and Ruby on Rails are both extremely productive frameworks, and ActionScript 3 and Ruby are great languages to develop production software in. Rails offers many productivity improvements, through its implementation of the principles of Convention Over Configuration, Don't Repeat Yourself (DRY) and being both Opinionated and Less Software. In *Flexible Rails*, I showed how Flex and Rails can be used together to rapidly build Rich Internet Applications. Doing so gives you the added power and improved productivity and maintainability of Flex over the typical AJAX-heavy Rails application. Combining Flex and Rails is, in many senses, combining two "best of breed" technologies to produce a hybrid application which embodies the strengths of both technologies.

DO I NEED TO HAVE ALREADY READ *FLEXIBLE RAILS* TO READ THIS BOOK?

No! While this book is in some sense the sequel to *Flexible Rails*, it can be read without having read *Flexible Rails*. Having read *Flexible Rails* obviously helps; however, if you already understand the basics of Adobe Flex and Ruby on Rails—or are willing to learn as you go—you can read this book effectively. In this chapter, I will explain the basics of these technologies and how they are used together. If you are an advanced developer, this may be sufficient—you may be able to figure the rest out by osmosis. However, if you want more information, I believe that reading *Flexible Rails* is a good investment of time and money: it is over 500 more pages of Flex and Rails learning by osmosis.)

However, as readers of *Flexible Rails* have noticed, while combining Flex and Rails is a very productive combination, it does not feel as integrated as pure Rails development. Rails is a tightly-integrated Model-View-Controller (MVC) framework, where ActiveRecord is the model, ActionView is the view and ActionController is the controller. When combining Flex and Rails, this combination—although more powerful—feels less integrated. This is to some degree inevitable: what I did in *Flexible Rails* (and continue to do in this book) is to essentially throw out ActionView and use ActionController to talk directly to Flex.

In *Flexible Rails*, I started on the Flex side by building a simple application which used **HTTPServices** directly in MXML to talk to Rails. As this application grew and the code became more complex, I refactored the code to using Cairngorm and added some useful utility classes. This was the point where I stopped convincing many of the more advanced Rails developers who were reading *Flexible Rails* (or who saw one of my presentations about the same material over the last two years): An advanced Rails developer told me after my Rails to Italy presentation that "you had me until Cairngorm." Cairngorm has this effect on many Rails developers, which makes sense since it is the result of applying J2EE patterns to Flex development. Nothing is more of a turnoff to many Rails developers than J2EE, so Cairngorm is a tough sell. (Since I like a challenge, I tried to sell it by saying that it was good for larger teams to develop in parallel without stepping on each others' toes. However, this didn't help much: while Rails can be used effectively in large teams, it is targeted at smaller, more agile teams in which protecting people from themselves and each other isn't the primary concern.)

In *Flexible Rails*, I tried to work around the J2EE style of Cairngorm by providing various helper classes and methods (e.g. the ServiceUtils class) in order to make using Cairngorm less verbose and more suited to HTTPService. However, these were half-measures. More generally, I came to realize that while

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=440>

Flex and Rails are extremely complementary, there is also a mismatch in the philosophies of the current state of the art in developing Flex and Rails applications.

I met Dima Berastau at the Vancouver Flash/Flex Meetup while I was still writing *Flexible Rails*. At the time, he didn't have much use for Ruby, preferring Scala. However, after reading a draft of *Flexible Rails*, he agreed that Ruby and Rails were very productive and enjoyable to program in. After I finished *Flexible Rails*, over many pints of beer Dima and I realized something:

This should be easier.

On the back of Flexible Rails, Manning chose a car analogy (a Ferrari) to describe the combination of Flex and Rails. (My joke was that this was more true when Flex was at 1.0: it was very expensive and required a lot of maintenance!) However, joking aside, I have realized that this analogy is more appropriate than even Manning realized: The approach taken in Flexible Rails is like driving a manual transmission car—you do all the shifting yourself, etc. Wouldn't it be nice if there was a higher level of abstraction to work at? Say, an application framework for Flex, but instead of being optimized for large teams used to J2EE (like Cairngorm), it would be optimized for the task of communicating with a Rails backend? More generally, a framework that was optimized for a Resource Oriented Architecture (ROA) on the back end, and that made the task of doing RESTful CRUD completely automatic?¹

Right about now, you may be worrying that this framework would abstract away the low level details, meaning that you would lose control. As Joel Spolsky eloquently argues in his blog post "The Law of Leaky Abstractions"², abstractions do leak. So, if a framework were to abstract away all the low level details, you would suffer a significant loss of control. Continuing Manning's car analogy, this is the weakness of an automatic transmission: it's more pleasant for the boring commute (and lets you drink coffee easier while you're at it), but you don't have the extra control when you need it.

Sticking with the car analogy a bit longer actually leads us somewhere: In the real world of *my car*, I own a Subaru Outback which has had head gasket issues³ recently. As such, I have had a number of loaner Subaru cars lately. Some of them have featured a Sportronic transmission, which can function in either fully automatic mode (for the boring commute) or in essentially manual mode (when you need or want the control). To me, this is the perfect analogy for describing what the ideal framework for making Flex and Rails work together should do: it should do the simple RESTful CRUD stuff for you (automatic shifting mode), while gracefully allowing you to bypass the framework (manual shifting mode) when you need to do something other than what the framework provides. This framework should learn from the best practices introduced by Rails, and apply them as appropriate to Flex development.

This is the lofty goal of the Ruboss framework (<http://code.google.com/p/ruboss/>), an Open Source framework which was designed and coded by myself and Dima Berastau, who is the co-author of this book and the co-founder of our company Ruboss Technology Corporation. (*Dima did most of the heavy lifting, which is why he's Ruboss's CTO!*) We are trying to bring Flex and Rails closer together and make developing using them together even more enjoyable. This is a *very* self-serving goal: we make our living doing custom Flex and Rails development, so we want to ensure we enjoy ourselves as much as possible while we're doing it. Furthermore, Dima likes to ski and spends too much time cycling, and I like to snowboard and I have a family—so we need our framework to make us as productive as possible so we can spend time doing other things. (*Also, we have a company to run, and we need to keep our editors at Manning happy!*)

¹ If that sentence didn't make any sense to you, don't worry: it was there for the architects.

² <http://www.joelonsoftware.com/articles/LeakyAbstractions.html>

³ *So please buy Flexible Rails instead of copying it—I need the royalties!*

Please post comments or corrections to the Author online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=440>

DOESN'T THIS MEAN YOU'RE BIASED?

Yes! Obviously, since we are the authors of this book and the creators of the Ruboss framework, we are completely biased—so please adjust your expectations accordingly. (This isn't unprecedented though: many good books about frameworks, for example Manning's *Java Persistence with Hibernate*, are written by their creators.)

This book explains the Ruboss framework and shows how using it can simplify your Flex and Rails development. Furthermore, it shows how you can scale the Flex + Rails combination using Ruboss to the more demanding scale of the Enterprise—hence the title *Enterprise Flexible Rails*. Without sounding too confident, I am convinced that this is currently one of the most productive ways to build Rich Internet Applications—and the best way to build a RESTful CRUD-based Rich Internet Application. If this describes your application (it describes most Rich Internet Applications, so there is a good chance that it does), then the Ruboss framework and this book may be for you.

DOES THE RUBOSS FRAMEWORK MEAN THAT *FLEXIBLE RAILS* IS OBSOLETE?

No! While the Ruboss framework simplifies the common use cases (*the goal is to handle the typical 80% of the cases with 20% of the code, a variant of the Pareto principle*), the Ruboss framework does not do everything for you. It is not a magic coding fairy, and does not abstract you from the need to understand what is going on. So, since *Flexible Rails* teaches the manual way of doing everything, it is *by no means* obsolete. Just because automatic transmissions exist, you are a better driver if you can drive a standard transmission! ⁴ What I presented in *Flexible Rails* was—and still is—the state of the art for making Flex talk to Rails *when you're doing everything yourself*. Finally, even when using Ruboss, there will be times when you need to step out of the framework and do everything yourself. (This is true when what you're doing does not lend itself to the RESTful approach.)

So, what next for this chapter?

