The Pragmatic Programmers

Rails for .NET Developers

Jeff Cohen and Brian Eng

Edited by Susannah Davidson Pfalzer

The Facets of Ruby Series

What readers are saying about *Rails for .NET Developers*

Eng and Cohen, like a modern-day Lewis and Clark, have blazed a trail that .NET developers can follow to the new frontier. If you are a .NET developer and are considering moving to Ruby on Rails, then this book is the place to start.

- **James Avery**
  President and CEO, Infozerk, Inc.

If you're ready to make the rewarding trip from .NET to Rails, this book will give you the road map you need.

- **Mike Gunderloy**
  Former .NET Developer, http://afreshcup.com

This book will be a tremendous aid to anyone making the transition from .NET to Ruby on Rails. All the major topics a new Rails developer should become familiar with are covered in great detail.

- **Michael Leung**
  Lead Developer, Urbis.com

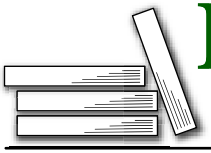
Jeff and Brian have done a wonderful job of explaining Ruby on Rails to .NET developers in this book...but there's more value here than just learning a hot technology. Jeff and Brian show you how Ruby on Rails can make you a better developer no matter what platform you use, as well as how it can influence how you design and write web applications.

- **Brian Hogan**
  Principal Consultant, New Auburn Personal Computer Services LLC

Rails for .NET Developers

Jeff Cohen
Brian Eng

The Pragmatic Bookshelf



Pragmatic Bookshelf

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 Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Copyright © 2008 Jeff Cohen and Brian Eng.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.

ISBN-10: 1-934356-20-4

ISBN-13: 978-1-934356-20-3

Contents

Acknowledgments

We are very grateful to those who helped make this book a reality. Susannah Pfalzer's time, patience, and insight were enormously valuable to us time and again during the writing of this book. We also had great help from our reviewers—Geoffrey Grosenbach, James Avery, Michael Dwan, Dianne Siebold, Mike Gunderloy, Michael Leung, Scott Hanselman, and Ron Green—and our publishers, Dave Thomas and Andy Hunt. They not only helped us with technical details, but their thoughtfulness and feedback elevated our writing to the next level. We would also like to thank Michael Manley, Scott Epskamp, and the rest of Jeff's colleagues at Leapfrog Online, whose understanding and flexibility helped make this book possible.

Jeff says: I could not have written this book without the encouragement and support of my beautiful wife, Susannah, and our two wonderful children, Laura and Emily. I also thank my parents, Bill and Marilyn Cohen, who first taught me how to read and write—two skills that came in quite handy.

Brian says: I'd like to thank my wonderful wife, Erika, for her support and inspiration—but especially for caring for our two baby girls, Katie and Abbie, all day and night while I've been working on this book! Also thanks to my mom and dad, Alfred and Patricia Eng, for the Apple II that got all this computer stuff started.

Preface

As a .NET developer, you've probably heard the buzz surrounding Ruby on Rails. It's true: Rails enables you to create database-driven web applications with remarkable speed and ease. And like many other open source projects, Rails has been most easily adopted by individuals and organizations already immersed in the open source community. That means, as a Microsoft developer, you face unique challenges learning not just Rails but all the open source technologies that go along with it. This book will be your guide as you navigate this new terrain.

For a .NET developer, learning Rails is as much about the cultural and philosophical shifts in thinking as it is about the technical learning curve. In this book, we hope to break down some of these barriers for you. We have learned a lot of valuable lessons from Rails that we've applied to our .NET development too; if you take anything away from this book, it will be a new way of thinking about software development—the Rails way.

What's in This Book

To get things going, we'll introduce the Rails development environment and the core set of tools you'll need to be an effective Rails developer from the very beginning. Nothing is better to get your feet wet than to actually write some code and build a small application, so that's exactly what we'll do.

Becoming a skilled Rails developer is all about learning Ruby. So, we'll take an in-depth look at the Ruby language and how to understand it from a Microsoft developer's perspective. We'll ease you into it and gradually move into more advanced topics.

Once we've established a solid base of Ruby knowledge, we'll dive head-first into the Rails framework. We'll address common programming

We will look at almost all aspects of the framework from at least a high level, from data access to the controller to the presentation layer, including Ajax.

Throughout the book we'll be emphasizing agile development practices, including a tour of basic unit testing approaches for Rails applications. You'll also get a head start on integrating your new Rails applications with existing .NET web services, as well as learn how you can write .NET programs to use Rails web services.

Finally, we'll wrap up by talking about the Rails ecosystem, its philosophies and tools, and how (and why) Rails is quickly becoming one of the frameworks of choice for the truly agile web developer.

Who This Book Is For

This book is written for experienced .NET developers who are interested in exploring Rails for web development. A strong background with web development is not necessary, but we're assuming you have at least some Microsoft-centric programming experience. We are .NET developers, so we've geared the material in this book to suit the unique needs of those who think the way we do about programming problems.

The hurdles of learning a new language and a new framework can be daunting. Once we get you up and running, we'll start with a gentle introduction to the object-oriented Ruby language, including examples and direct comparisons with C#, so you'll quickly feel at home writing Ruby code for the first time.

About the Environment and Version Requirements

Most .NET developers we know write software on a Windows PC. As such, we've written the code examples and most of the walk-through console commands and tools on the Windows platform (some examples also show Mac/Linux-style shell sessions for those developing or deploying on those platforms). The Rails community is well-known for being Mac-friendly, but it's just as easy to develop Rails applications on Windows—we'll highlight the differences along the way. The majority of the screenshots are from our Windows XP and Windows Vista development environments; the web application screenshots were taken from Internet Explorer 7 and Firefox 2.0.

Most of the code examples compare .NET/ASP.NET 3.5 with Rails 2.1. The Ruby examples will run on most newer Ruby versions, but version 1.8.6 or newer is recommended. Because Rails is ever-changing technology, we've made our best effort to present the examples using the latest and greatest, Rails 2.1. However, that means many of the code examples will not work in earlier versions of Rails.

Most of the .NET code is vanilla, out-of-the-box ASP.NET 3.5. Although numerous plug-ins and third-party assemblies exist that give .NET developers some of the same capabilities as Rails (we explore some in Chapter 13, *Inspired by Rails*, on page 252), we won't cover them in depth in this book.

Conventions

There is a lot of code in this book, and to make it easier to distinguish between the .NET and Ruby/Rails code, we've marked snippets of code with an icon that corresponds to the language in which it's written.

Download preface/hello.cs

```
public void Hello()
{
  Console.WriteLine("Hello");
}
```

Download preface/hello.rb

```
def hello
  puts 'hello'
end
```

Online Resources

All code samples are available for download online.[1] If you're reading this book in PDF, you can access the downloadable code samples directly by clicking the little gray boxes before code excerpts. In addition, we encourage you to interact with us by participating in our forums.[2]

Let's get started!

1. http://www.pragprog.com/titles/cerailn/source_code

Part I

Hello, Rails

Chapter 1

Getting Started with Rails

Grand adventures and trips to the grocery store begin the same way. You step outside your door and close it behind you. Soon your intentions will become clear: you're either just going to get some more milk or on your way to some other part of the planet.

As software developers, we work with the same technologies day after day. We simply tend to stick with what we already know. The best software developers, however, make time to explore new ideas and learn new things. These journeys enrich their understanding of the art and science of software engineering.

We all say we want to try new things, but actually doing so is another story. It's the difference between glancing at an adventure guide and actually booking a ticket on the next flight out.

This book is for those .NET developers who are ready for their next adventure.

Why Rails?

Rails offers attractive benefits to all website developers. Let's highlight some of the most-often cited reasons that developers are trying Rails.

The first is the issue of cost. Already got a computer? Then you're in luck, because everything else you need can be had for free. The Ruby interpreter, the entire Rails framework library, a wide choice of good code editors, various deployment tools, and industrial-strength database engines such as MySQL are all available at no cost to the developer. Even free online support can be found within the commu-

Second is the notion of writing beautiful code. Ruby is a powerful and incredibly flexible programming language for web application development. Most C# and VB .NET developers will find themselves writing less code to accomplish the same thing in Ruby. Ruby code tends to be more readable, easier to test, and easier to change than the statically typed .NET languages.

Next up is Rails itself. Rails was built on the premise that the vast majority of web applications are simply HTML forms on top of a database, and thus it makes these types of applications very easy to build. If you've never had a chance to build your code on top of a Model-View-Controller framework before, you're going to love it. (In fact, Microsoft has recently added an MVC project type to ASP.NET 3.5 projects as its response to the success of MVC frameworks like Rails. We'll take a peek at it in Chapter 13, *Inspired by Rails*, on page 252.) The separation of concerns brings both simplicity and flexibility to your application's internal architecture. Although many aspects of Rails are configurable, Rails requires no XML configuration files. The tight integration between view templates and Ruby code also makes it much easier for website designers and programmers to collaborate on the same project.

Rails also appeals to developers who are well-versed in "agile" techniques. The built-in support for unit testing, refactoring, and incremental database modeling provides a simpler, faster path to large-scale application development.

The Ruby community has experienced a recent explosion of alternate Ruby implementations that allow integration with non-Ruby environments and code libraries. JRuby and IronRuby are the most prominent examples that provide seamless integration with Java and .NET code, respectively. The once-high bar for Rails adoption in "the enterprise" is getting lower and lower.

Finally, it's been our experience that developing applications with Rails is just plain *fun*. There's a particular rhythm and flow that occurs with Rails development that most developers enjoy. The Rails framework is geared to generate visible results quickly, leading to the pattern of "small victories" often cited as a key reason for a project's success.

These are all great and valid reasons why Rails is so compelling. But even seasoned .NET developers willing to give Rails a shot can find the experience disorienting enough to give up before they can really ever get started. Let's talk about these challenges and how we're going to

Culture Shock and Its Treatment

.NET developers who take their first step into the Ruby on Rails framework are joining a new developer community. As a result, they often feel like they've been transported to a foreign land for the first time. In this new land, there seem to be some very happy people bustling about, but the surroundings are a bit strange. Local customs are unfamiliar. Everyone is speaking a different language than the one they know. It can add up to a kind of culture shock that can be a bit discouraging at first. As a result, some first-time visitors to the land of Rails quickly give up and take the first flight back to a more familiar place.

This book will be your personal tour guide to help you get your bearings in the world of Rails. Pretty soon, you won't just be coping with Rails; you'll be a very happy programmer, bustling around in your own way as you begin to discover the joy of developing web applications in Ruby on Rails.

Let's start by addressing the most obvious change: the Ruby programming language. Learning Ruby isn't just a matter of memorizing new keywords, although you will have to do that. It's also not just a matter of learning a new class library, although you will have to do that, too. It's like memorizing the vocabulary of a new spoken language. You may be able to say the words, but you're not going to sound like a native speaker until you adopt the particular idioms, nuances, and inflections that lead to true language fluency. Writing excellent Ruby code is similar: there is certainly some new "vocabulary" to learn, but the idioms, style, and nuances are the key to writing natural-looking Ruby programs.

Along with a new programming language comes a new set of development tools. If you are a Visual Studio user who never leaves the IDE, you may not realize the variety of tools you've been using:

- A code editor
- A debugger
- A unit testing framework[1]
- A build system
- A distribution/deployment packaging tool

1. At the time of this writing, only the paid editions of Visual Studio Team System provide any support for integrated unit testing. However, many Visual Studio users have long used tools like NUnit to fill this need.

One day, Visual Studio may support the writing of Ruby on Rails applications directly in the IDE. Some products such as Ruby in Steel[2] integrate into Visual Studio to provide an environment for building Rails applications inside the IDE, and the IronRuby project (see Section 13.1, *IronRuby*, on page 253) will also provide better support for Rails. In the meantime, many developers choose to use alternative coding environments on Windows.

Many developers, however, switch to a new operating system altogether. Rails, like most open source projects, began life on Linux. The best tools and support thrive primarily on Mac and Linux systems. Since you will likely deploy your Rails applications onto Linux servers, there's an advantage to building Rails applications on a *nix-based OS as well. Developing on Windows but then deploying to Linux can complicate the develop-test-deploy cycle. On the other hand, switching development platforms just might not be an investment you're ready to make right off the bat, and that's precisely why we'll show you how to develop Rails apps on Windows throughout this book.

Getting help when you need it is also going to be different from what you're accustomed to—there's nothing at the moment that compares to the vast repositories of information like the one found at MSDN. Although you can find basic information on the Ruby and Rails home pages,[3] you can find in-depth user-to-user support by visiting one of the online Google Groups or IRC channels.[4] Whether you need an answer to a specific question or just want to keep up with the latest news, they are both invaluable in helping you get the information you need. Lastly, your best friend may well be the official Ruby on Rails documentation, which ships with Rails and is also available online in a variety of different formats.[5]

That leads to the final, and perhaps most exciting, step you can take to smooth your transition into your new community: participate! Once you've learned something about Rails, no matter how small, it's immediately time to give back. Find questions on the Rails Google Group from newcomers that you're able to answer, and help them out. And remember that Rails is an open source project. That means absolutely anyone is allowed submit code patches for consideration. Whether you'd like to

2. http://sapphiresteel.com/
3. http://www.ruby-lang.org/ and http://www.rubyonrails.org
4. Start with http://groups.google.com/group/rubyonrails-talk or #rubyonrails on IRC.

Getting Help Online (Or, Help Them Help You)

We'd like to take a moment to provide a public service announcement. It's important to learn the etiquette that's expected when using the various Google Groups and IRC channels that are open to all Ruby on Rails developers.

Please remember that you've stepped into a land of volunteerism, and etiquette is paramount. Unlike getting support from Microsoft for your VB .NET questions, those who participate in the various online Ruby on Rails communities do so for free and on their own time.

Before asking a question, do your homework first. Google is your friend. It's unlikely that you're the first one to ask any particular question, so search for the answer first.

If you can't find what you're looking for, then by all means feel free to participate in the discussions. Just be sure to mention what you've tried so far, and reduce any code samples to the minimum necessary to demonstrate the issue you're having. And when you do get an answer, be sure to thank those who took the time to help you.

We now return you to your regularly scheduled programming.

fix a bug, add a feature, or improve the documentation, there are many ways to lend a hand. You're encouraged to get personally involved in the continued success of Rails.

Let's Get This Party Started

Remember when you were a kid and someone in your house was having a birthday? Before the guests came over, the cake would be all set out, but it wasn't time to eat it yet. When Mom wasn't looking, what did you want to do? Get a taste of some frosting, of course!

The rest of the this book will be like that birthday cake: there will be a lot of layers to see, some cool decorations to admire, and more than one fire-breathing implement placed on top. The party will really get started in Chapter 2, *Switching to Ruby*, on page 31. But who can wait that long? Let's get a taste right now.

To get things rolling, we'll show you how to get a Ruby and Rails development environment set up on your Windows machine. It's a fairly straightforward process that takes just a few minutes. In fact, building a Rails development environment from scratch is arguably easier on Windows than on any other platform.

The ingredients for creating a productive Rails development environment are Ruby, Rails, a database engine, and a few supporting tools. In this chapter, we'll walk through getting all these necessary components of your development environment installed. Then, we'll get right to work and exercise your new developer's toolset by building a small Rails application.

Installing Ruby and Rails

The first thing we'll look at is how to install the Ruby language, followed by the quick-and-easy installation of the Rails framework.

Instant Rails

Instant Rails,[6] written by Curt Hibbs, has long been the Windows tool of choice for getting a Rails development environment up and running quickly. However, there is a lot of valuable information to be learned about the inner workings of Ruby and Rails by going through the installation process ourselves, so that's what we're going to do for the remainder of this section.

The Ruby Language

Before we can install Rails, we'll need to install the Ruby language and interpreter. For this, the Ruby One-Click Installer (OCI)[7] does the trick. After downloading and stepping through the installation, you can make sure that Ruby is indeed installed by entering a simple command at the command prompt:

```
C:\> ruby -v
```

And this should yield something like the following:

```
ruby 1.8.6 (2007-09-24 patchlevel 111) [i386-mswin32]
```

6. http://instantrails.rubyforge.org
7. http://rubyforge.org/projects/rubyinstaller/

RubyGems

The Ruby OCI includes a Ruby-specific package and software distribution system called RubyGems, which we cover in much greater depth in Section 12.1, *Getting to Know RubyGems*, on page 232. You can think of it being similar to Windows Installer, but for Ruby programs. It is accessible via the gem command, and we'll use it now to install Rails.

Git

Git[8] is a distributed *version control system* that is quickly gaining popularity in the open-source world, especially with Ruby developers. Many of the popular Ruby-based open-source projects, including Rails and many of its plugins, are now hosted using Git, so installing Git on our development machine is definitely recommended.

The easiest way to get Git running on Windows is to install msysgit[9] with its default configuration. Once it's installed, we can simply right-click on any directory in Windows Explorer to open up an interactive shell in which we can run Git commands.

Rails

The gem command can take a variety of parameters, which we will dig deeper into in Section 12.1, *Getting to Know RubyGems*, on page 232, but for now, we're going to simply tell the gem command that we want to install a package called rails:

```
C:\> gem install rails
```

Rails actually consists of several smaller packages—ActiveRecord, ActiveSupport, ActionPack, ActionMailer, and ActionWebService—all of which are installed when you install Rails using the gem command. We'll take a much closer look at each one of these packages in Chapter 4, *A Bird's Eye View of Rails*, on page 75, but for now, if everything goes well, we'll see some output that lets you know that each one of these packages was successfully installed and that, in addition, all the documentation for these gems are installed. To make sure everything worked, you can test your Rails installation by issuing this command:

```
C:\> rails -v
```

8. http://git.or.cz/
9. http://code.google.com/p/msysgit/

Code Editing on Windows

Code editors for Rails development on Windows are not nearly as mature or full-featured as what you may be used to with Visual Studio. However, you have some several solid choices, ranging from simple text editors to more comprehensive integrated development environments. As you become more proficient at developing Rails applications, you'll find that the features of your editor/IDE are going to be less important than simply finding an environment in which you are comfortable writing code and navigating through the Rails directory structure. That said, here are several good choices for the Windows platform:

E Text Editor A text editor that is similar in spirit to TextMate for Mac OS X.

Ruby in Steel A Rails development plug-in for Visual Studio.

Aptana IDE (RadRails) Formerly a stand-alone application, RadRails is now a plug-in for the Aptana IDE and is a mature Rails development environment that supports Ruby and Rails out of the box, including syntax highlighting and basic code completion.

WordPad If you are on Windows, you've got it. Simple and effective.

Notepad++ Like WordPad but includes simple syntax highlighting and an integrated file viewer.

Scite Comes with the Ruby OCI and provides very good syntax highlighting for many languages, including Ruby.

Ultraedit A more full-featured text editor with support for Ruby syntax highlighting.

And you should see something like this:

```
Rails 2.1.0
```

If you see this output, congratulations! You've successfully installed Ruby on Rails.

Connecting to a Database

Rails is great at a variety of web-based applications, but its sweet spot is the rapid development of database-backed websites. So naturally, we'll need to install and set up a database. Rails supports the use of most popular relational database systems, so what you decide upon for your projects is largely based on what you're comfortable using. MySQL is free and powerful, so Rails developers who don't otherwise have a preference tend to choose it for production use. SQL Server is also supported for those of us who are comfortable with it from the .NET world. However, for development and testing purposes, Rails 2.*x* uses SQLite, a simple file-based RDBMS, by default. Since it's supported out of the box, most of the examples in this book will assume a SQLite 3[10] database.

Installing SQLite 3 on Windows

Although support for SQLite 3 is built in to Rails, it does not come preinstalled with any version of Windows (like it does with Mac OS X 10.5). Thankfully, it's straightforward (and free) to obtain and install it on your Windows box. Once you've downloaded and unpacked the latest version available on the SQLite website, copy sqlite3.dll and sqlite3.def to your C:\windows\system32 directory. The last thing to do is to install the supporting files that will allow access to the SQLite 3 API from Ruby. This is all nicely packaged up in a single gem:

```
C:\> gem install sqlite3-ruby
```

Just to make sure, let's test that SQLite 3 was successfully installed.

```
C:\> sqlite3
SQLite version 3.5.4
Enter ".help" for instructions
sqlite>
```

If your output looks like this, you are all done with your database set-up! Type .exit to leave the SQLite 3 shell, and let's get to building your application.

10. http://www.sqlite.org/

Instant Gratification—Your First Rails App

The best way to "get it" when starting out with Rails is to just go for it by building a simple application and playing around with its capabilities. In this section, that is exactly what we'll do. We'll build a basic application that's going to help us keep track of all the books (yes, paper!) in our library.

We're going to write almost the entire application using just a few simple commands. You'll have to see it to believe it, so let's fire up a command prompt, cd to where you'd like your application to live (we've chosen the C:\dev directory), and create a new Rails application:

```
C:\dev> rails book_tracker
C:\dev> cd book_tracker
```

The rails command results in the default Rails directory structure being created. Like File > New Project in Visual Studio, it creates a shell of an application for us. Let's go ahead and list the directory contents to see exactly what was created:

```
C:\> dir
app
components
config
db
doc
lib
log
public
script
test
tmp
vendor
```

We'll go into the Rails directory structure in further detail later, but for now, let's focus on the two directories where we'll likely spend the most time. The app directory contains all your main application code, broken down into four subfolders: controllers, helpers, models, and views. And the config directory contains our application's configuration. Among other things, this application configuration describes how we'll connect to our database.

Once we've created the Rails project, it's time for some Rails magic.

Scaffolding—An App in One Line

Scaffolding, in Rails terms, is a lot like real-world scaffolding—it is boilerplate code to help keep our application in place while we're building the real production-quality code behind it. We can put it up very quickly, and when we're done, we should tear it down so that it doesn't get in the way. Let's put the scaffolding up now:

```
C:\dev\book_tracker> ruby script/generate scaffold book title:string
 author:string on_loan:boolean
      exists  app/models/
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/books
      exists  app/views/layouts/
      exists  test/functional/
      exists  test/unit/
      create  app/views/books/index.html.erb
      create  app/views/books/show.html.erb
      create  app/views/books/new.html.erb
      create  app/views/books/edit.html.erb
      create  app/views/layouts/books.html.erb
      create  public/stylesheets/scaffold.css
  dependency  model
      exists    app/models/
      exists    test/unit/
      exists    test/fixtures/
      create    app/models/book.rb
      create    test/unit/book_test.rb
      create    test/fixtures/books.yml
      create    db/migrate
      create    db/migrate/20080722191828_create_books.rb
      create  app/controllers/books_controller.rb
      create  test/functional/books_controller_test.rb
      create  app/helpers/books_helper.rb
       route  map.resources :books
```

The generate command, in its simplest form, takes two parameters, the first being what you'd like to generate—in this case a scaffold—and the second being the name of the new class. By convention, the scaffold generator expects the singular form of the resource you're trying to create, so we've passed in the singular book argument to the command.

In addition, we've also told the generator about what fields a book has. We've told it that each book will have two string fields, title and author, and a boolean field that indicates whether we've lent that book to a friend.

All the scaffold generator command (or any generator command, for that matter) does is create a bunch of files for us. We could have created these files ourselves, but this is a whole lot easier! We'll get into what all these files are for in a moment, but for now, let's concentrate on the db/migrate/20080722191828_create_books.rb file. This file is what's known as a *migration*.

Versioning the Database with Migrations

A *migration* is a Ruby script that uses a very simple *domain-specific language* (DSL) for manipulating databases. As .NET developers, we might be accustomed to inventing our own ways of creating and versioning database schemas. If we were developing an app with .NET and SQL Server, a simple example might go something like this:

1. When developing the first cut of your application, create an initial database creation script by using SQL or by using a graphical tool such as SQL Management Studio and then dumping the schema to a text file.
2. While developing, make changes to the database schema through a similar process—using various SQL scripts—sharing throughout with team members so they can keep up-to-date.
3. After our application is deployed to staging or production environments, use additional SQL scripts we've created to keep the database schemas on your development environment in sync with these other environments in our IT infrastructure.

Migrations are simply the Rails way of doing the same thing. Except that instead of using SQL, it's all written in Ruby. This approach has a couple of benefits:

- Since our app and database manipulation are all written in the same language, there's very little context switching or deep knowledge about database-specific intricacies necessary to be successful building your app.
- Rails does all the low-level SQL for you, and it is completely platform-agostic. We can easily switch from MySQL to Postgres to SQL Server if we want and even have a different database platform per environment. Imagine developing on a Mac with SQLite 3, staging to a Linux box with MySQL, and deploying to a Windows server with SQL Server for production (not that we recommend

Let's open db/migrate/20080722191828_create_books.rb and take a peek at what a migration looks like:

Download instant/20080722191828_create_books.rb

```
class CreateBooks < ActiveRecord::Migration
  def self.up
    create_table :books do |t|
      t.string :title
      t.string :author
      t.boolean :on_loan
      t.timestamps
    end
  end

  def self.down
    drop_table :books
  end
end
```

The fields we passed as parameters to the scaffold generator command are already in this migration file. We are free to modify them in any way at this point; the actual changes to the database schema have not been made yet.

The name of the file is crucial—well, at least the number at the beginning is. This is a time stamp of when the migration was created, and it represents the database version. As we add more migrations to our application, that number will increase; that's how Rails knows the order in which to execute them.

A migration class has two methods: self.up and self.down. The self.up method tells Rails what to do when migrating up; likewise, the self.down method gets executed when rolling the database back.

Remember, the generate command creates a bunch of files—nothing else. The actual database manipulation doesn't take place until we execute the migration file. Let's do that now:

```
C:\dev\book_tracker> rake db:migrate
(in c:/dev/book_tracker)

== 20080722191828 CreateBooks: migrating ==============================
-- create_table(:books)
   -> 0.2267s
== 20080722191828 CreateBooks: migrated (0.2269s) =====================
```

rake is Ruby's automation and task-running utility (more about rake in Chapter 12, *Finishing Touches*, on page 232), and we've just used it

to run the db:migrate task, thus creating our books table. Wait, where? Remember that, unless you specify otherwise, Rails will use the SQLite 3 engine for its development and test databases by default. If you now list the contents of the db directory, you'll see that your development database (the development.sqlite3 file) has just been created.

Fire It Up

Believe it or not, a complete application that we can use to maintain our book collection is now ready to use! But first, we'll want to start up WEBrick, the lightweight web server that comes with the default Rails installation.[11] WEBrick is the Rails equivalent of Web Developer Server for us ASP.NET devs—it's a technology that lets us run our app locally while developing. And, like Web Developer Server, we access it via localhost on a high-numbered port that doesn't interfere with other services on our machine. WEBrick starts up on port 3000 by default:

```
C:\dev\book_tracker> ruby script\server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000
=> Ctrl-C to shutdown server; call with --help for options
```

Head over to http://localhost:3000 using your web browser of choice. You should see the standard Rails welcome screen, as shown in Figure 1.1, on the following page. Now, simply add the name of the resource you're interested in—in this case books—to the end of that URL so that you end up with http://localhost:3000/books. Then you can marvel at what you've accomplished with one simple command.

As you browse, you'll find that everything you need to create, update, read, and delete books has been automatically generated for you. Not only is the application perfectly usable, but the code that's been generated is a great starting point for understanding how a Rails application is supposed to be built.

Time to Tweak

OK, that was easy. Let's take it a step further now. Let's say we'd now like to capture the date on which each book was purchased.