We will start by getting an extremely brief overview of Flex and Rails, in case you haven't used one of them before. (See, I told you I wasn't going to leave you out if you hadn't read *Flexible Rails*.) We will then install everything that you need to follow along with the book. After this, the fun begins. We will briefly look at what a very small Flex and Rails application looks like, in case you have never seen one before. The application will be a trivial To Do list application, since I want the application to be as small as possible while still doing the full range of CRUD operations. (I considered subsequently refactoring this application to use Cairngorm, but decided against it: Cairngorm is not intended to be used for very small applications, so this would be a "straw man" argument at best.)

After building the To Do list application, we will refactor it to using Ruboss. This will give you a taste of what Ruboss has to offer. Furthermore, since we will be doing everything manually at first, you will see that the primary productivity improvements of Ruboss are provided when you are writing code yourself. (Like Rails, Ruboss offers generators—however, also like Rails, the generators are just a small part of the story that make getting started easier.)

So, without further ado, let's get going...

⁴ I learned to drive in a manual transmission Chevette—in Saskatchewan, in 10 feet of snow, uphill both ways, parallel parking between cows and igloos—so I am a strong believer in the ability to do things the difficult way yourself.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

1.1 The 30,000 Foot View of Flex and Rails

NOTE

If you have read *Flexible Rails* or the Flexible Rails "Refcard" at <http://www.dzone.com/>, you can skip this section and proceed with section 1.2.

1.1.1 Overview of Rails 2

Rails provides a standard three-tier architecture (presentation tier, model tier, persistence tier) as well as a Model-View-Controller (MVC) architecture. As shown in Figure 1, Rails takes care of everything between the web server and the database.

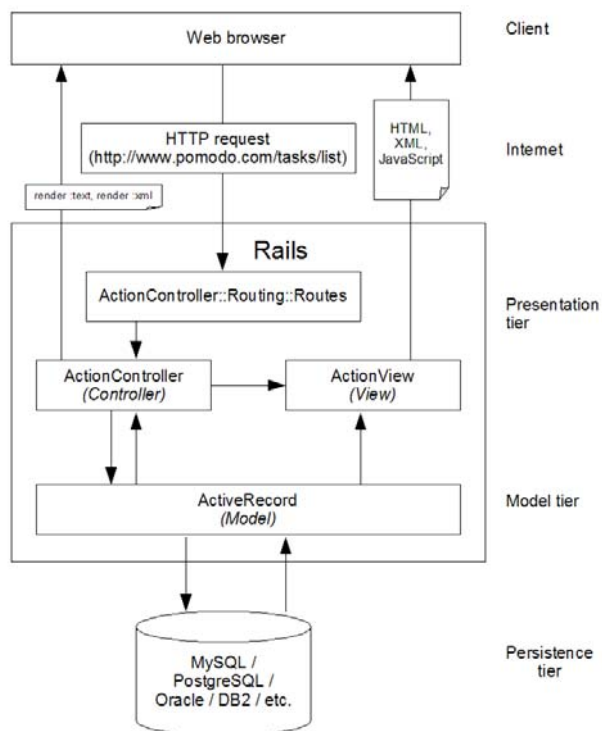


Figure 1: Rails provides a standard three-tier architecture (presentation tier, model tier, persistence tier) as well as a Model-View-Controller architecture.

The typical sequence is as follows:

1. A user visits a particular URL in their web browser (makes an HTTP request).
2. This request goes over the Internet to the web server in which Rails is running.
3. That web server passes the request to the routing code in Rails, which triggers the appropriate controller method call based on the routes defined in `config/routes.rb`.
4. The controller method is called. It communicates with various ActiveRecord models (which are persisted to and retrieved from a database of your choosing). The controller method can then do

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

one of two things: (1) Set some instance variables and allow a view template (a specially named .html.erb file, for example) to be used to produce HTML, XML, or JavaScript, which is sent to the browser. (2) Bypass the view mechanism and do rendering directly via a call to the render method. This method can produce plain text (`render :text => "foo"`), XML (`render :text => @task`), and so on.

1.1.2 Overview of Flex 3

In Flex 3, you write code in MXML (XML files with a .mxml extension; *M* for Macromedia) and ActionScript (text files with a .as extension) files and compile them into a SWF file, which runs in the Flash player. This SWF is referenced by an HTML file, so that when a user with a modern web browser loads the HTML file, it plays the Flash movie (prompting the user to download Flash 9 if it's not present). The SWF contained in the web page can interact with the web page it's contained in and with the server it was sent from.

Even if you have never created a Flash movie in your life, do not consider yourself a "designer" and wouldn't recognize the Timeline if you tripped over it, you can use Flex to create attractive applications that run in the Flash player. Flex development is easily learned by any intermediate-level developer with either web or desktop UI (such as Windows Forms or Java Swing programming experience).

FLASH 9? ARE YOU KIDDING ME?

The reference to Flash 9 may have set off alarm bells in your head: "Isn't Flash 9 somewhat new? How many people will be able to run my app?" Well, while not everyone has Flash 9, over 95% of your market probably does, according to Adobe's "version penetration" chart⁵. *This is better than Windows*. Furthermore, despite how productive Flex 3, Rails 2 and Ruboss are for development, it will still take you *some time* to build your killer app. And in that time, your target market is getting larger by the day. (If you haven't accomplished much in a given day, you can still feel good that you grew your target market.) Finally, note that most of your early adopters will be, well, early adopter types. These are the TechCrunch reading, Digg / del.icio.us / reddit using types. These people will have Flash 9 or won't mind getting it.

1.1.3 Overview of using Flex 3 and Rails 2 together

Flex and Rails can be used together with XML over HTTPService or with AMF. The XML over HTTPService approach is shown in Figure 2.

⁵ http://www.adobe.com/products/player_census/flashplayer/version_penetration.html

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

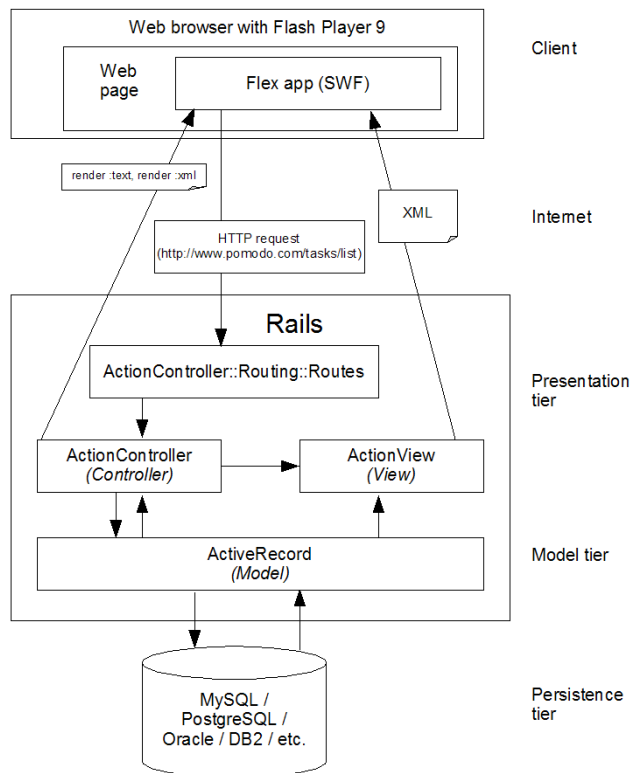


Figure 2: How you can use Flex and Rails together with XML over HTTPService

While the HTTPService story is pretty straightforward, the AMF waters are a bit muddier. There are currently three ways that Flex can talk to Rails using AMF and RemoteObject:

1. RubyAMF
2. WebORB for Rails
3. BlazeDS (with Rails running on JRuby)

My personal bet is currently on **BlazeDS**, since it's made by Adobe and since it was extracted from the Flex Data Services / LiveCycle Data Services product. We will discuss the BlazeDS + Rails + JRuby combination later in this book. (In *Flexible Rails* I discussed RubyAMF; that chapter is freely available from <http://manning.com/armstrong/> so I don't need to discuss it here.)

1.2 Installing Everything

We will start by installing everything. You need to download and install the following software:

- Adobe Flex Builder 3 (<http://www.adobe.com/products/flex/>: free 60 day trial, \$249 standard edition, \$699 professional edition) or the free, Open Source Adobe Flex SDK (http://www.adobe.com/products/flex/features/flex_framework/). The book will assume Flex Builder; if you are using the SDK you are experienced enough to follow along without any help.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

- Ruby 1.8.6
- Rails 2.0.2
- MySQL 5 (although you should be able to adapt everything in the book to SQLite or any Rails-supported database without any trouble)

I'm not going to go into detailed instructions for installing these tools; there are many fine tutorials online. (I also devoted a chapter to this in *Flexible Rails*, in which I wrote [and rewrote, repeatedly] three separate sets of instructions for Windows + Flex Builder, Windows + Flex SDK and Mac OS X + Flex SDK. That was so excruciatingly tedious that I'm never doing it again—you only live once.) If you are stuck getting up and running and learning the basics, may I recommend a copy of *Flexible Rails*: the PDF is only \$20 USD from <http://manning.com/armstrong/>—so if it saves you a few hours getting started, or prevents you from tearing out your hair, it should be well worth the money (hair replacement is expensive!).

1.3 Building a To Do List with Flex, Rails and HTTPService

NOTE

If you have read *Flexible Rails* or the Flexible Rails “Refcard” at <http://www.dzone.com/>, you can skip this section and proceed with section 1.4.

We will start by creating a simple To Do list with Flex, Rails and HTTPService. No, the world doesn't need Yet Another Todo List, but it's going to get one⁶. This will serve as an extremely quick introduction to Flex and Rails, in case you haven't seen one or both of them before.

NOTE ABOUT COMMANDS

This book will assume you are using OS X. Primarily, this is because this is what I am using, and because it's the standard assumption with Rails tutorials. In *Flexible Rails* I assumed Windows, so I'm not playing favorites. If you are following along with Windows, just run commands in a command prompt instead of a Terminal window and use backslashes (“\”) instead of forward slashes (“/”). Your Ruby scripts will be run with **ruby**, not **./**, for example **ruby script\generate rconfig** not **./script/generate rconfig**. Also, instead of creating the projects in your home directory (which looks like ~ in my command listings), I recommend using **c:** or **c:\book**. This way, the paths will be fairly short.

Open a Terminal window and run the following commands:

```
~ $rails todo
  create
  create  app/controllers
...
  create  log/development.log
  create  log/test.log
```

In this tutorial we will use the default database of SQLite, which is the new default database choice in Rails as of Rails 2.0.2. The default in Rails 2.0.1 and below was MySQL; to configure the `config/database.yml` file to use it now you need to say `rails -d mysql todo` instead of `rails todo`. (The reason why we installed MySQL earlier is that in the remaining chapters of the book we will use MySQL.)

⁶ Actually, it got two more: this section is inspired by the Flexible Rails “Refcard” at <http://www.dzone.com/>. That refcard was obviously inspired by the *Flexible Rails* Manning book, so the cycle begins again I guess...

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

Next, we create a couple directories:

```
~ $cd todo
~/todo $mkdir app/flex
~/todo $mkdir public/bin
```

Next, switch to Flex Builder 3 and create the new Flex project:

1. Do File > New > Flex Project
2. Choose to create a new project named Todo in ~/todo (which for me is /Users/peter/todo, since my home directory is /Users/peter)
3. Leave its type set to "Web application" and its "Application server type" set to None and click Next
4. Set the Output folder of the Flex project to public/bin and click Next
5. Set the "Main source folder" to app/flex, leave the "Main application file" as Todo.mxml and set the output folder to <http://localhost:3000/bin> and click Finish. Your new Flex project will be created. (Note that public isn't part of the path since it's the root; 3000 is the default port for the server).

You will see a new Application, as shown in listing 1.1.

Listing 1.1 app/flex/ToDo.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  layout="absolute">

</mx:Application>
```

The top-level tag is **mx:Application**; the root of a Flex application is always an **Application**. The **mx:** part identifies the XML namespace that the Application component is from. By default, an Application uses an absolute layout, where you specify the x,y of each toplevel container and component.

CODE SAMPLES, DIFFS AND WIDTH

I'm modifying the generated code to be 64-column code, because I need to fit the code and the cueball annotations within 68 columns. Because these diffs would be distracting, I'm not using the bold font for them. Normally, I will format code changes ("diffs") in **bold** for new code, ~~strikethrough~~ for deleted code and plain text code font for code which remains unchanged. This way, you can follow along with everything in the book without pain: you will know exactly where to add code, and where to delete it. This is an extremely tedious process for me to do, but I'm convinced it makes the book much better. However, sometimes when almost the entire file is changing, I will just show the entire file as **new code**, with the implication being that it will be easier for you to just copy the code fresh from the book. (If you have the PDF of the book, you can copy code directly from the PDF to Flex Builder.)

Next, let's create a new Task resource using the scaffold command, which is RESTful by default as of Rails 2:

```
~/todo $./script/generate scaffold Task name:string notes:text
exists  app/models/
exists  app/controllers/
...
create  app/helpers/tasks_helper.rb
route   map.resources :tasks
```

Here we are creating a Task that has a **name** attribute which is a **string** and a **notes** attribute which is **text**. Running this command generates the various Rails files, including the model, helper, controller, view templates, tests and database migration. (This is going to be one of the simplest Todo lists in history: Tasks have names, notes and nothing else. Furthermore, there are no users even, just a global list of tasks.)

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

The Task model is shown in listing 1.2.

Listing 1.2 app/models/task.rb

```
class Task < ActiveRecord::Base
end
```

Because the **Task** model extends (with <) `ActiveRecord::Base`, it can be mapped to the equivalent database tables. Because we also created the controllers and views with the `script/generate scaffold` command and ensured that we specified all the fields, we can use a prebuilt web interface to Create, Read, Update, and Delete (CRUD) them.

The `CreateTasks` migration that was created is shown in listing 1.3:

Listing 1.3 db/migrate/001_create_tasks.rb

```
class CreateTasks < ActiveRecord::Migration #1
  def self.up #2
    create_table :tasks do |t|
      t.string :name
      t.text :notes

      t.timestamps #3
    end
  end

  def self.down #4
    drop_table :tasks
  end
end
```

Cueballs in code and text

#1 extend ActiveRecord::Migration
#2 up method creates new tasks table
#3 create created_at and updated_at columns
#4 down method drops table

The `CreateTasks` class extends `ActiveRecord::Migration` (#1), as all migrations do. The `up` (#2) method creates a new `tasks` table with the `create_table` method call, which takes a block that does the work; the `down` method (#4) deletes it with the `drop_table` call. In the `up` method, we specify the data types of each new column, such as `string`, `boolean`, `integer` or `text`. These are then mapped to the equivalent database data types. The `timestamps` (#3) call adds two columns: `created_at` and `updated_at`, which Rails treats specially, ensuring that they're automatically set. This is often a good thing to have, so we'll leave them there.

1.3.1 A tour of the TasksController

The `script/generate scaffold` command created more code in the controller and views than in the model. In order to demystify it, we'll look at the code that was created for the `TasksController`, shown in listing 1.4.

Listing 1.4 app/controllers/tasks_controller.rb

```
class TasksController < ApplicationController #A
  # GET /tasks
  # GET /tasks.xml
  def index #1
    @tasks = Task.find(:all)

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

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

```

    end
  end

  # GET /tasks/1
  # GET /tasks/1.xml
  def show #2
    @task = Task.find(params[:id])

    respond_to do |format|
      format.html # show.html.erb
      format.xml { render :xml => @task }
    end
  end

  # GET /tasks/new
  # GET /tasks/new.xml
  def new #3
    @task = Task.new

    respond_to do |format|
      format.html # new.html.erb
      format.xml { render :xml => @task }
    end
  end

  # GET /tasks/1/edit
  def edit #4
    @task = Task.find(params[:id])
  end

  # POST /tasks
  # POST /tasks.xml
  def create #5
    @task = Task.new(params[:task])

    respond_to do |format|
      if @task.save
        flash[:notice] = 'Task was successfully created.'
        format.html { redirect_to(@task) }
        format.xml { render :xml => @task, :status => :created,
          :location => @task }
      else
        format.html { render :action => "new" }
        format.xml { render :xml => @task.errors,
          :status => :unprocessable_entity }
      end
    end
  end

  # PUT /tasks/1
  # PUT /tasks/1.xml
  def update #6
    @task = Task.find(params[:id])

    respond_to do |format|
      if @task.update_attributes(params[:task])
        flash[:notice] = 'Task was successfully updated.'
        format.html { redirect_to(@task) }
        format.xml { head :ok }
      else
        format.html { render :action => "edit" }
        format.xml { render :xml => @task.errors,
          :status => :unprocessable_entity }
      end
    end
  end
end