11. If you have the mongrel gem installed, it will be used instead of WEBrick.

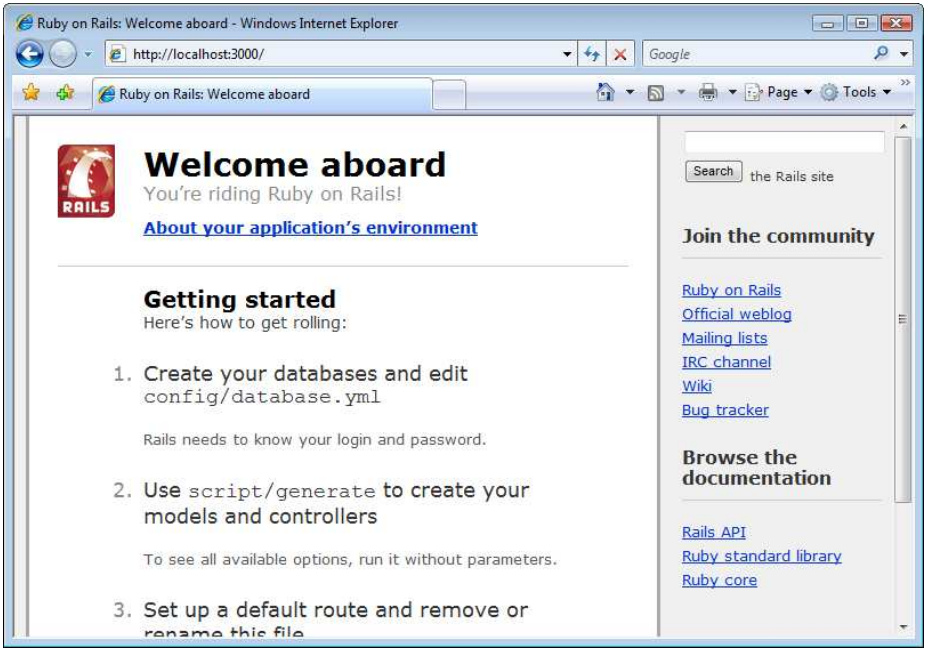


Figure 1.1: Rails welcome screen

Thanks to migrations, adding the field to the database is easy. Simply generate a new migration with this command:

```
ruby script\generate migration add_purchased_on_to_books
    exists  db/migrate
    create  db/migrate/20080722191930_add_purchased_on_to_books.rb
```

This will create a new migration file at db/migrate/20080722191930_add_purchased_on_to_books.rb. Now we'll add code to the self.up method to indicate what should happen when we upgrade from the current version (version 20080722191828) to the next version (version 20080722-191930—as indicated by the first part of the migration's filename). And, we'll also add code to the self.down method in case we ever want to roll back to the previous version.

Joe Asks...

How Do I Roll Back My Database to a Previous Version?

Migrations can also be undone:

```
C:\dev\book_tracker> rake db:rollback
```

In addition, the rake task used to migrate up also accepts an optional parameter with the target version. For example, the following command would also migrate the database back to version 20080722191828:

```
C:\dev\book_tracker> rake db:migrate VERSION=20080722191828
```

Download instant/20080722191930_add_purchased_on_to_books.rb

```
class AddPurchasedOnToBooks < ActiveRecord::Migration
  def self.up
    add_column :books, :purchased_on, :date
  end

  def self.down
    remove_column :books, :purchased_on
  end
end
```

Now, we'll execute the migration:

```
C:\dev\book_tracker> rake db:migrate
(in C:/dev/book_tracker)
== 20080722191930 AddPurchasedOnToBooks: migrating ====================
-- add_column(:books, :purchased_on, :date)
   -> 0.0470s
== 20080722191930 AddPurchasedOnToBooks: migrated (0.0470s) ===========
```

Now that we've added the new field to the database, the last step is to add the new field to the pages used to create, edit, and display books. Open app/views/books/new.html.erb, and you'll notice that the scaffolding created code that generates text fields for the title and author fields, as well as a checkbox for the "on loan" flag. The methods text_field and check_box in this code are known as *form helpers*. These helper methods tell Rails to show text input and checkbox input HTML tags, respectively, when the page is rendered. Since we've added a date column to the database, we could use a text input field if we wanted, but in most cases, the date_select helper method to display the date input using

Download instant/new.html.erb

```
<h1>New book</h1>

<%= error_messages_for :book %>

<% form_for(@book) do |f| %>
  <p>
    <b>Title</b><br />
    <%= f.text_field :title %>
  </p>

  <p>
    <b>Author</b><br />
    <%= f.text_field :author %>
  </p>

  <p>
    <b>On loan</b><br />
    <%= f.check_box :on_loan %>
  </p>

  <p>
    <b>Purchased on</b><br />
    <%= f.date_select :purchased_on %>
  </p>

  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>

<%= link_to 'Back', books_path %>
```

Head on over to http://localhost:3000/books again, and give it a try. Upon visiting the "create book" page, you should see something similar to what's shown in Figure 1.2, on the next page. Using the same technique, you should also go ahead and enhance app/views/new.html.erb in the same way.

More One-Liners—Validating Input

As we've seen so far, Rails is the king of the one-liners. These simple commands represent a lot of the little things you need to turn your app from a set of simple forms into a full-blown web application.

Try to create a new book with no title and no author. You'll find that there's no validation preventing that from happening. Luckily, there's an easy way to fix that. Open app/models/book.rb, and add a new line of

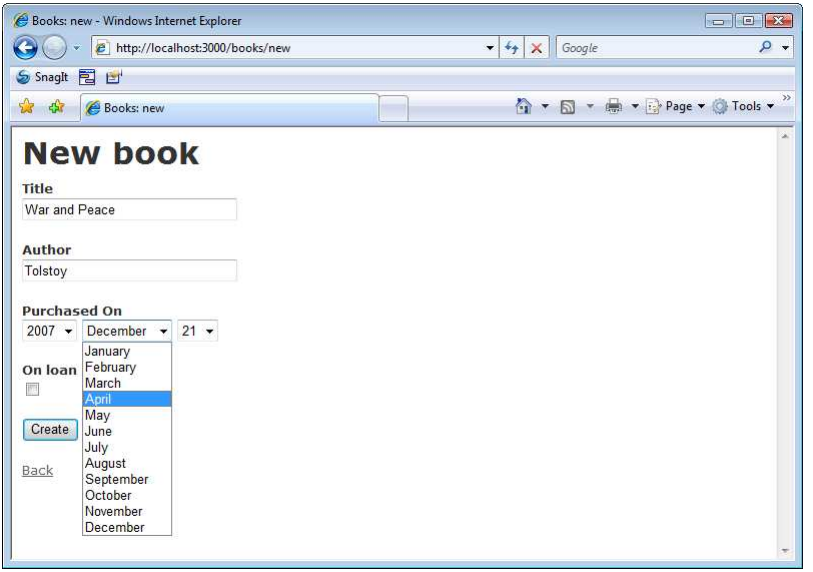


Figure 1.2: Creating a book with a "Purchased On" date

Download instant/book.rb

```
class Book < ActiveRecord::Base
        validates_presence_of :title, :author
end
```

Try to create a new book with no title or author one more time. Voila! Now Rails catches that error and displays an appropriate error message on the page.

In this chapter, we've gotten a solid Rails development stack installed, and we've seen how easy it is to get a simple application up and running. We've already built a real application that talks to a database, learned how to do basic database versioning, and even done some simple form validation. That's not bad for just a few lines of code.

We also built it all without the help of an IDE, which is probably different from what you're used to, if you've been living in the .NET world for a while.

You're now prepared to go a lot deeper into the anatomy of a Rails application and understand how the Ruby language plays a big part in what makes Rails what it is.

Chapter 2

Switching to Ruby

All good Rails developers share a common trait: they are also good Ruby programmers. A solid understanding of the Ruby programming language is essential to reaching those "ah-ha!" moments that will truly elevate your Rails expertise. Ruby is an object-oriented language, like C# or VB .NET, so many concepts in Ruby will be very familiar and easy to pick up.

However, because Ruby is also a scripting language and does not distinguish between a compilation phase and an execution phase, some details of Ruby semantics may not be as apparent. Ruby is also a *dynamically typed programming language*, which means any variable in memory is allowed to change its type while the program is running. This can be unsettling for those of us who are accustomed to a statically typed language like C# or VB .NET. Statically typed languages require that variables are not only declared as belonging to a specific type, but also they must remain so for the duration of the program's lifetime. An object's callable methods, properties, and internal field structures are immutable for the life of the object. Attempting to treat an object with the wrong type specification can be disastrous. The C# and VB .NET compiler catches type mismatch errors for us so that we're much less likely to find a type error at runtime.

In contrast, since Ruby objects can, by design, change their list of callable methods at runtime, Ruby developers learn to rely on unit tests to not only test the type-compatibility of their code but also to ensure the correctness of all facets of a Ruby application.

But there is a lot of good news. The similarities between Ruby and the .NET languages far outweigh their differences. In this chapter, we'll

highlight the essential elements of Ruby, often comparing them with their .NET counterparts, so that you can get up to speed with Ruby quickly and with confidence.

For a complete introduction to the Ruby language, see *Programming Ruby* [TFH05].

Ruby vs. .NET for the Impatient

If you've never written any code in Ruby, you're probably anxious to find out what's different and whether you'll have to throw out everything you already know about .NET and start over. Let's dive right in by answering some of the most frequently asked questions that .NET developers have when they start learning Rails:

- **In Ruby's object-oriented world, we work with *objects* and *methods*.** Unlike VB .NET, where some subroutines return a value (Functions) and others do not (Subs), all Ruby methods must return a value. When an explicit return statement is not used, the last-evaluated expression automatically becomes the return value.
- **Variables are not declared prior to their use.** Ruby automatically allocates memory for variables upon first use, and it also assigns their type based on inference. Like .NET, a garbage collector will reclaim memory automatically.
- **There is no distinction made between methods, properties, and fields (or "member variables") like there is in .NET.** Ruby has only the concept of methods. However, Ruby classes do sport a convenient syntax for defining "attribute methods," which are equivalent to defining .NET properties. attr_reader and attr_accessor automatically define methods that provide property-like access to instance variables.
- **Comments start with the hash character (#), unless the hash occurs inside a double-quoted string.** Everything after the hash is ignored. There is no multiline comment character in Ruby.
- **Ruby classes may define *instance methods* and *class methods*.** Class methods are called *static methods* in .NET.
- **Methods can be declared *public*, *protected*, or *private*, and these visibility scopes have the same meaning as they do in .NET.** There is no Ruby equivalent for "internal" or "assembly-

- **Instance variable names must start with an at (@) sign and are always private.** Class variables, or what we might call *static variables* in .NET, start with two at signs. The rules for memory allocation and object assignment for class variables can get pretty strange in Ruby, so we tend to avoid using them, especially since Rails provides an alternative syntax for using class variables in Rails applications. Here is an example of how to use simple instance variables in a Ruby class:

Ruby Download ruby101/objects.rb

```
class Flight

  def prepare
    # Assign a value to an instance variable
    # This variable can be seen by all other instance methods
    # as well
    @num_engines = 2
    @num_wings = 2
  end

  def report
    puts "We have #{@num_engines} engines
          and #{@num_wings} wings."
  end
end
```

- **Ruby classes can be derived from only one base class but can "mix in" any number of *modules*.** A module in Ruby is simply a set of related methods packaged together using the module keyword instead of class.

- **There is no separate compilation step in Ruby.** If we execute this Ruby code:

Ruby Download ruby101/objects.rb

```
class Flight

  puts "This is a flight"

  def fly_somewhere
    # ...
    # implementation code goes here
    #...
  end

end
```

you might not expect anything to happen, because it appears that

to_s vs. ToString()

.NET's base class, Object, provides a ToString method. All .NET classes override ToString so that each object can provide a string representation of themselves when needed.

Ruby's equivalent is to_s. Ruby objects override to_s to provide a suitable string representation.

In our example, we needed to emit the value of the len variable as a string. We used the to_s method to first convert the integer value to a string so we could concatenate them together.

string "This is a flight" emitted to the console! Ruby scripts are interpreted—that is, actually executed—one line at a time as the interpreter reads the file.

Let's get some Ruby code under our belt so we can examine the language in detail.

Our First Ruby Program

Start a new file named hello.rb, and open it with your favorite text editor. Type the following code into your file:

Download ruby101/hello.rb

```
name = 'Joe'
len = name.length

puts name + " has " + len.to_s + " letters in his name."
```

We don't need to define a Main method anywhere. Ruby is an interpreted language, and the Ruby interpreter simply starts at the top of the file and executes each Ruby statement in order until it gets to the end of the file.

Let's walk through this short example one line at a time to understand what this code will do. We immediately see how easy it is to create new variables in Ruby. The variable name is assigned to the string value Joe. Next, we create a variable called len and assign it the length of the string held by name. Finally, we use the built-in Ruby method puts to print a string to standard output.

Let's see it in action. Save the file, open a Windows command prompt, and go to the directory where you saved it (in our case, it's in c:\dev):

```
c:\dev> ruby hello.rb
Joe has 3 characters in his name.
c:\dev>
```

Next, we'll build upon the Ruby program we've written to explore specific, practical elements of Ruby that you'll need to know as you write Ruby code. The best place to start is to learn about how we work with string variables and string literals in Ruby.

Working with String Objects

In our previous example, we used single quote marks for the string 'Joe'. In Ruby, single quotes are similar to using the @ syntax for strings in C#. If you want to include special control characters in your string, such as a tab or carriage return, then you need to use double quotes instead. Using double quotes also allows us to use the string interpolation feature of Ruby, where we can embed any Ruby expression into a double-quoted string by surrounding the expression with #{ }:

Download ruby101/hello.rb

```
puts "\t#{5*10}"
puts "Hello, #{name}.  You have #{name.length} letters in your name"
```

```
c:\dev> ruby hello.rb
   50
Hello, Joe.  You have 3 letters in your name.
```

The String class provides so many helpful methods that we will just look at some of the more common methods we use in Rails applications. Let's start with how to search inside a string for a substring or pattern.