```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=440>

```

    end
  end
end

# DELETE /tasks/1
# DELETE /tasks/1.xml
def destroy #7
  @task = Task.find(params[:id])
  @task.destroy

  respond_to do |format|
    format.html { redirect_to(tasks_url) }
    format.xml { head :ok }
  end
end
end
end

```

No Cueball

#A All controllers extend ApplicationController

Cueballs in code and text

#1 index method
#2 show method
#3 new method
#4 edit method
#5 create method
#6 update method
#7 destroy method

First, note that all controllers extend **ApplicationController**. Next, we see that **scaffold** gives us a lot, creating the seven standard Rails methods—**index** (#1), **show** (#2), **new** (#3), **edit** (#4), **create** (#5), **update** (#6), and **destroy** (#7)—that are used by RESTful controllers. These methods are explained in the table 1.1, which is taken from *Flexible Rails*.

Table 1.1 The seven standard RESTful controller methods

#	Method	Sample URL paths	Pretend HTTP method	Actual HTTP method	Corresponding CRUD method	Corresponding SQL method
1	index	/tasks	GET	GET	READ	SELECT
2	show	/tasks.xml /tasks/1	GET	GET	READ	SELECT
3	new	/tasks/1.xml /tasks/new	GET	GET	-	-
4	edit	/tasks/new.xml /tasks/1/edit	GET	GET	READ	SELECT
5	create	/tasks /tasks.xml	POST	POST	CREATE	INSERT

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

6	update	/tasks/1	PUT	POST	UPDATE	UPDATE
7	destroy	/tasks/1.xml	DELETE	POST	DELETE	DELETE
		/tasks/1				
		/tasks/1.xml				

Note that each method has a comment showing the URL + fake HTTP request combination that triggers it. For example, the **index** method (#2) (which is triggered by a **GET** to /tasks or /tasks.xml) finds all the tasks and stores them in **@tasks**. Then, inside the **respond_to** block, it checks if the requested format was **html** (and renders index.html.erb if it was) or **xml** (and renders **@tasks**, which is syntactic sugar for **@tasks.to_xml**). Furthermore, let me emphasize that these seven methods are the *only* methods you should have in a RESTful controller: If you think you're special and need more, well, as Scott Raymond⁷ quoted *Fight Club*, "You are not a special or unique snowflake." If you think you're unique, try adding a noun.

WHAT'S REST?

REST (Representational State Transfer) is a way of building web services that focuses on simplicity and an architecture style that is "of the web." This can be described as a Resource Oriented Architecture (ROA); see *RESTful Web Services* (O'Reilly Media, Inc.) for details. As you can probably infer from its \$20-word full title, REST grew out of a PhD thesis—Roy Fielding's, to be precise. However, unlike most PhD theses, it has grown into something revolutionary. Briefly, the reason to use a RESTful design in Rails is that it helps us organize our controllers better, forces us to think harder about our domain, and gives us a nice API for free.

Be sure to keep in mind that the automatically generated code in the controller and views that is created by the **scaffold** command is just that: scaffolding to help us get going. There's nothing magical about it: the first thing we typically do is start modifying and deleting methods generated by the **scaffold** command. At this point, if you want to scan through the .html.erb templates (so named because they're processed by ERb, Embedded Ruby) in app/views/tasks, feel free to do so. These scaffolded views (and the similar ones in app/views/projects and app/views/locations) are useful for testing. Because we're using Flex instead of Action View, we'll leave explanations of how Action View works to other books. If you have to output HTML, Rails has you covered, with Action View for HTML and RJS templates for JavaScript. (As you will see shortly, Ruboss takes this scaffolding idea even further.)

Next, we run the CreateTasks migration that we created earlier as part of running the scaffold command:

```
~/todo $rake db:migrate
(in /Users/peter/todo)
== 1 CreateTasks: migrating =====
-- create_table(:tasks)
   -> 0.0041s
== 1 CreateTasks: migrated (0.0044s) =====
```

We then run our server:

```
~/todo $./script/server
=> Booting Mongrel (use 'script/server webrick' to force WEBrick)
=> Rails application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
** Starting Mongrel listening at 0.0.0.0:3000
...
```

⁷ http://conferences.oreillynet.com/presentations/rails2007/ramond_scott.pdf, slide 30.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

At this point, it's useful to play with creating, editing and deleting tasks, to ensure that everything works:

1. Go to <http://localhost:3000/tasks> to see an empty task list.
2. Click the New link to go to <http://localhost:3000/tasks/new>.
3. Create a new Task with a name of "drink coffee" and notes of "espresso" and click Create.
4. Go back to <http://localhost:3000/tasks> to see the task list with the new "drink coffee" task present.

Now, let's do something interesting and hook this up to Flex. What we want is a nice, simple UI; thanks to 37signals, simple is where it's at these days. We want to build something like figure 1.1.

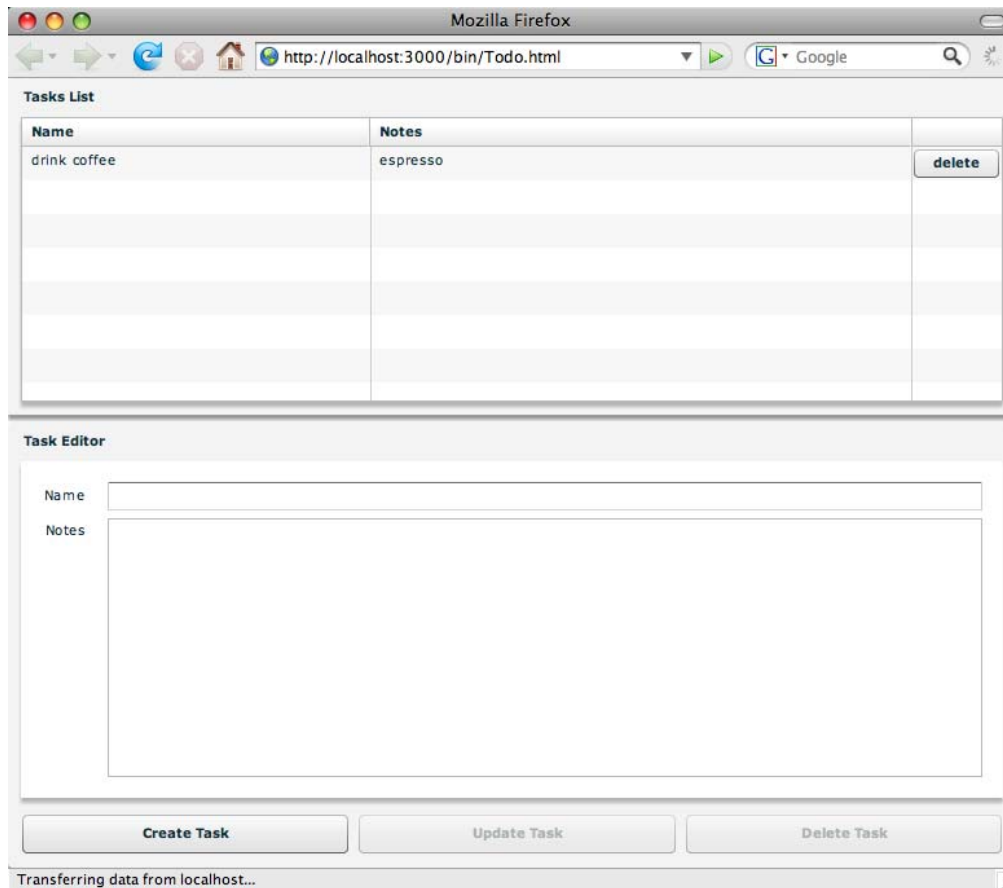


Figure 1.1 A Simple To Do List

The screenshot shows the ability to create, update and delete tasks, with the lower "Update Task" and "Delete Task" buttons only enabled if a task is selected. So, let's do this all at once; replace the contents of the Todo application with those shown in listing 1.9.

Listing 1.9 app/flex/ToDo.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    layout="vertical" #1
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

```

    styleName="plain"
    initialize="listTasks()" #2
<mx:Script><![CDATA[
    import mx.events.ListEvent;
    import mx.controls.Alert;
    import mx.rpc.events.ResultEvent;

    [Bindable]
    private var task:XML;

    private function createTask():void { #3
        svcTasksCreate.send();
    }

    private function listTasks():void { #4
        svcTasksList.send();
    }

    private function updateTask(task:XML):void { #5
        var params:Object = new Object();
        params['task[name]'] = nameTI.text;
        params['task[notes]'] = notesTA.text;
        params['_method'] = "PUT";
        svcTasksUpdate.url = "/tasks/" + task.id + ".xml";
        svcTasksUpdate.send(params);
    }

    public function destroyTask(task:XML):void { #6
        svcTasksDestroy.url = "/tasks/" + task.id + ".xml";
        svcTasksDestroy.send({_method: "DELETE"});
    }
]]></mx:Script>
<mx:HTTPService id="svcTasksCreate" url="/tasks.xml" #7
    contentType="application/xml" resultFormat="e4x"
    method="POST" result="listTasks()">
    <mx:request>
        <task>
            <name>{nameTI.text}</name>
            <notes>{notesTA.text}</notes>
        </task>
    </mx:request>
</mx:HTTPService>
<mx:HTTPService id="svcTasksList" url="/tasks.xml" #8
    resultFormat="e4x" method="POST"/>
<mx:HTTPService id="svcTasksUpdate" resultFormat="e4x" #9
    method="POST" result="listTasks()"/>
<mx:HTTPService id="svcTasksDestroy" resultFormat="e4x" #10
    method="POST" result="listTasks()"/>
<mx:XMLListCollection id="tasksXLC"
    source="{XMLList(svcTasksList.lastResult.children())}/> #11
<mx:Panel title="Tasks List" width="100%" height="100%">
    <mx:DataGrid id="tasksDG" width="100%" height="100%"
        rowHeight="26" dataProvider="{tasksXLC}"
        change="task = XML(tasksDG.selectedItem)" #12
    <mx:columns>
        <mx:DataGridColumn dataField="name" headerText="Name"/> #13
        <mx:DataGridColumn dataField="notes" headerText="Notes"/>
        <mx:DataGridColumn headerText="" width="70"
            editable="false">
            <mx:itemRenderer> #14
                <mx:Component>
                    <mx:HBox horizontalAlign="center" paddingLeft="2"
                        paddingRight="2">

```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=440>

```

        <mx:Button label="delete" width="100%"
            click="outerDocument.destroyTask(XML(data))"> #15
        </mx:Button>
    </mx:HBox>
</mx:Component>
</mx:itemRenderer>
</mx:DataGridColumn>
</mx:columns>
</mx:DataGrid>
</mx:Panel>
<mx:Panel title="Task Editor" width="100%"> #16
    <mx:Form width="100%" height="100%">
        <mx:FormItem label="Name" width="100%">
            <mx:TextInput id="nameTI" width="100%"
                text="{task.name}"/> #17
        </mx:FormItem>
        <mx:FormItem label="Notes" width="100%">
            <mx:TextArea id="notesTA" width="100%" height="200"
                text="{task.notes}"/> #18
        </mx:FormItem>
    </mx:Form>
    <mx:ControlBar width="100%" height="100%">
        <mx:Button label="Create Task" width="34%" height="30"
            click="createTask()"/>
        <mx:Button label="Update Task" width="33%" height="30"
            enabled="{tasksDG.selectedItem != null}"
            click="updateTask(task)"/>
        <mx:Button label="Delete Task" width="33%" height="30"
            enabled="{tasksDG.selectedItem != null}"
            click="destroyTask(task)"/>
    </mx:ControlBar>
</mx:Panel>
</mx:Application>

```

Cueballs in code and text

- #1 vertical layout
- #2 list Tasks when the initialize event handled
- #3 create (the C of CRUD)
- #4 list (this is a Read, the R of CRUD)
- #5 update (the U of CRUD)
- #6 destroy (D is the Delete of CRUD)
- #7 HTTPService to create Tasks
- #8 HTTPService to list Tasks
- #9 HTTPService to update Tasks
- #10 HTTPService to destroy Tasks
- #11 bind source property
- #12 handle change event
- #13 column for name property
- #14 custom itemRenderer
- #15 call function in outerDocument
- #16 Task editor Panel
- #17 edit the Task name
- #18 edit the Task notes

As you might have guessed, this produces the screenshot shown in figure 1.1.

First, note that we use a vertical layout to make the components flow vertically (#1). Other choices are horizontal (for horizontal flow) and absolute (which we saw before). Second, we want to display all the tasks as soon as our application has finished initialization. This is accomplished by defining a callback function `listTasks()` for "initialize" property of the Application (#2).

Next we define a number of helper functions that wrap around various HTTP services. These functions encapsulate low level details such as URLs and parameters necessary to perform remote create, read,

Please post comments or corrections to the Author online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=440>

update and delete actions. More specifically, we define createTask (#3), listTasks (#4), updateTask (#5) and destroyTask (#6) functions.

We also need the actual Flex HTTP services that are responsible to communicating with remote Rails controllers. We define an HTTPService svcTasksList (#8), which does a GET to /tasks.xml (thus triggering the index action of the TasksController) and specifies a resultFormat of e4x so the result of the service can be handled with the new E4X XML API. We then take the lastResult of this service, which is an XML document, get its children (which is an XMLList of the tasks), and make this be the source of an XMLListCollection called tasksXLC (#11). We do this with a binding to the source attribute.

Similarly, we define svcTasksCreate (#7), svcTasksUpdate (#9) and svcTasksDestroy (#10) to be used for the other CRUD operations. Note that we can pass data to Rails via data bindings using curly braces {} inside XML (as shown in svcTaskCreate) or by sending form parameters that Rails would be expecting (as shown in the updateSelectedTask method). Note that for svcTasksUpdate and svcTasksDestroy we are not setting the url property statically, but instead dynamically setting it to include the id of the task we are updating or destroying.

Finally, a hack: you can't send HTTP PUT or DELETE from the Flash player in a web browser, so we need to fake it. Luckily, since you can't send PUT or DELETE from HTML in a web browser either, Rails already has a hack in place—we just need to know how to use it. Rails will look for a _method parameter in its params hash and, if there is one, use it instead of the actual HTTP method. So, if we do a form POST with a _method of PUT, Rails will pretend we sent an HTTP PUT. (If you're thinking that it's ironic that at the core of a "cleaner" architecture is a giant hack, well, you're not alone.) This _method can be added to a params Object (`params['_method'] = "PUT";`) or to an anonymous object (`svcTasksDestroy.send({_method: "DELETE"});`) If you're new to Flex, {} can be used for both anonymous object creation and for data binding. Think of an anonymous object like a hash in Ruby.

So far we have been writing code that is basically responsible for low level plugging between Flex HTTP services and Rails controllers. Before we move on, it is necessary to create a few visual components that actually display available tasks and allow the user to create, edit and delete tasks.

Let's start by creating a Panel that contains a DataGrid. The DataGrid (#12) uses tasksXLC collection (defined previously) as its data source. It also provides an in-place callback function for DataGrid "change" property. All this function does is set the task variable to the currently selected DataGrid item. This task is then used as an argument to updateTask and destroyTask functions. Our DataGrid defines a number of columns such as name (#13), notes and a special column that contains a delete button for each task (#15). Delete button is a complex property and as such cannot be displayed by the default DataGrid renderer. This is where custom item renderers come in handy (#14). Our item renderer contains a nested component declaration which defines a delete button for every task in the DataGrid. Instead of defining another function to delete a task inside the nested component we simply call the function we already defined in the parent document. In order to do this we need to prefix the function call with "outerDocument". This is a special property available inside the nested component to refer to functions and properties outside its own scope.