Searching and Replacing

Sometimes we want to know whether a string contains a particular substring and, if so, at which position it was found. This is done with the index method, which can take a literal substring to search for or a regular expression pattern.

If you've worked with regular expressions in .NET, you'll be glad to know that working with them in Ruby is significantly easier. Regular expression patterns are first-class citizens in Ruby, and denoting a regular expression in your code is as easy as working with strings. Instead of

using quotation marks, patterns are simply surrounded with forward slashes.

Regular expression patterns may look odd at first, because languages such as C# and VB .NET usually require the use of the RegExp class instead of being able to directly insert a pattern in your code.

In this next code sample, we use the index method to find out where each substring or pattern is found in our string:

Download ruby101/hello.rb

```
word = "restaurant"
puts word.index('a')         # prints 4
puts word.index("ant")       # prints 7
puts word.index(/st.+nt$/)   # prints 2
puts word.index(/ANT$/i)     # prints 7
puts word.index('buffet')    # prints "nil"
```

That last line printed something strange, didn't it? When index can't find a match, it returns nil. In Ruby, nil serves the same purpose as NULL in SQL, null in C#, and Nothing in VB.

Sometimes we need to replace a given substring with another. The sub method comes to our aid here. sub will find the first occurrence of a substring or pattern and replace it with something else. To perform multiple replacements in the same string, we can use gsub:

Download ruby101/hello.rb

```
flight = "United Airlines, Flight #312, ORD to LAX, 9:45AM to 11:45AM"
puts flight.sub('United', 'American')
puts flight.sub(/(\w+)to/, 'PDX to')
puts flight.gsub('AM', 'PM')
```

The resulting output will be as follows:

```
American Airlines, Flight #312, ORD to LAX, 9:45AM to 11:45AM
United Airlines, Flight #312, PDX to LAX, 9:45AM to 11:45AM
United Airlines, Flight #312, ORD to LAX, 9:45PM to 11:45PM
```

Notice that we did not actually modify the contents of the flight variable at all. This is because sub and gsub do not modify the string but simply return a new string with the desired replacements. We could have captured the result in a new variable if we wanted to work with the replaced version further. Or, we could have used the *bang* or "dangerous" versions of these methods, sub! and gsub!, to change the flight value in place. In Ruby parlance, "dangerous" methods are those that will modify the value of the object directly, and the Ruby convention is

to use an exclamation point in the method name to help communicate that fact.

Trimming Whitespace

Another common task when working with strings is removing leading and trailing whitespace. We could choose to combine our knowledge of gsub with regular expressions to do something like this:

Download ruby101/hello.rb

```
# notice extra whitespace
flight = "   United Airlines, Flight #312, 9:45AM to 11:45AM   "
flight = flight.gsub(/^\s+/, '') # remove leading whitespace
flight = flight.gsub(/\s+$/, '') # remove trailing whitespace
```

But the String class provides us with a much simpler alternative with the strip method:

Download ruby101/hello.rb

```
# notice extra whitespace
flight = "   United Airlines, Flight #312, 9:45AM to 11:45AM   "
flight = flight.strip  # removes leading and trailing whitespace
```

More likely, you'll want to use the "dangerous version" strip! instead: flight.strip! will remove leading and trailing whitespace from flight.

Splitting a String into Parts

We'll wrap up our quick overview of Ruby strings with the split method, which is handy when you need to break up a string into pieces based on a delimiter or delimiting pattern of some kind. The split method returns an instance of an Array class:

Download ruby101/hello.rb

```
flight = "United Airlines, Flight #312, ORD to LAX, 9:45AM to 11:45AM"
info = flight.split(',')
puts info.size    # prints 4
puts info         # prints the contents of the info array

info = flight.split(/\s*,\s*/)
puts info.size    # prints 4
puts info         # show the info array, this time without extra spaces
```

This concludes our quick look at handling strings. You're encouraged to explore the rest of the String methods on your own. Next, we'll take a look at irb, a handy utility that makes exploring Ruby objects easier.

irb Is Your New "Immediate Mode"

Let's learn about a useful command-line tool that comes with Ruby called irb (interactive Ruby). irb is an interactive tool for learning Ruby, and for doing all sorts of Ruby experiments, without having to endure the laborious edit/save/run cycle that we've been doing so far. If you've ever used Visual Studio's "immediate mode" window, irb should feel very familiar.

To use irb, just open a Windows command prompt, and type irb. You should see something like this:

```
c:\dev> irb
irb(main):001:0>
```

Enter any Ruby expression, and irb will evaluate it and emit the result:

```
irb(main):001:0> 1 + 2
=> 3
irb(main):002:0>
```

irb prefixes the result with =>. In fact, irb always displays the resulting value of the expression or statement entered, even when it might seem unexpected; for example:

```
irb(main):002:0> puts 'Hello'
Hello
=> nil
irb(main):003:0>
```

We had indeed wanted the string Hello to be displayed, but why did irb also emit the => nil after that? Like every method call in Ruby, puts must return a value, even if it's nil, and irb always displays the return value of the expression or method call that we've entered at the irb prompt. Here, we've discovered that the return value from puts is nil.

Arrays

When we are building an application in Rails, we will find ourselves working with arrays quite a bit. Arrays in Ruby are always *dynamic* and *heterogeneous*. They are said to be dynamic because arrays have variable length. We don't have to decide ahead of time how many elements the array will have. The array will grow automatically as elements are inserted into the array. They are said to be heterogeneous because we can store elements of any type into any element of an array.

The built-in Array class provides a lot of functionality right out of the box. So, let's take a look at how to get started with arrays in Ruby.

Creating an Array

We will begin by looking at a small C# console application that creates a couple of arrays and displays some information on the console:

Download ruby101/array.cs

```
List<String> colors = new List<String>();

// Add some elements
colors.Add("Purple");
colors.Add("White");

// Emit first color to console
Console.WriteLine(colors[0]);   // prints "Purple"

// Create a new array, initialized with some data
List<String> sports = new List<string>
                { "Hockey", "Baseball", "Football" };

// Emit first sport to console
Console.WriteLine(sports[0]); // or, sports.First(); if using LINQ

// Combine two arrays
List<String> favorites = new List<string>();
favorites.AddRange(colors);
favorites.AddRange(sports);

// Emit the size of the combined array
Console.WriteLine(favorites.Count); // prints 5
```

Now let's see how we accomplish the same thing in Ruby. There are two easy ways to create new array objects in Ruby. The first is to explicitly declare a new Array instance. The second uses a special, implicit bracket syntax to create an array on the fly.

In the following example, we create a new array instance and assign it to the variable colors. Remember that in Ruby, we don't need to declare the type of variable to the interpreter. Instead, Ruby surmises the variable type at runtime.

Download ruby101/array.rb

```
colors = Array.new
colors.push 'Purple'
colors << 'White'  # => same as .push
puts colors[0]   # => prints "Purple"
```

```
sports = [ 'Hockey', 'Baseball', 'Football' ]
puts sports.first # => prints "Hockey"

favorites = colors + sports
puts favorites.size  # prints 5
```

The sports array shows an alternate style of using arrays. In Ruby, arrays are denoted by square brackets. Here we've used the bracket syntax to easily populate our array with an initial list of values.

The Array class defines many useful methods that help us operate on arrays, and here we've used the push to append new elements to the array. Elements in an array can be retrieved directly by providing a zero-based index value. Therefore, puts colors[0] will display the first element of our array.

The Array class also provides many convenient "operator overloads." We created the favorites array by simply "adding" the elements of the two previous arrays together. When operating on Array instances, the addition operator creates a new array by starting with the elements of the first array and then appending all the elements from the second array. We then captured this new array into a variable named favorites.

Here is an example of an array that contains both strings and numbers:

Download ruby101/array.rb

```
mixed = [ "Cars", 36, 10*50, "Tables" ]
puts mixed[0]   # prints "Cars"
puts mixed[2]   # prints 500
```

Common Array Operations

Given an array, we can determine how many elements are in the array with either the length or size method. This is an example of how Ruby sometimes provides more than one method to perform the same function. Simply choose the method that, in your opinion, makes your code more understandable or more readable. While other languages strive for a *minimal interface*, Ruby tends toward what is sometimes called a *humane* interface, because Ruby tends to value readability and clarity of code over the principle of having as few public methods as possible.

In addition to accessing an element directly by its index, we can also obtain the first and last elements and search for elements that meet some desired criteria with the select and find methods.

Take a look at this code:

Download ruby101/array.rb

```
sports = [ 'Hockey', 'Baseball', 'Football' ]
puts sports.first   # prints "Hockey"
puts sports.last    # prints "Football"

puts sports.find   { |sport| sport =~ /ball/ }  # prints "Baseball"
puts sports.detect { |sport| sport =~ /ball/ }  # prints "Baseball"

uses_a_ball = sports.select { |sport| sport =~ /ball/ }
puts uses_a_ball.size  # prints 2
```

The select and find methods are examples of methods where you supply a *block* to the method. The select method returns a list of all matching elements, whereas find will find and return only the first match.

You may have noticed that detect and find returned the same result. Again, you are free to choose whichever method gives your code the most natural feel.

Transforming an Array into a String

Earlier we learned how to use the split method on String to create an array of strings. Sometimes we want to do the inverse: given an array of strings, we need to join them together into one string. If we call the join method without any arguments, it will simply glue the strings together. We can also pass an string value that we would like to use as the "glue" between the elements as they are joined. Here are some examples:

Download ruby101/array.rb

```
colors = [ "Blue", "Green", "Orange" ]
puts colors.join
puts colors.join(' and ')
puts colors.join("\n")
```

```
c:\dev> ruby array.rb
BlueGreenOrange
Blue and Green and Orange
Blue
Green
Orange
```

Notice that for the last line, we used double quotes around our joining string so that we could specify a newline (carriage return) to join the strings together. This is how we got each element of the array to be printed on its own line.

Joe Asks...

What's a Block?

Blocks are a lot like .NET anonymous methods: sections of code that behave like a function but just don't have a name. Some methods expect to collaborate with a snippet of code that you must provide. Your code snippet is a block. Simply enclose your code in a do...end block (one-liners should instead use curly braces).

Blocks can even take parameters, just like a method can. Block arguments are surrounded with pipe | symbols.

Download ruby101/blocks.rb

```
# The Array.detect method expects us to define a block
# and for our block to return true
# when the given element is the desired
# element we want to find.
sports.detect { |sport| sport.index('ball') != nil }

# Logic that requires more than one line
# should use a do..end pair instead.
# Here, we detect the first sport
# that can be played on a grass surface.
sports.detect do |sport|
  category = find_category(sport)
  category == 'played on grass'
end
```

Blocks arrange your code right alongside the method that's requiring it, making your code more readable than locating a method somewhere else in your program.

This is just one way that Ruby promotes more generic, decoupled, and collaborative program architectures.

Finding Array Elements with Regular Expressions

You may already be familiar with grep, a common command-line utility that searches text using regular expressions. Some Ruby classes also implement a method named grep, which indicates it has some kind of similar searching feature. The Array class provides a grep method so that you can easily search an array's elements for those that match a given regular expression:

Download ruby101/array.rb

```
colors = [ "Blue", "Green", "Orange", "Red", "Purple" ]
my_colors = colors.grep(/u/)
puts my_colors
```

```
c:\dev> ruby array.rb
Blue
Purple
```

Shortcut for Creating an Array of Strings

Ruby provides a shortcut when we want to create an array from a list of words:

Download ruby101/array.rb

```
colors = %w(Blue Green Orange Red Purple)
```

The %w is a special sequence in Ruby code. The parentheses are commonly used, but you can use any character that you want to use to mark the beginning and ending of your list of words. Whitespace is used to split the string up into the array of words.

Deleting Elements from an Array

Removing an object from an array is easy. Just call the delete method. You must provide a reference to the object you want to delete. If the object is found in the array, it will be removed and returned to you. If the object is not found, the delete method will simply return nil, instead of raising an exception like .NET languages would.

Alternatively, if you don't have a reference to the object but you do know the index position of the element you need to delete, you can use the remove_at method to remove whatever element is at that position and have it returned to you. You'll get nil back if you supply an index that's out of range for the array.

Download ruby101/array.rb

```
colors = [ "Blue", "Green", "Orange", "Red", "Purple" ]
colors.delete 'Blue'  # returns "Blue"
colors.delete 'Green' # returns "Green"
colors.delete 'Brown' # returns nil
colors.delete_at(0)   # returns "Orange"
colors.delete_at(5)   # returns nil

# colors is now [ "Red", "Purple" ]
```

Symbols

The string and array classes we've examined so far correspond well to similar .NET classes. Symbols, on the other hand, don't have an exact counterpart. They play a vital role in Rails programming, so we need to become familiar with them before we move on.

We will start by looking at two common .NET constructs: constants and enums. Let's start with a typical use of the const keyword:

```
public const int Circle = 1;
public const int Square = 2;
public const int Octagon = 3;

public void DrawShape(int shape)
{
    switch (shape)
    {
        case Circle:
          // draw a circle
          break;
        case Square:
          // draw a square
          break;
        case Octagon:
          // draw an octagon
          break;
    }
}

// Let's draw a square
DrawShape(Square);
```

By using names instead of passing raw integers, our code becomes clearer. Inside DrawShape(), we used the names of our shapes instead of relying on hard-coded integer values. Outside the class, the benefit is just as large. We could have written DrawShape(2) instead, but that would be harder to read and understand.

Some .NET developers want to guarantee that the DrawShape() method can receive only a valid shape constant, instead of just any old integer. After all, the actual values we assigned to each shape constant were arbitrary; it didn't really matter what value we assigned to each constant. Enumerated values in .NET are useful for associating names with constant values when you don't really care about the actual value. For example, we can rewrite the previous example using an enum instead:

```
public enum Shape { Circle, Square, Octagon }

public void DrawShape(Shape shape)
{
    switch (shape)
    {
        case Shape.Circle:
          // draw a circle
          break;
        case Shape.Square:
          // draw a square
          break;
        case Shape.Octagon:
          // draw an octagon
          break;
    }
}

// Let's draw a square
DrawShape(Shape.Square);
```

This code retains the same level of clarity, and we didn't have to specify any integer values anywhere. (Behind the scenes, the C# compiler will assign integer values to each enumerated value, but it's a hidden, unimportant detail as far as the programmer is concerned.)