Next, we create another Panel which contains a form with input elements and control buttons to create, edit and destroy the currently selected task (#16). We are going to use a TextInput control for task name (#17) and a TextArea control for the task notes (#18).

Finally, let's run our new application by clicking the Run button. If you have a web browser open, a new tab will be created; if not, your default browser will be launched. You will see the Flex 3 application shown in figure 1.1. Play with it a bit, creating, updating and deleting tasks. Not bad.

1.4 Rewriting the To Do List using Ruboss

So far, we have seen how Flex and Rails work together with HTTPService. It looks pretty good—we built an entire application in 100 lines of MXML! This is true: it is good. However, it could be better. For

Please post comments or corrections to the Author online forum at

<http://www.manning-sandbox.com/forum.jspa?forumID=440>

example, there are HTTPServices mixed together with UI layout code. Also, the hacking to send form parameters should really be encapsulated somewhere. These types of problems are addressed to a degree in *Flexible Rails*, by using Cairngorm to structure the application and by using utility classes like ServiceUtils to hide the complexity. (You can download the complete *Flexible Rails* book source code <http://flexiblerails.com> to see what I mean.)

We will address these issues by rewriting the code using Ruboss. Why a rewrite instead of a refactoring? Frankly, because it will be faster and more impressive. I'm going to gloss over many of the details in this section; everything will be explained in far more detail in chapters 2 and 3.

First, let's create a brand new project called todoboss. We will use MySQL instead of SQLite this time, since the rest of the book uses MySQL on the server side.

```
~ $rails -d mysql todoboss
  create
  create  app/controllers
...
  create  log/development.log
  create  log/test.log
```

Next, we will install Ruboss from Google code:

```
~ $cd todoboss
~/todoboss $./script/plugin source http://ruboss.googlecode.com/svn/trunk/rails
Added 1 repositories.
~/todoboss $./script/plugin install ruboss
+ ./README
+ ./Rakefile
...
+ ./generators/rconfig/templates/ruboss.swc
...
installing multiscaffold into script folder
installing yamlscaffold into script folder
installing ruboss.yml into config folder, customize if necessary
~/todo $
```

This installs the Ruboss Rails plugin, which includes a compiled library of the Ruboss framework (ruboss.swc). If you want to depend on the source code, you can create a new Flex library project based on the framework source code at <https://ruboss.googlecode.com/svn/trunk/framework>. (If you don't know how to make a Flex library project, don't worry—it's only for the gurus who compile everything themselves.)

Next, we create a new Flex project with the Ruboss rconfig generator:

```
~/todoboss $./script/generate rconfig
  create  .flexProperties
  create  .actionScriptProperties
  create  .project
  create  html-template/history
  create  html-template/index.template.html
  create  html-template/AC_OETags.js
  create  html-template/playerProductInstall.swf
  create  html-template/history/history.css
  create  html-template/history/history.js
  create  html-template/history/historyFrame.html
  create  app/flex/todoboss/components
  create  app/flex/todoboss/controllers
  create  app/flex/todoboss/commands
  create  app/flex/todoboss/models
  create  app/flex/todoboss/events
  create  app/flex/todoboss/components/generated
  create  lib/ruboss.swc
dependency  rcontroller
  create  app/flex/todoboss/controllers/TodobossController.as
  create  app/flex/Todoboss.mxml
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

```
~/todoboss $
```

At this point, let's switch to Flex Builder and import the newly-created project, unchecking "use default location" as shown in figure 1.2.

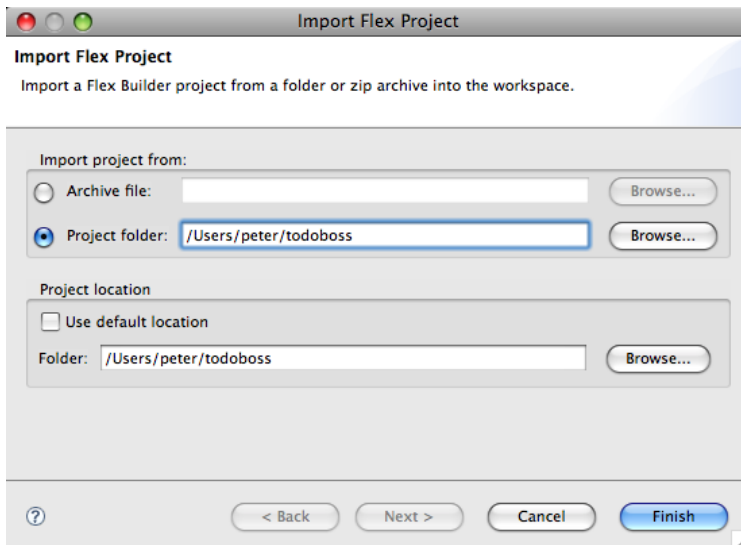


Figure 1.2 Importing the todoboss Project Created by the rconfig Generator

Click Finish to import the project.

Next, if you have a root password defined for your local MySQL database (like I do), you'll also need to run this next step:

```
~/todoboss $rake db:stage ADMINPASS=<mysql root password> USER=<application username>
PASS=<application password>
```

For example, if your MySQL root password is secret and you want a todoboss user with a password of fo000, run the following command:

```
~/todoboss $rake db:stage ADMINPASS=secret USER=todoboss PASS=fo000
(in /Users/peter/todoboss)
```

Updated config/database.yml and staged the database based on your settings

This rake task is part of Ruboss. It will create modify config/database.yml (and save original database.yml definitions into database.yml.sample) to use the user/password combination you've defined above. It will also do some other database stuff which will be redundant since we will be recreating our database momentarily.

Next, we run another Ruboss generator: rscaffold. We will use this generator to create the task model and view for us. This generator works just like the scaffold generator in Rails (in fact, it delegates to it for the Rails portion: the scaffold generator is a dependency), but it also generates Flex model and view code for us.

```
~/todoboss $./script/generate rscaffold Task name:string notes:text
dependency scaffold
  exists app/models/
  exists app/controllers/
  exists app/helpers/
  create app/views/tasks
  exists app/views/layouts/
  exists test/functional/
  exists test/unit/
  create app/views/tasks/index.html.erb
  create app/views/tasks/show.html.erb
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

```

create      app/views/tasks/new.html.erb
create      app/views/tasks/edit.html.erb
create      app/views/layouts/tasks.html.erb
create      public/stylesheets/scaffold.css
dependency  model
exists      app/models/
exists      test/unit/
exists      test/fixtures/
create      app/models/task.rb
create      test/unit/task_test.rb
create      test/fixtures/tasks.yml
create      db/migrate
create      db/migrate/001_create_tasks.rb
create      app/controllers/tasks_controller.rb
create      test/functional/tasks_controller_test.rb
create      app/helpers/tasks_helper.rb
route       map.resources :tasks
create      app/flex/todoboss/models/Task.as
create      app/flex/todoboss/components/generated/TaskBox.mxml
force       app/controllers/tasks_controller.rb
identical   app/models/task.rb
dependency  rcontroller
force       app/flex/todoboss/controllers/TodobossController.as
~/todoboss $

```

Now that we have created our model, let's run the rconfig generator again to set up the default test app (type y when prompted to overwrite Todoboss.mxml):

```

~/todoboss $ ./script/generate rconfig
identical   .flexProperties
identical   .actionScriptProperties
identical   .project
exists      html-template/history
identical   html-template/index.template.html
identical   html-template/AC_OETags.js
identical   html-template/playerProductInstall.swf
identical   html-template/history/history.css
identical   html-template/history/history.js
identical   html-template/history/historyFrame.html
exists      app/flex/todoboss/components
exists      app/flex/todoboss/controllers
exists      app/flex/todoboss/commands
exists      app/flex/todoboss/models
exists      app/flex/todoboss/events
exists      app/flex/todoboss/components/generated
identical   lib/ruboss.swc
dependency  rcontroller
identical   app/flex/todoboss/controllers/TodobossController.as
overwrite   app/flex/Todoboss.mxml? (enter "h" for help) [Ynaqdh] y
force       app/flex/Todoboss.mxml
~/todoboss $

```

Now is a good time to run the migrations and load the fixtures generated as a result of running r scaffold in the previous step. Run the following rake task:

```

~/todoboss $ rake db:refresh
(in /Users/peter/todoboss)
== 1 CreateTasks: migrating =====
-- create_table(:tasks)
--> 0.2302s
== 1 CreateTasks: migrated (0.2305s) =====

~/todoboss $

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

This rake task is defined by the Ruboss plugin as a little helper. It essentially amounts to running `rake db:drop` followed by `rake db:create`, `rake db:migrate`, and `rake db:fixtures:load` but saves a bit of typing along the way. (Note that this works on the current database defined in `RAILS_ENV`, for example an environment of development means the database is `todoboss_development`.) Having this task is really handy for following along with a book like this, since it lets you start at any chapter and follow along after running one command.

Alternatively, if you like typing, you can run the following two commands instead:

```
~/todoboss $rake db:migrate
~/todoboss $rake db:fixtures:load
```

At this point, switch back to Flex Builder and do Project -> Clean, and do a clean build on the todoboss project:

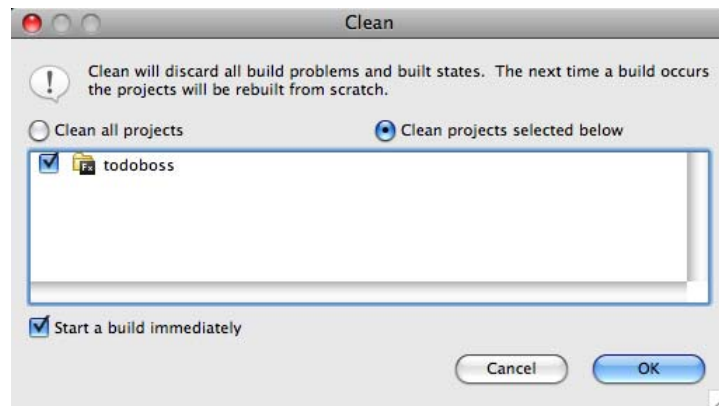


Figure 1.3 Doing a Clean Build in Flex Builder

Next, start your server:

```
~/todoboss $./script/server
=> Booting Mongrel (use 'script/server webrick' to force WEBrick)
=> Rails application starting on http://0.0.0.0:3000
...
** Mongrel 1.1.4 available at 0.0.0.0:3000
** Use CTRL-C to stop.
```

Finally, run the Todoboss application. You will see the following screen:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

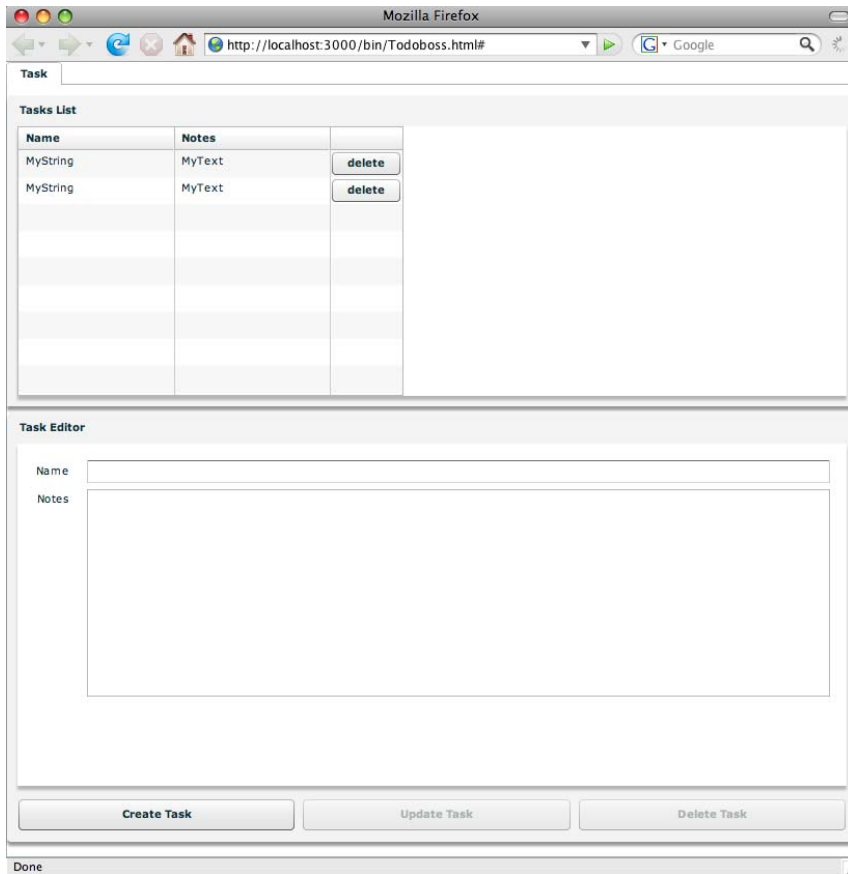


Figure 1.4 The Totally Generated Todoboss Application

Now *that's* a scaffold!

The application is fully functional; you can create, update and delete tasks. Note that it is slightly uglier than the one we wrote ourselves, since the DataGrid uses fixed width columns. (This is because we want the DataGrid to be functional when a generated model has, say, 15 attributes.)

Let's take a quick tour of the generated Flex code, so that you can see what is there. I'm not going to explain everything in too much detail, since I'd rather do that in chapters 2 and 3 when we are building a real application. However, it's useful to see the code, so that you have a rough idea what's there.

Switch to Flex Builder and take a look in the Flex Navigator. You will see the structure shown in Figure 1.5.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

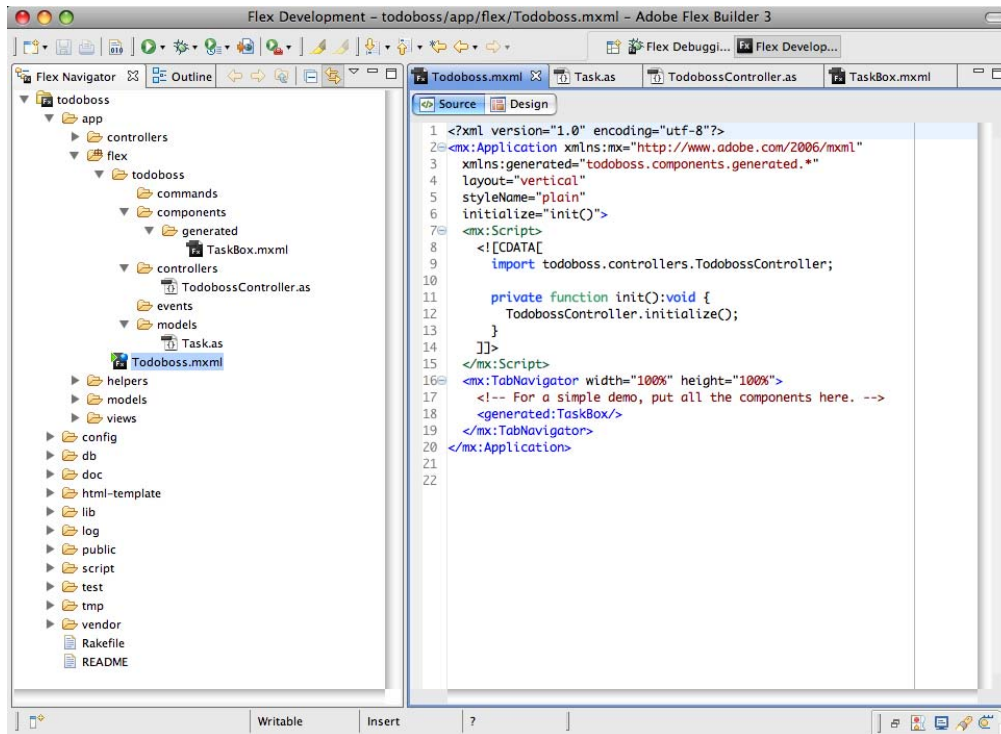


Figure 1.5 The Generated Todoboss Application Code

The rconfig generator created the standard Ruboss directory structure inside app/flex, including the name of the application (todoboss) as the package, and the commands, components, components/generated, controllers, events and models directories. We'll look at these in more detail in the next iteration, but for now just know that this is the Rails philosophy of Convention Over Configuration applied to Flex development.

Next, let's take a look at the generated code. We'll start with the Task model, shown in listing 1.10.

MY CODE HAS GENERATED COMMENTS. WHERE DID THE COMMENTS GO?

I'm deleting the comments, since Manning books don't have comments in the code (the book text is the comment).

Listing 1.10 app/flex/todoboss/models/Task.as