Here's the key point: a .NET enum value is a *globally unique, named representation of a single location in memory*. The actual value held at that memory location is unimportant. When all we want is a named value in C#, enums are our friends.

At last, we're ready for Ruby symbols. What is a *symbol*?

A symbol is globally unique, named representation of a single location in memory. (Sound familiar?) So, any time we want to have a nice name for something and we don't care about what value it really has, we use a symbol.

Here's the same code, this time written in Ruby:

```
def draw_shape(shape)
  case shape
    when :circle
       # draw a circle
    when :square
       # draw a square
    when :octagon
       # draw an octagon
  end
end

# Let's draw a square
draw_shape(:square)
```

Symbols always start with a colon, as in :circle. Unlike .NET enumerations like Shape, we don't need to define symbols before we use them. Ruby will automatically spring them into existence for us the first time they're used.

Symbols can be used like any other type in Ruby. They are objects of type Symbol, and you can call methods on them. Here's how we can get a string representation of a symbol, for example:

```
puts :square.to_s   # prints "square"
```

As another example, here's how we create an array of them:

```
# An array of three symbols
[:circle, :square, :octagon]
```

The most common place we're going to find ourselves using symbols in Rails is when we work with hashes, which we will explore next.

Hashes

In Ruby, a *hash* is very much like a .NET Dictionary. Like an array, a hash is a data structure that contains a series of elements. Unlike an array, hash elements are pairs of objects: a *key* and a *value*. Hashes are extremely easy to use in Ruby, and they come in handy in web development where we often want to associate one object with another.

Like arrays in Ruby, hashes are also heterogeneous structures. There is no requirement that all the objects in the hash are of the same type. We can mix and match any type of objects into any of the keys and values. Ruby doesn't care. It's up to you to do whatever you think is appropriate for your situation.

Let's start by looking at a few simple examples. There are two common ways of creating hashes in Ruby. We can create a new instance of the Hash class:

```
irb(main):001:0> hangar_status = Hash.new
=> {}
irb(main):002:0> hangar_status['waiting'] = 3
=> 3
irb(main):003:0> hangar_status['repairing'] = 7
=> 7
irb(main):004:0> hangar_status
=> {"waiting"=>3, "repairing"=>7}
```

Here we see that we can create new key/value pairs by using the [] operator. If the given key does not exist, it is created automatically, and the given value is paired with the given key. If the key already exists, it is simply paired up with the new value.

If we try to access a key that doesn't exist, we simply get a nil value back (whereas some .NET implementations would raise an exception):

```
irb(main):003:0> hangar_status['damaged']
=> nil
```

Alternately, we can use curly braces to directly create a populated Hash:

```
irb(main):001:0> hangar_status = { 'waiting' => 3, 'repairing' => 7 }
=> {"waiting"=>3, "repairing"=>7}
```

Notice that when we work with hash elements, we use the special => syntax to specify the key/value pair.

We can use any kind of objects we want in a hash at any time. Let's keep track of how many passengers are onboard our airplanes. Here we create two Airplane objects and use them as keys in a hash. The values in the hash represent the number of passengers onboard each plane. Note how we create an initially empty hash by just using two curly braces:

```
irb(main):001:0> ord_to_jfk = Airplane.new '747'
=> #<Airplane:0x8c050 @altitude=0, @model="747", @speed=0>
irb(main):002:0> pdx_to_sfo = Airplane.new '707'
=> #<Airplane:0x870a0 @altitude=0, @model="707", @speed=0>
irb(main):003:0> passengers = {}
=> {}
irb(main):004:0> passengers[ord_to_jfk] = 165
=> 165
irb(main):005:0> passengers[pdx_to_sfo] = 104
=> 104
irb(main):006:0> passengers[ord_to_jfk]
```

Hashes are a powerful way to store data. It's very important that you become comfortable working with hashes as you begin to develop Rails applications.

Finally, here are some examples that use symbols as keys into a hash:

```
options = { :model => '747', :capacity => 250, :engines => 2 }

puts options[:capacity]   # prints 250
options[:engines] = 3     # we now have 3 engines
```

We've completed a quick tour of four of Ruby's data types that we will use frequently in Rails applications: strings, arrays, symbols, and hashes. We haven't yet talked much about the object-oriented nature of the Ruby language. We need to do that now before we can continue to explore the other fundamental elements of the language.

Everything Is an Object

In Ruby, we like to say everything is an object. Although many .NET languages like C# and VB .NET are considered to be object-oriented, Ruby's definition of object-oriented is more hardcore. Every element of code in Ruby really is an object. To illustrate what we mean, let's look at some of the more startling examples of how objects work in Ruby as we continue to highlight the similarities and differences with .NET.

Even Built-in Types Are Objects

Ruby has built-in types like strings, arrays, and fixnums. Fixnums are like the Int32 data type in .NET. When we do something like this in Ruby:

Download ruby101/objects.rb

```
puts 1 + 2    # prints 3
```

we are still using objects even though it looks like we're using built-in literal values for 1 and 2. The literal number 1 in Ruby is an object! It's actually an instance of the Fixnum class. So is the literal number 2. The plus sign is syntactic sugar that allows us to call a method named + on the 1 object.

Mirror, Mirror on the Wall

An interesting feature of the Ruby language is the built-in ability for every class and object to tell us about themselves. Every Ruby class is derived directly or indirectly from a built-in class named Object. Object

object's type. Here is an example of how we can use the class method to find out the class of a given object:

Download ruby101/objects.rb

```
puts 1.class  # prints "FixNum"
puts 2.class  # prints "FixNum"
```

.NET calls this technique *reflection*, because we're asking an object to look itself in the mirror and tell us what it sees. Rubyists sometimes like to call it *introspection*, but it's the same concept. Reflection is very useful and powerful in a dynamic language like Ruby, since an object's behavior is subject to change at runtime. Let's learn two more ways to have an object tell us more about itself.

We can use methods to find out what methods we can call on an object:

```
irb(main):001:0> s = "hello"
=> "hello"
irb(main):002:0> s.methods
=> ["%", "select", "[]=", "inspect", "<<", "each_byte", "clone",
"method", "gsub", "casecmp", "public_methods", "to_str", "partition",
"tr_s", "empty?", "instance_variable_defined?", "tr!", "equal?",
"freeze", "rstrip", "*", "match", "grep", "chomp!", "+", "next!",
"swapcase", "ljust", "to_i", "swapcase!", "methods", "respond_to?",
"upto", "between?", "reject", "sum", "hex", "dup", "insert",
"reverse!", "chop", "instance_variables", "delete", "dump", "__id__",
"tr_s!", "concat", "member?", "object_id", "succ", "find", "eql?",
"each_with_index", "strip!", "id", "rjust", "to_f",
"singleton_methods", "send", "index", "collect", "oct", "all?",
"slice", "taint", "length", "entries", "chomp", "frozen?",
"instance_variable_get", "upcase", "sub!", "squeeze", "include?",
"__send__", "instance_of?", "upcase!", "crypt", "delete!", "detect",
"to_a", "unpack", "zip", "lstrip!", "type", "center", "<",
"protected_methods", "instance_eval", "map", "<=>", "rindex",
"display", "any?", "==", ">", "split", "===", "strip", "size",
"sort", "instance_variable_set", "gsub!", "count", "succ!",
"downcase", "min", "kind_of?", "extend", "squeeze!", "downcase!",
"intern", ">=", "next", "find_all", "to_s", "<=", "each_line",
"each", "rstrip!", "class", "slice!", "hash", "sub", "tainted?",
"private_methods", "replace", "inject", "=~", "tr", "reverse", "nil?",
"untaint", "sort_by", "lstrip", "to_sym", "capitalize", "max",
"chop!", "is_a?", "capitalize!", "scan", "[]"]
```

Wow, that's a lot to look at. Let's make it easier to read by sorting the methods first and by using grep to find only the methods that start with, say, the letter *s*:

```
irb(main):003:0> s.methods.sort.grep(/^s/)
=> ["scan", "select", "send", "singleton_methods", "size", "slice",
"slice!", "sort", "sort_by", "split", "squeeze", "squeeze!", "strip",
"strip!", "sub", "sub!", "succ", "succ!", "sum", "swapcase",
```

Use methods inside an irb session when you want to discover exactly what a class or object can do. Here we learned how handy the sort and grep methods can be.

Quacking Like a Duck

Every Ruby object, like the string object we just examined, ultimately inherits from Object. This is the same as it is in .NET. As a result, many of the methods that a string object has are inherited from Object. Perhaps the most interesting method of all of these is the respond_to? method. With this method, we can find out at runtime whether an object can respond to a particular message or method:

```
irb(main):005:0> s.respond_to?('split')
=> true
irb(main):006:0> s.respond_to?('translate_to_spanish')
=> false
```

Here we've used the respond_to? method to discover that the object s implements a method called split but does not have a method named translate_to_spanish.

This is the main idea behind *duck typing*: Ruby can detect available runtime behaviors at the method level. Instead of achieving polymorphic behavior only through base class inheritance or interface implementation, Ruby code can work with objects that simply "talk" and "walk" as expected, regardless of their actual type.

Classes and Objects

Declaring classes and using classes are the bread and butter of any object-oriented system, and the reasons for using classes, instantiating objects, and calling their methods are the same in Ruby as they are in .NET. Let's look at some concrete code examples to see how they compare.

Let's Fly an Airplane

We will start by reviewing how we define classes in the .NET language C#. Here is a C# class that represents an airplane, perhaps to be used in a flight simulator program or as part of an air traffic control system:

Download ruby101/classes.cs

```
using System;
using System.Collections.Generic;
```

```
public class Airplane
{
    private string model;
    private int altitude;
    private int speed;
    public Airplane(string model)
    {
        this.model = model;
        this.altitude = 0;
        this.speed = 0;
    }

    public string Model
    {
        get
        {
            return this.model;
        }
    }

    public int Altitude
    {
        get
        {
            return this.altitude;
        }
    }

    public int Speed
    {
        get
        {
            return this.speed;
        }
    }

    public void Fly()
    {
        if (this.model == "777")
            this.altitude = 40000;
        else
            this.altitude = 30000;

        this.speed = 500;
    }

    public void Land()
    {
        this.altitude = 0;
        this.speed = 0;
    }
```

```
    public static List<String> Models
    {
            string[] availableModels = { "707", "747", "777" };
            return new List<string>(availableModels);
    }
}
```

We've defined a public class called Airplane that must be initialized with a model parameter. A static method called Models() returns a list of available models that can be used. An Airplane instance can keep track of its speed and altitude, and these are changed by calling the Fly() and Land() methods. Finally, we defined three read-only properties, Model, Altitude, and Speed.

Now, let's translate this class directly into Ruby. Our first attempt may not result in the most concise Ruby code ever written, but we'll refine the code after we've performed our initial translation:

Download ruby101/classes.rb

```
class Airplane

  def initialize(model)
    @model = model
    @altitude = 0
    @speed = 0
  end

  def model
    return @model
  end

  def altitude
    return @altitude
  end

  def speed
    return @speed
  end

  def moving?
    return @speed > 0
  end

  def fly
    if @model == '777'
      @altitude = 40000
    else
      @altitude = 30000
    end
```

```
    @speed = 500
  end

  def land
    @altitude = 0
    @speed = 0
  end

  def self.models
    return [ '707', '747', '777' ]
  end


end
```

Ruby is a very readable language, but let's go through it a step at a time. We introduce a new class scope with the class keyword. Our Airplane class doesn't explicitly derive from any other class, so it will be derived from the built-in Object class by default.

On line 3, we see how to define the *initializer* for our class. The initializer is Ruby's equivalent of a .NET constructor. You can define your initializer to accept parameters, but note that only one initializer is allowed, since Ruby does not support method overloading. Here we've declared that our initializer will require one parameter, the "model" of airplane that should be created.

In the body of the initializer, we have defined three instance variables, @model, @altitude, and @speed. The @ sign declares them as private instance variables, whose type is automatically determined by the assigned values.

On line 9, we see how to declare a method that will give us property-like syntax for reading the value of an instance variable. We've defined the model method, which simply returns the value of our @model attribute. In this way, we provide read-only access to the @model variable. We've done the same thing for the @altitude value, by providing a wrapper method for it as well.

The fly method on line 25 uses a simple if statement to determine at which altitude we should fly, based on the type of airplane being flown. The instance variable @speed is set to 500 regardless of the airplane type. The land method, as its name implies, bring us back to Earth and brings the plane to a stop.

Finally, the models method is an example of a *class method*. Class methods are like static methods in .NET. They're a convenient way to house methods that logically belong to a class, even though a specific instance of that class is not needed to call the method. Here, we call Airplane.models to retrieve a list of the valid models that we can use when constructing an airplane instance:

Download ruby101/classes.rb

```
model_to_use = Airplane.models[1]

airplane = Airplane.new(model_to_use)
puts "Our #{airplane.model} is starting at #{airplane.altitude} feet"

airplane.fly
puts "Our #{airplane.model} is currently at #{airplane.altitude} feet"

if airplane.moving?
  puts "Yes, the plane is moving."
end

airplane.land
puts "Our #{airplane.model} is now at #{airplane.altitude} feet"
```

What happens when we execute this Ruby script? See whether you guessed correctly:

```
c:\dev> ruby classes.rb
Our 747 is starting at 0 feet
Our 747 is currently at 30000 feet
Yes, the plane is moving.
Our 747 is now at 0 feet
c:\dev>
```

A More Idiomatic Approach

The class we just wrote was fairly readable, and it reflects how it would probably look if it were written by a C# programmer making his first Ruby program. However, in practice, this code would be a bit different. Let's learn about just a couple of standard idioms commonplace in Ruby programming. Here is a revised example that shows one way our code could have been written:

Download ruby101/classes.rb

```
class Airplane

  attr_reader :model
  attr_reader :altitude
  attr_reader :speed
```

```
  def initialize(model)
    raise "Model not recognized!" unless Airplane.models.include?(model)

    @model = model
    @altitude = 0
    @speed = 0
  end

  def fly
    @speed, @altitude = 500, cruising_altitude
  end

  def land
    @altitude = 0
    @speed = 0
  end

  def moving?
    @speed > 0
  end

  def self.models
    [ '707', '747', '777' ]
  end

private

  def cruising_altitude
    @model == '777' ? 40000 : 30000
  end

end
```

What did we change?