```

package todoboss.models {                                     #1
    [Resource(controller="tasks")]                           #2
    [Bindable]                                              #3
    public class Task {
        public static const LABEL_FIELD:String = "name";    #4

        public static const NONE:Task = new Task("- None -"); #5

        public var id:int;

        public var name:String;

        public var notes:String;

        public function Task(                                #6
            name:String = "",
            notes:String = ""
        ) {

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>


```

private function onTaskSelect():void { #5
    editedTask = Task(tasksDataGrid.selectedItem).clone();
}

private function onTaskCreate(task:Task):void { #6
    editedTask = new Task;
}

private function onTaskUpdate(task:Task):void { #7
    tasksDataGrid.selectedItem = task;
    editedTask = task.clone();
}

private function onTaskDestroy(task:Task):void { #8
    onTaskCreate(task);
}
]]</mx:Script>
<mx:Panel id="tasksPanel" title="Tasks List"
width="{width}" height="300">
<mx:DataGrid id="tasksDataGrid" #9
    height="100%"
    horizontalScrollPolicy="auto"
    dataProvider="{Ruboss.models.index(Task)}" #10
    change="onTaskSelect()">
<mx:columns>
    <mx:DataGridColumn dataField="name" headerText="Name"
        width="150" minWidth="100"/>
    <mx:DataGridColumn dataField="notes" headerText="Notes"
        width="150" minWidth="100"/>
    <mx:DataGridColumn headerText="" width="70"
        minWidth="70" editable="false" dataField="name">
        <mx:itemRenderer>
            <mx:Component>
                <mx:HBox horizontalAlign="center" paddingLeft="2"
                    paddingRight="2">
                    <mx:Button label="delete" width="100%"
                        click="outerDocument.destroyTask()">
                    </mx:Button>
                </mx:HBox>
            </mx:Component>
        </mx:itemRenderer>
    </mx:DataGridColumn>
</mx:columns>
</mx:DataGrid>
</mx:Panel>
<mx:Panel title="Task Editor" width="100%" #11
height="{height - 320}" verticalScrollPolicy="auto">
<mx:Form width="100%" height="100%">
    <mx:FormItem label="Name" width="100%">
        <mx:TextInput id="nameTextInput" width="100%"
            text="{editedTask.name}"/>
    </mx:FormItem>
    <mx:FormItem label="Notes" width="100%">
        <mx:TextArea id="notesTextArea" width="100%"
            height="200" text="{editedTask.notes}"/>
    </mx:FormItem>
</mx:Form>
<mx:ControlBar width="100%" height="100%">
    <mx:Button label="Create Task" width="34%" height="30"
        click="createTask()"/>
    <mx:Button label="Update Task" width="33%" height="30"
        enabled="{tasksDataGrid.selectedItem != null}"
        click="updateTask()"/>

```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=440>

```

        <mx:Button label="Delete Task" width="33%" height="30"
            enabled="{tasksDataGrid.selectedItem != null}"
            click="destroyTask()"/>
    </mx:ControlBar>
</mx:Panel>
</mx:VBox>

```

Cueballs in code and text

```

#1 the Task being edited
#2 create new Task
#3 update the editedTask
#4 delete the editedTask
#5 edit a copy of the selected Task
#6 reset editedTask on create
#7 clone editedTask on update
#8 reset editedTask on destroy
#9 the DataGrid of Tasks
#10 bind to result of Ruboss.models.index
#11 the Task editor

```

This early in the book, we don't want to get bogged down in detail. (This is iteration 1, after all.) Briefly, however, we have an instance variable for the edited task (#1), a bunch of methods (#2 - #8) which create, update and destroy tasks and handle the results of these operations, and generated UI components including a DataGrid (#9) for all (#10) the Tasks and a Panel (#11) which lets you edit the editedTask. Note that even though you have never seen Ruboss before, the code is pretty readable—and a model of brevity and elegance, in terms of generated UI code. (Being able to do complete CRUD in about a hundred lines of readable, nicely formatted, totally generated Flex code for an arbitrary model object is very impressive.)

Before moving on, I want to highlight one of the key features of the Ruboss framework: Notice that there is no **HTTPService** mentioned anywhere—XML over HTTPService is the default transport mechanism used by Ruboss, but not the only choice. We could change the transport mechanism to, say, AMF without changing one line of the code in this class.

Next, the `TodobossController`; see listing 1.12.

Listing 1.12 `app/flex/todoboss/controllers/TodobossController.as`

```

package todoboss.controllers {
    import todoboss.models.*;
    import todoboss.commands.*;

    import org.ruboss.Ruboss;
    import org.ruboss.controllers.RubossCommandController;

    public class TodobossController extends
        RubossCommandController { #1
        private static var controller:TodobossController;

        public static var models:Array = [Task]; #2

        public function TodobossController(
            enforcer:SingletonEnforcer,
            extraServices:Array,
            defaultServiceId:int = -1) {
            super([] /* Commands */,
                models, extraServices, defaultServiceId);
        }

        public static function get instance():TodobossController { #3
            if (controller == null) initialize();

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=440>

```

        return controller;
    }

    public static function initialize(
        extraServices:Array = null,
        defaultServiceId:int = -1):void {
        controller = new TodobossController(
            new SingletonEnforcer,
            extraServices,
            defaultServiceId);
        Ruboss.commands = controller;
    }
}

class SingletonEnforcer {}

```

Cueballs in code and text

#1 extend RubossCommandController
#2 create models Array containing Task class
#3 Singleton stuff

This class, a Singleton (#3) which extends RubossCommandController (#1) and does the main Ruboss framework magic, hooking up the commands and models (#2). We'll cover it in greater depth in iterations 2 and 3.

Finally, the Todoboss application itself:

Listing 1.13 app/flex/Todoboss.mxml

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:generated="todoboss.components.generated.*"
    layout="vertical"
    styleName="plain"
    initialize="init()">
    <mx:Script>
        <![CDATA[
            import todoboss.controllers.TodobossController;

            private function init():void {
                TodobossController.initialize();           #1
            }
        ]]>
    </mx:Script>
    <mx:TabNavigator width="100%" height="100%">
        <!-- For a simple demo, put all the components here. -->
        <generated:TaskBox/>                               #2
    </mx:TabNavigator>
</mx:Application>

```

Cueballs in code and text

#1 call the TodobossController initialize method
#2 include generated TaskBox

This class is the main Application. It calls the TodobossController initialize method (#1) in its init method (triggered when its own initialize event is broadcast). It has an mx:TabNavigator with a tab for the generated TaskBox (#2). This is a nice way to create a stub UI: one tab for each model. Note that this code was generated when we ran rconfig the second time—that's why we did it. Yes, we could have typed

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=440>

it in, but we're being lazy. As you'll see in the next iteration this is even nicer with a more complex data model.

1.5 Summary

Flex and Rails are a good combination, and the Ruboss framework makes them an extremely well integrated combination. While Ruboss does not cover all the use cases you will need, it does not attempt to. If you need to do something other than RESTful CRUD, you can do so by writing your own commands—and, as you will see, the Ruboss framework makes it easy for you to bypass it and hook up custom commands with fewer steps than are required when using Cairngorm. Finally, using Ruboss makes it easier to change the transport protocol from XML over HTTPService to AMF (or JSON over HTTPService or YAML over HTTPService), and it also makes it extremely easy to get going doing CRUD with a local SQLite database as an AIR application!

In the next chapter, we will go deeper into the Ruboss framework, starting by generating a much more complex and realistic application—all from one YAML file.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=440>