- We omitted return statements wherever it seemed reasonable. Ruby always uses the value of the last-evaluated expression as the return value of a method, so often it's not needed to use the return statement explicitly.
- Like a *getter* in .NET properties, we used the built-in attr_reader method to automatically define read-only access methods to the @model and @altitude values. This helps us avoid writing boilerplate code like we did before to return those values.
- Because type safety is not ensured as it would be with a statically typed language such as C#, we added a check in the initializer to make sure a valid model was given and to raise an exception if we get an unexpected model.

- Finally, we've added a private section to the class and defined a new private method called cruising_altitude. This allowed us to make the fly method more readable and easier to maintain. We assign multiple values at once in Ruby using a comma-separated list of values on each side of an assignment statement.

Writing Ruby that feels natural takes some time and experience. Seek out Ruby libraries that have been written by longtime Rubyists to get a sense of how to recognize good Ruby style when you see it. Reading other code written by good Ruby coders is one of the best ways to learn how to write code in Ruby.

Loops

We will close this chapter by looking at another essential building block of Ruby programs: the concept of loops and iterators. Web applications spend a lot of time working with sets of data, usually enumerating over them for some purpose, perhaps to display a list of some kind, select a subset of data based on some kind of criteria, or transform a set of objects from one kind to another.

Some looping constructs in Ruby are quite different from the constructs you have experienced in .NET, while some of them are almost the same. There are two main differences to keep in mind when learning to read and write loop statements in Ruby as compared with C# or VB .NET:

- We don't need to explicitly declare any types, since Ruby always determines variable types automatically.
- Everything is an object, including seemingly built-in literals, so some loops are easier to write.

Simple Iterations

Here's a simple for loop that prints the digits 0 to 9 in C#:

Download ruby101/loops.cs

```
for (int i = 0; i < 10; ++i)
{
  Console.WriteLn(i);
}
```

And here's one way to do it in Ruby:

```
irb(main):001:0> 0.upto(9) { |n| puts n }
0
1
2
3
4
5
6
7
8
9
=> 0
```

Iterating Over Every Element of an Array

The most common way to simply loop through each element of an array is to use the each method:

```
irb(main):008:0> odds = [1, 3, 5, 7, 9]
=> [1, 3, 5, 7, 9]
irb(main):009:0> odds.each { |n| puts n }
1
3
5
7
9
=> [1, 3, 5, 7, 9]
```

The each method class is one example of an *iterator*. An iterator is a powerful pattern in Ruby programs by which a data structure's elements are visited in collaboration with a user-supplied block. Iterators are just one of the important dimensions of the Ruby language we'll be exploring in the next chapter.

We've started from the simplest of Ruby scripts; seen how primary elements like strings, arrays, and loops compare with their .NET counterparts; and dipped our toes in some advanced topics such as introspection and a few rules of idiomatic Ruby. It's time to learn more about iterators, a central concept in all Rails applications; see some advanced Ruby syntax; and take a closer look at code reuse techniques. These are the skills that will take us beyond thinking about simple .NET-to-Ruby translations so that we can instead "think in Ruby" as our native language.

Chapter 3

Ruby Skills for the Rails Developer

In this chapter, we will explore two pillars of the Ruby language: working with data collections and strategies for code reuse. Both of these are aspects of Ruby that play a central role both in the construction of Rails applications and inside the Rails framework.

If you've ever written an application that is primarily driven by data retrieved from a database, you've had to tackle the thorny problems of retrieving, filtering, sorting, and transforming raw data into a format that is required by the users of your application. Rails has powerful facilities for transparently moving sets of data in and out of Ruby collection classes, but that's of no use if we don't know how to efficiently manipulate those collections in our Ruby code to prepare it for display to our users.

Code reuse is an important topic in any application language or framework. Agile development speaks of the *DRY* principle[HT00], which stands for Don't Repeat Yourself. In other words, we never want to see a section of code duplicated elsewhere in our program. Redundant code is harder to maintain, debug, and understand. We want our programming language to make it easy to reuse code so that we can write a method or class and reuse it whenever we need it. The DRY principle is a mantra of the Rails community and is a driving force behind the design of Rails. We will be learning how Ruby enables easy code reuse in our applications as well.

Working with Collections and Iterators

Web applications spend a good deal of time manipulating various sets of data. In .NET we often use the term *collection* to refer to an array, a dictionary, or some other data structure that contains a series of data elements. We've already learned about the Array and Hash classes and how to perform basic manipulation of the elements they contain. Ruby takes the notion of a collection class a step further than pure containment, beyond the insert/retrieve/remove interface that is familiar to every .NET developer.

When we work with data collections in our programs, a particular pattern called the *iterator* pattern often emerges. This pattern describes the frequent need to examine a collection of some kind, visit each element of the collection, and then do something with that element. For example, starting from a given collection, we may way want to synthesize an entirely new data set containing only those elements that meet certain criteria. The new data structure might even contain elements of a completely different type than the original elements. Or perhaps we just want to visit all of the elements, inspecting them along the way, to perform some kind of aggregate calculation or statistical analysis of the data at hand.

Ruby's collection classes, such as Array and Hash, expose public methods that help encapsulate these usage patterns so that the code we have to write can be reduced to a bare minimum. To get a sense of the power of these iterator methods, we will start with some typical examples of how we work with collections in both .NET and compare them with their Ruby equivalents to see the power of Ruby iterators in action.

Looping Over a collection

Suppose we have a collection of Airplane objects. Each Airplane has a large number of properties, including model number, passenger capacity, the year it was built, the date of last service, and more. Let's imagine further that our airplanes are required to be serviced every four months.

We can find out how many planes are due for service by using a simple loop. Here's a typical loop in C#:

Download ruby201/each.cs

```
public int CountPlanesDueForService(Plane[] planes)
{
```

```
  foreach (Plane plane in planes)
  {
    if (DateTime.Now.Subtract(TimeSpan.FromDays(90)) >
          plane.LastServiceDate)
    {
      count += 1;
    }
  }

  return count;
}
```

Let's accomplish the same task in Ruby:

Download ruby201/each.rb

```
def count_planes_due_for_service(planes)
  count = 0
  planes.each do |plane|
    if (Date.today - 90 > plane.last_service_date)
      count += 1
    end
  end
  return count
end
```

This simple example reveals the central role of Ruby's block syntax. The each method is an iterator method. It visits each element, one at a time. As it does so, it passes the current element to a Ruby code block that we must provide.

Ruby blocks are analogous to .NET 2.0 anonymous methods: they are a section of code that looks and behaves just like any other method definition. They can even take parameters just like a method can. The only real difference is that instead of defining your method elsewhere in your class definition, you're providing the method definition inline with the code that's calling it.

Grasping the block syntax in Ruby depends on understanding the following key concepts about Ruby syntax:

- Blocks are bounded by a simple do...end pair, similar to the pairing def...end we use for named methods.
- Blocks can specify a list of expected arguments, just like regular named methods. However, instead of the typical parentheses we place around arguments in normal method definitions, we use the vertical pipe symbol instead.

- Blocks return a value, just like all other methods in Ruby. If an explicit return statement is omitted, the last-evaluated expression is used as the return value from the block.

With these rules in mind, let's return to the block we wrote:

Download ruby201/each.rb

```
planes.each do |plane|
  if (Date.today - 90 > plane.last_service_date)
    count += 1
  end
end
```

It should now be clear that the block:

- accepts a parameter named plane, and
- increments the count variable for planes that haven't been serviced in the last 90 days.

Now that we know the basics of how we write a block to collaborate with the each() method, let's now see how Ruby utilizes this same pattern when we need to select certain elements from a collection.

Selecting Elements from a Collection

Suppose that instead of merely obtaining a count of planes due for service, we want to create a list of those airplanes that are due for service. How would we write code to do that? Let's start with a C# 1.1 example:

Download ruby201/iterator.cs

```
using System.Collections;

public Plane[] FindPlanesDueForService(ArrayList planes)
{
  // Example for .NET 1.1

  ArrayList dueForService = new ArrayList();

  foreach (Plane plane in planes)
  {
    if (DateTime.Now.Subtract(TimeSpan.FromDays(90)) >
                                  plane.LastServiceDate)
    {
      dueForService.Add(plane);
    }
  }

  return (Plane[]) dueForService.ToArray(typeof(Plane));
```

Here we start with a data structure (in this case, an array of airplanes) and we want to return our results by generating a new data structure that contains only the data in which we're interested. To accomplish that, we write a very typical foreach loop so that we can visit each plane in the array. We then examine each plane's service record and determine whether it is due for service. If it is, we then add that airplane to our list of planes that must be scheduled for service. Finally, we return our newly created list of airplanes.

Selecting elements from a data structure based on some criteria became a bit easier in C# 2.0 with the use of generics. We can use the FindAll() method to help us find the planes we want:

Download ruby201/iterator.cs

```
using System.Collections.Generic;

// Using C# 2.0 Generics

public List<Plane> FindPlanesDueForService(List<Plane> planes)
{
    List<Plane> dueForService = planes.FindAll(IsDueForService);

    return dueForService;
}

public bool IsDueForService(Plane plane)
{
   DateTime cutoffDate = DateTime.Now.Subtract(
                              TimeSpan.FromDays(90));
   return cutoffDate > plane.LastServiceDate;
}
```

The introduction of LINQ provides a more succinct way to do all these things at once, using a more SQL-esque syntax:

Download ruby201/iterator.cs

```
public List<Plane> FindPlanesDueForService(List<Plane> planes)
{

    // Using C# 3.5 Generics and LINQ

    var dueForService =
        from plane in planes
        where plane.LastServiceDate < DateTime.Now.AddDays(-90)
        select plane;

    return dueForService.ToList();
}
```

At last we come to the Ruby approach:

Download ruby201/iterator.rb

```
require 'date'

def find_planes_due_for_service(planes)
  due_for_service = planes.select do |plane|
    Date.today - 90 > plane.last_service_date
  end
  return due_for_service
end
```

This time, instead of each, we use the select method, which will do the following:

- Create a new, internal collection that is initially empty
- Loop over our planes collection and call our block for each plane it finds
- Push the plane into the internal collection, if and only if the block returns true
- Return the newly created collection

By wrapping up all the boilerplate code, the select method does all the tedious work. Our code must implement only the interesting part, which is to define the rule by which planes will be pushed into the new collection.

We capture the returned collection into a local variable named due_for_service and return it. Eagle-eyed readers will notice that we can take advantage of Ruby's ability to use the last-evaluated expression as a return value, enabling us to simplify our code to just one line:

Download ruby201/iterator.rb

```
def find_planes_due_for_service(planes)
  planes.select { |plane| Date.today - 90 > plane.last_service_date }
end
```

The select method creates a subset of our original collection. The elements in the new collection are of the same type as the original. But just as often, we will want to transform the elements into something else as they are moved into the new collection.

Transforming a Collection

Two of the most powerful and commonly used iterators in Ruby are the

Joe Asks...

When Do I Use do...end Instead of Curly Braces?

Sometimes you will see blocks that use the do...end keywords. Other times you may see curly braces instead. They may seem interchangable, but that's not entirely true.

In Ruby, curly braces have a slightly different binding precedence than a do...end pair. Braces will bind to the rightmost variable or method call on the line. A do...end pair bind to the leftmost. As a result, using braces to delimit a block can have unintended consequences.

A common Ruby idiom has emerged to help avoid unexpected behavior in this regard. A good Ruby programmer (that means you!) will therefore follow these guidelines:

- If the block can be written in one line, use curly braces.
- If the block requires more than one line, use a do...end pair.

us with two methods that do the same thing, because depending on your situation, one name will be obviously more appropriate than the other. These iterators allow us to transform a collection of one kind into a collection of a different kind.

Let's return to our array of airplanes. Suppose we want to generate a list that contains just the FAA identifiers of each plane. In C# 1.1, we would probably do something like this:

Download ruby201/iterator.cs

```
public static String[] GenerateIdentificationList(ArrayList planes)
{
    ArrayList identificationList = new ArrayList();

    foreach (Plane plane in planes)
    {
        identificationList.Add(plane.FaaIdentifier);
    }

    return (String[]) identificationList.ToArray(typeof(String));
}
```

But it gets quite a bit easier with the use of generics in C# 2.0:

Download ruby201/iterator.cs

```
public static List<String> GenerateIdentificationList(List<Plane> planes)
{
  converter = new Converter<Plane, String>(ConvertPlaneToIdentifier);
  List<String> identifiers = planes.ConvertAll(converter);

  return identifiers;
}

public static String ConvertPlaneToIdentifier(Plane plane)
{
  return plane.FaaIdentifier;
}
```

Similarly, in Ruby, without using map or collect, we might do something like this:

Download ruby201/iterator.rb

```
def generate_identification_list(planes)
  identification_list = []
  planes.each do |plane|
    identification_list << plane.faa_identifier
  end
  return identification_list
end
```

We start with an empty array to hold our identification numbers. Then we loop through each plane in our collection and append each plane's identifier to the array. The identification_list is now a simple list of identification numbers.

Fortunately, Ruby once again saves us from impending tedium by removing the boilerplate code into an iterator method. We can supply a block to the map method that transforms (or maps) the elements from the original container into the new one:

Download ruby201/iterator.rb

```
identification_list = planes.map { |plane| plane.faa_identifier }

puts identification_list
```

This code is so short that it might not be obvious how it works. Let's take it one step at a time:

1. We call planes.map. Invisibly, a new, empty collection is created to hold the results of the mapping operation.

2. map passes each element, one at a time, to our block. Just like a function that takes one parameter, our block takes one plane object each time it's called.
3. The block is ultimately responsible for just one thing: returning the value we want to be mapped into the new collection. In our example, we want to reduce the original airplane collection down to a bare list of airplane identifiers. So for each plane we're given, we return the plane's identification number.
4. Each identifier is (again, invisibly) appended to the new collection.
5. When all elements have been visited, the new collection is then returned, which we capture into the identification_list variable.

That's a lot of work being done by one line of code. We will use the map (or collect) method time and again as we develop web applications in Rails.

Reusing Code with Base Classes

Ruby equips the programmer with several options for efficient code reuse. We'll start with a strategy with which .NET developers are already familiar with, inheriting from base classes, so that derived classes can inherit properties and behavior and not have to repeat the same code in the derived classes.

All object-oriented programming languages support the notion of inheritance: a class can derive from another class. In Ruby, we can derive our classes from at most one base class.

Let's take a look at how we set up an inheritance chain in C#, and then we will see how it compares to Ruby's syntax. Let's have our Airplane class derive from a Vehicle class. Every vehicle has a current speed, so let's put that into the Vehicle class.

Download ruby201/baseclass.cs

```
class Vehicle
{
  private int speed;

  // Constructor
  public Vehicle()
  {
    speed = 0;
  }
```

```
  // Read-write property for speed
  public int Speed
  {
    get
    {
      return speed;
    }

    set
    {
      speed = value;
    }
  }
}
```

Next, let's create an Airplane class that derives from Vehicle. C# employs a simple syntax for derivation; we just use a colon between the class names:

Download ruby201/baseclass.cs

```
class Airplane : Vehicle
{
  // Airplane-specific properties and methods go here
}
```

Ruby's syntax is similar, but we use a less-than sign instead of a colon. Here's the Ruby code for both classes:

Download ruby201/baseclass.rb

```
class Vehicle

  # Read-write property for speed
  attr_accessor :speed

  # Initializer
  def initialize
    @speed = 0
  end

end
```

Download ruby201/baseclass.rb

```
class Airplane < Vehicle

  # Airplane-specific methods go here

end
```

Aside from Ruby's more compact syntax for defining properties, the code is quite similar. As you would expect, derived classes behave as

We can ask an airplane for its speed in C# like this:

Download ruby201/baseclass.cs

```
Airplane plane = new Airplane();
plane.speed = 350;
System.Console.WriteLine(plane.speed);
```

And it's just as easy in Ruby:

Download ruby201/baseclass.rb

```
plane = Airplane.new
plane.speed = 350
puts plane.speed
```

Where'd My Interfaces Go?

Static languages in .NET, like C# and Visual Basic, encourage the use of interfaces. Interfaces define a set of behaviors classes can implement. Methods that depend upon interfaces, instead of concrete classes, tend to be more reusable. Let's look at how we use interfaces in .NET and then see how we accomplish the same goal in Ruby.

Perhaps we'd like to say that some vehicles like airplanes and helicopters are flyable vehicles. We could introduce a class called FlyableVehicle in between Vehicle and Airplane, but there's probably not much concrete implementation that would be prove to be reusable among all flyable vehicles. An interface would make more sense in this case. We know that all flyable vehicles need to take off and land, right?

Download ruby201/baseclass.cs

```
public interface IFlyableVehicle
{
    void TakeOff();
    void Land();
}
```

The big value we get from having defined interfaces is the ability to write functions that can accept objects that implement that interface, without knowing the concrete class of the objects themselves. In other words, it enables the creation of this kind of C# code:

Download ruby201/baseclass.cs

```
void AllowAllTakeoffs(List<IFlyableVehicle> flyables)
{
    foreach (IFlyableVehicle flyable in flyables)
    {
        flyable.TakeOff();
```

We'd like to show you the Ruby equivalent of the interface keyword, but we can't. Ruby doesn't have one.

Yes, that's right—Ruby does not advocate the proliferation of interfaces as a way to enable polymorphic code. To understand why, let's think again about why an interface is appealing in a statically typed language. Would this code be dangerous in C#?

Download ruby201/baseclass.cs

```
// Dangerous code ahead?
static void AllowAllTakeoffs(List<Object> objects)
{
  foreach (Object obj in objects)
  {
      Plane flyable = (Plane)(obj);
      flyable.TakeOff();
  }
}
```

This code appears dangerous because this method cannot guarantee that it will succeed. It's been given a list of objects, but there's no way to know those objects are flyable airplanes. We attempt to cast each object to an airplane, but an exception will be thrown if the cast fails and we try to call the nonexistent TakeOff() method.

Without interfaces, there are only two ways we can make sure that this code is safe:

- Practice test-driven development, which requires that we write unit tests to ensure that only flyable objects are passed to AllowAllTakeoffs().
- Use reflection to test for the existence of a public method named TakeOff() before attempting the cast.

Ruby encourages the use of both of these techniques. It's often been observed that interfaces tend to grow in size over time, when polymorphic functions such as AllowAllTakeoffs() tends to care about the existence of only a small minority of these methods (in this case, only one).

Here then is the Ruby approach to writing a method that can safely work with objects that can take off:

Download ruby201/baseclass.rb

```
def allow_all_takeoffs(flyables)
  flyables.each do |flyable|
    flyable.take_off if flyable.responds_to?(:take_off)
  end
```

We use the responds_to? method to find out whether the given object has defined a method named take_off. If so, then we call it.

If you're feeling insecure about this notion of not being able to have the compiler ensure that your objects implement specific interfaces, remember the second part of our equation: unit tests. If you're new to unit testing, be sure to read Chapter 10, *Test-Driven Development on Rails*, on page 191.

It's time to look at another cornerstone of Ruby application structure: modules.

Code Reuse Using Modules

In addition to code inheritance through base classes, the other strategy for code reuse that we will examine is the judicious use of Ruby *modules*. Modules serve two purposes in Ruby: code reuse and namespace identification. This duality is often surprising for newcomers to Ruby. Let's take a look at each of these purposes one at a time.

Modules As Mixins

Ruby classes can derive from one base class, which is a classical way to enable code reuse in object-oriented systems. Sometimes, we want to inherit or "mix in" behaviors that are otherwise orthogonal to the responsibilities of both the class and the base class. Languages that do not support multiple inheritance often provide some facility for "mixin" code reuse. Ruby *modules* fulfill this role.

A module is similar to a class definition, but it cannot be instantiated. A module must by "included into" another class before its methods can be called. In fact, a module is generally nothing more than a set of method definitions, scoped within a particular namespace. Modules can also contain other modules, classes, attributes, and any other valid Ruby code.

Let's look at a simple example of how a module can be used. Suppose we are writing a text editor of some kind. We want our text editor to include some simple spell checking, be able to suggest a synonym for a word, and perhaps display the definition of a word.

We may decide that these tasks lie outside the core responsibility of a text editor, but the text editor would like to be able to support these operations. Moreover, we want this behavior to be reusable outside the

We first wrap the code we want to isolate into a module. We define a module just like we do for a class, substituting the module keyword instead. We then define our methods as we normally would:

Download ruby201/reuse.rb

```
module Dictionary
  def synonym(word)
    # lookup synonym here
  end

  def definition(word)
    # return definition of word
  end

  def spelled_correctly?(word)
    # return true if we find the word in our dictionary
  end
end
```

To mix in our new functionality in our TextEditor class, we use the include keyword and specify which module we want to include into our class definition:

Download ruby201/reuse.rb

```
class TextEditor

  include Dictionary

  # Other TextEditor methods go here

end
```

And presto, our TextEditor now has Dictionary abilities:

Download ruby201/reuse.rb

```
editor = TextEditor.new

# We can call Dictionary methods as if
# they were originally defined in
# the TextEditor class

editor.spelled_correctly?("Airplane")
suggestion = editor.synonym("Delayed")
```

Modules As Namespaces

The .NET namespace keyword is used to partition a system into logical components and to avoid identifier collisions between classes, interfaces, and other named constructs. Namespaces serve as a container

Method Names That End in Question Marks

The spelled_correctly? method in our Dictionary module may look funny. It demonstrates that certain punctuation marks can be added to Ruby method names in order to give more meaning to the method name.

Methods ending with a question mark are, by convention, expected to be a "getter" method that returns a boolean value. (In .NET, it would be common to use the word *Is* as part of the method name, as in IsSpelledCorrectly(), to indicate that the return value will be a boolean value.)

for these constructs and can also contain other namespaces to provide a nested namespace hierarchy.

Ruby modules provide this same kind of namespace facility. Every module serves to define a Ruby namespace. Here's an example of how we might use modules to provide namespace isolation for two classes that would otherwise clash:

Download ruby201/reuse.rb

```
module Utilities
  module Text
    module Dictionary
      # method definitions go here
      # ...
    end
  end
end
```

We've wrapped our Dictionary module inside two other modules. In .NET we use a period to construct a fully qualified class name. In Ruby, however, we use two colons. Here is how we would now include the Dictionary module into our TextEditor class:

Download ruby201/reuse.rb

```
class TextEditor

  # Use fully-qualified name of
  # the Dictionary module
  include Utilities::Text::Dictionary

  # Other TextEditor methods go here
```

Using namespaces becomes essential in large Ruby systems, even in very small components when you want to share your code with others. The Rails framework is composed of a hierarchy of modules. Rails plug-ins are always wrapped in a module that helps identify the source of the plug-in so that class and method name collisions can be avoided between plug-ins.

Ruby Wrap-Up

Over the past two chapters, we've used short Ruby scripts to explore some of the core language features that are the lifeblood of every Rails application: strings, arrays, hashes, iterators, blocks, modules, and duck typing. Ruby is a language that lends itself well to agile development practices, such as unit testing, refactoring, and the DRY principle. We will explore more fully in Chapter 10, *Test-Driven Development on Rails*, on page 191. For those who want to dive deeply into Ruby, we recommend *Programming Ruby* [TFH05] and *The Ruby Way* [Ful06]

But right now, it's time to put our Ruby onto some Rails.

Part II

Rails in Action

Chapter 4

A Bird's Eye View of Rails

Now that we're armed with a basic knowledge of Ruby, we're ready understand how Rails is put together. Rails is installed as a single gem but is actually made up of several smaller components that are installed as separate gems. Each gem is simply Ruby code and may in fact be used outside of Rails if desired (for a full introduction to gems, see Section 12.1, *Getting to Know RubyGems*, on page 232):

- *ActiveRecord* is the object-relational mapper (ORM) that allows our Rails app to talk to databases. We dive into ActiveRecord in Chapter 6, *CRUD with ActiveRecord*, on page 100.
- *ActionPack* consists of *ActionController* and *ActionView*, giving us controller/routing and view/templating capabilities, respectively. We discover ActionController in Chapter 7, *Directing Traffic with ActionController*, on page 128 and ActionView in Chapter 8, *Exploring Forms, Layouts, and Partials*, on page 150.
- *ActiveSupport* is a collection of utility classes and methods that mostly extend Ruby built-in classes to make things easier for web developers. We'll see examples of ActiveSupport in action throughout the book.
- *ActionMailer* gives us the ability to send mail from our application. We'll see an example of this in Chapter 8, *Exploring Forms, Layouts, and Partials*, on page 150.
- *ActiveResource* allows easy communications between Rails applications. A discussion of ActiveResource is outside the scope of this book but is an essential library to look at if you want two or more Rails applications to talk.

Throughout the rest of this book, we're going to build various components of a simple flight reservations system as an example to demonstrate many of the features that the Rails framework has to offer. In the real world, this system would be used by travel agents and airline personnel to keep track of passenger lists, book flights, and maintain flight and airline data.

We'll be developing this application a little bit at a time, refactoring along the way, in keeping with the spirit of Rails' agile development culture.

In this chapter, we'll take a big-picture view of a Rails application. First, we'll take a look at the *stack*, which describes the large-scale software architecture that hosts a web application. We will compare the Rails stack with might be used in a typical ASP.NET hosting setup. We will also delve deeper into how the stack differs in different environments (for example, development vs. production). Finally, we'll explore the several components of the framework and the lifetime of a request to our application.

Comparing Web Architectures

In its simplest form, a web application can be broken down into two components—the *request* that is initiated by the client (likely to be a user sitting at a web browser) and the *response* that is sent back from the web server—with a lot of messy details in between.

ASP.NET WebForms does a lot of hard work abstracting these details away from developers. With ASP.NET, web application development behaves and feels a lot like desktop application development. This is a good thing for a developer who has a lot of desktop development experience; it certainly reduces the barriers to entry for learning to build a web application. Still, at the end of the day, an ASP.NET app is ultimately delivered to the end user as a mix of HTML, CSS, and JavaScript—the only things that the typical web browser is able to natively understand.

Frameworks like Rails, Django, and PHP take a different approach. Instead of creating much abstraction between the developer and the raw code that is given back to the web browser, we're right there in the trenches, entrusting only the dynamic parts of the application to the framework.

To fully understand what we're talking about here, let's first look at the application architecture of a typical web application and what role each of the pieces plays:

- When a client program (like a browser) issues an HTTP request, the *web server* is the first to receive it. It's responsible for handling it and deciding how to best generate an appropriate response. It might know what to do right away, in which case it simply returns a response. This is most often the case when the request is for a static HTML document, image, or one of the more common MIME types. Or, it might need to delegate this responsibility to the application server if dynamic content is required. The web server may also be in charge of other things like logging, bandwidth throttling, and basic authentication.
- The *application server* is primarily responsible for serving dynamic content, often by asking the database server for the information requested.
- The *database server* is where the (no surprise) data lives.

Both ASP.NET and Rails are database-agnostic, with support for all the major RDBMS systems. The real architectural differences between ASP.NET and Rails become more apparent when we compare the role of the web server and the application server in the processing of an HTTP request. The various components involved from the receipt of the request to the final delivery of the response are often referred to as a *web stack*. Let's briefly review how the ASP.NET and Rails stacks compare with each other.

The Typical ASP.NET Stack

In the average ASP.NET deployment, Internet Information Services (IIS) plays the role of both web server and application server. When an HTTP request hits IIS, it decides whether it has the ability to send back an appropriate response. Web stacks like IIS use the URL to identify the file that's being requested.

Sometimes, IIS can directly generate a response without resorting to any application logic. IIS does this by looking at the extension of the file requested. For example, if the extension is that of one of the known MIME types (for instance, .html, .jpg, or .txt), IIS will immediately serve the requested file from disk.

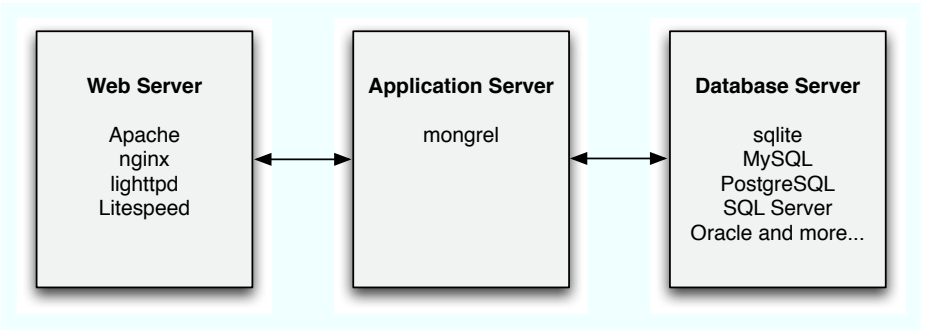


Figure 4.1: The typical Rails stack

If, instead, the extension is .aspx, it will have to delegate the request to the application server, which is in fact just the ASP.NET process. ASP.NET executes the .aspx page code, synthesizing the HTML markup to be sent back to the browser.

While our application is in development, we work on a local machine instead of the production web server. We want to focus our attention on how the application will respond to requests, so we often want to skip the web server part of the equation altogether. ASP.NET and Visual Studio lets us do this by providing a local web server (Web Developer Server) that bypasses IIS just for development purposes. A typical implementation of this usually involves different configuration files, for example, a different web.config file per environment. We can also compile and run our assemblies in different configurations, such as "Debug" or "Release," and define our own custom configurations if desired. We can set breakpoints in our code and set different logging levels per configuration. In addition, we can target a different database in development than we would in production.

All of these concepts we have with .NET have their equivalents in Rails. They're just implemented a little differently.

The Rails Stack

Unless we use an alternative platform for running our ASP.NET apps (say, Mono), ASP.NET is pretty much married to IIS, Windows, and, for the most part, other Microsoft technologies from a web architecture standpoint. With Rails, we have a lot of choices, which is a good thing—

Installing mongrel on Windows

Installing mongrel on Windows is very simple and can be done by typing gem install mongrel at the command line. Once mongrel is installed, the script\server command will use mongrel by default, instead of WEBrick.

but many choices can be daunting when we're first starting out. Let's take a quick look at some of our options, as shown in Figure 4.1, on the previous page.

Apache is certainly the old standby when it comes to web servers, and it has been a stable and proven solution for many developers and organizations. However, it is notoriously difficult to configure and get up and running, especially for those with little experience with it. For this reason, other web server applications like lighttpd, nginx, and Litespeed have grown in popularity, particularly in the Rails world.

As far as application servers go, mongrel is a lightweight, Ruby-based solution (written by Zed Shaw in 2006) that has quickly become the de facto standard for serving Rails applications in both production and development.

The recent introduction of Passenger[1] (aka mod_rails) gives us another option. A module that runs on top of the popular Apache web server, Passenger allows for simple, "PHP-like" deployments of Rails applications. Because of its speed and ease of use, Passenger is gaining popularity quickly in the Rails world.

In general, we want to develop our application in an environment that mirrors our production environment as closely as possible. So although it's not often necessary to run a web server application like Apache in our development environment (although it can be very helpful to do so when testing things like caching strategies and security), it is generally a good idea to run mongrel in development. Doing so is easy enough; script/server will use mongrel as the default if the gem is installed. To install the mongrel gem, simply gem install mongrel from the command prompt.

1. http://modrails.com/

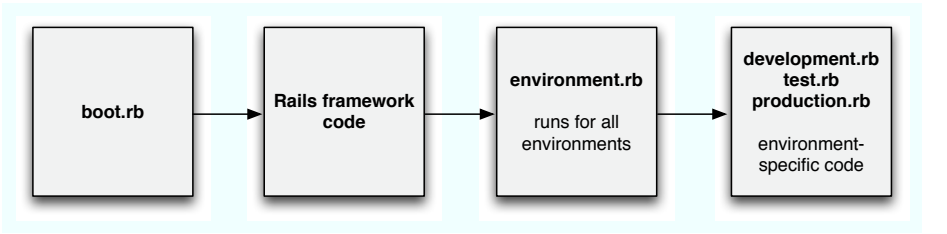


Figure 4.2: The Rails boot/initialization process

What Happens When a Rails Application Starts Up

Way back in Section 1.6, *Fire It Up*, on page 26, we saw that the script\server command starts up our Rails application server. Let's now take a look at what really happens when we do that (see Figure 4.2).

It's important to realize that Rails is nothing more than a (rather large) Ruby program. Recall that Ruby programs do not have a Main() function or any other predefined starting point. The Ruby interpreter simply starts at the top of the first file it's asked to process and executes all the code from top to bottom.

The first Ruby script that is the starting file for all Rails applications is config/boot.rb. This file, which is generated for you when you create a new Rails application and is not meant to be modified by hand, loads all the dependencies needed to start up our Rails application, including the loading of the Rails gem itself (unless you've "frozen" a specific version of Rails into your application; see Section 12.4, *Freezing a Rails Application*, on page 246).

boot.rb then goes on to execute all the code in config/environment.rb, which is code we want to run regardless of which "environment" we're in. Rails has a notion of distinct runtime environments. By default, it knows about three: development, test, and production. We often need different settings depending on the environment, like which database should be used and how much debugging and logging support we want.

For settings that are common to all environments, environment.rb is the best place to put code that is required throughout the application that is not already part of Rails core. For instance, let's pretend we installed a gem called xyz and wanted to use it in every model and controller

in our application. Instead of putting require 'xyz' in every model and controller file, we could put that in our environment.rb file instead.

Lastly, Rails will look for environment-specific code that should be executed. Rails defined a convention for how it should locate this code by searching for a file named config/environments/{environment}.rb, for example, config/environments/development.rb, config/environments/test.rb, or config/environments/production.rb.

Remember that because Ruby is a scripting language, whatever code comes last wins. So, a method that is defined in environment.rb will be overridden by a method with the same name if it's defined in our environment-specific code. Likewise, since the environment.rb and environment-specific files are loaded after the Rails framework code, we can override anything in Rails at this point as well.

As you may have already guessed, this concept of a particular Rails runtime environment plays an understated but very important role in Rails development, so let's take a closer look at it.

Environments in Rails

Environment is the Rails term that refers to what we're doing with our code at the time of execution. And, just like in .NET, having different environments allows us to define different database connections, variables, code, and any other things that may differ between development, testing, integration, staging, and production setups.

By default, three environments are defined in a Rails application—*development*, *test*, and *production*. Environments in Rails are defined by simple Ruby configuration files in the config/environments directory. Here is an excerpt from the three files:

Download http/development.rb

```
config.cache_classes                          = false
config.action_controller.perform_caching      = false
config.action_mailer.raise_delivery_errors    = false

FOO = "bar"
```

Download http/test.rb

```
config.cache_classes                          = false
config.action_controller.perform_caching      = false
config.action_mailer.raise_delivery_errors    = false
```

Download http/production.rb

```
config.cache_classes                              = true
config.action_controller.perform_caching          = true
config.action_mailer.raise_delivery_errors        = true

FOO = "baz"
```

Let's now take a look at the first line of each file. With this line, we are telling Rails that in development and test we don't want to cache classes. Caching classes speeds Rails' performance up considerably but requires that we restart the application server any time code changes are made. This is probably what we want in production, but we certainly don't want that in development. Along the same lines, we want caching turned on and want errors to be raised if we can't send mail in production, but we don't care about those things in development or test. There are a lot of configuration options that can be set per environment; check out the Rails documentation to see all of them. We can also set our own variables, like we've done in the last line of each environment file.

In Visual Studio, we can use different configuration files to define custom environments if we want. Adding our own custom environments in Rails is just as simple, except that, instead of XML configuration files, they're written in Ruby. For example, if we wanted a *staging* environment, we would simply add a staging.rb file to the config/environments directory and set the appropriate options that we wanted.

We specify the environment we want to run when starting up the application server. For example, to start up our application in production using script\server, we would simply type this:

```
c:\dev\flight> ruby script\server -e production
```

This starts up Rails on the default port (3000) in the production environment. The default environment is development, so if we don't specify one using the -e argument, that's what we'll get.

One last thing about environments: the current environment is always available for us to use in code by accessing the RAILS_ENV constant. Here is just a simple example of how we might use it to display a footer on a page:

Download http/_footer.html.erb

```
<div id="footer">
  <% if RAILS_ENV == 'development' -%>
```

```
  <% else -%>
    &copy; 2008 Super-Mega Corp.  All rights reserved.
  <% end -%>
</div>
```

Here, we are using the value of RAILS_ENV to show a different footer to those running in development mode than in any other environment.

Configuring Data Access

If we were talking typical ASP.NET, we'd probably use our web.config file to configure access to our database by supplying a different connection string per environment. In Rails, information about data access across all environments is contained in a single file, the config/database.yml file. Here's an example:

Download http/database.yml

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  timeout: 5000

test:
  adapter: sqlite3
  database: db/test.sqlite3
  timeout: 5000

staging:
  adapter: mysql
  database: flight_staging
  username: foo
  password: bar

production:
  adapter: mysql
  database: flight_production
  username: deploy
  password: secret
```

Each of our environments *must* have an entry in this file to let the application know how we'd like to access the data for that environment. Here, we are telling our application that in development and test we'd like to use a simple SQLite file-based database and that in staging and production we're going to use MySQL. We've also supplied the credentials necessary to access these databases.

Joe Asks...

What Is YAML?

The database.yml file is in a format called YAML (rhymes with "camel" and stands for YAML Ain't a Markup Language or Yet Another Markup Language—depending on who you ask!). YAML is a data serialization format that's designed to be very human-readable and editable. Data in YAML is represented in a simple hierarchical structure, and indentation matters. You can find more information about YAML at the YAML website.*

*. http://yaml.org

Receiving HTTP Requests

We talked earlier about how some web frameworks like IIS use the URL to basically determine which file should be processed by the server. Rails takes a different approach. Some files, like images, are still handled directly by the web server. But any time the request is not matched to a physical file on disk, it gets handed off to Rails' *dispatcher*. The dispatcher does not try to locate a file that matches the URL's directory structure. Instead, the URL maps to a specific Ruby method in our Rails application.

It probably seems funny to think about a URL as just a way to invoke a method in our application, but that's exactly what Rails does. To determine which method should be called, the dispatcher first consults our *routes* to determine which *controller* (a class in our app/controllers directory) and which *action* (a public method of our controller class) to send the request off to. We will explore routes in depth in Chapter 7, *Directing Traffic with ActionController*, on page 128. For now just know that unless a URL represents a nondynamic HTML file or asset, it must be translated into a method call. Not just any method can be called, only public methods of controller classes are callable by the dispatcher.

In Rails-speak, public controller methods are called *actions*. Once the controller dispatcher identifies the controller class and action to call, it instantiates the controller, and the action method is called.

Our method can do anything it wants. After our method runs, Rails

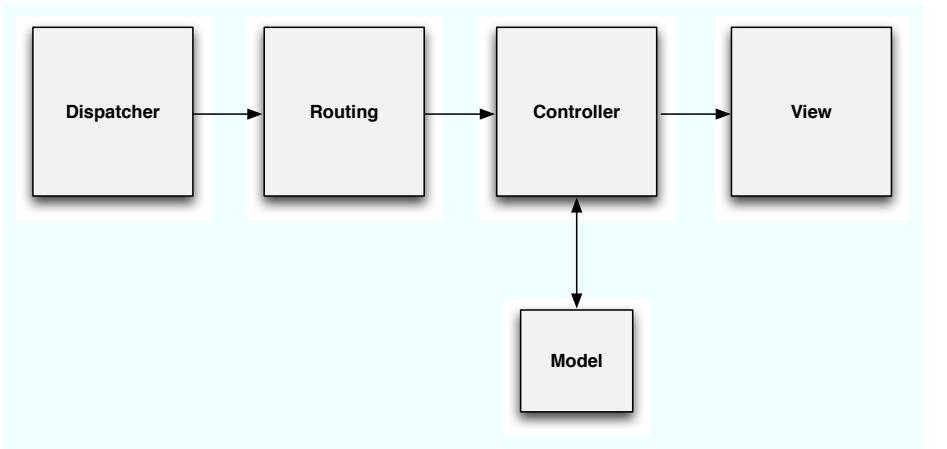


Figure 4.3: The journey of an HTTP request in a Rails app

the controller talks to one or more model classes to send or retrieve some data, and a view containing an embedded Ruby script is rendered to the end user (see Figure 4.3).

Generating HTTP Responses

Most of the work involved in receiving HTTP requests is behind the scenes in Rails, just as it is in .NET. The responsibility for generating the response, however, is placed squarely on the shoulders of the application. Most of the remainder of this book will focus on generating HTTP responses of various kinds and in various ways, but we'll first outline the primary concepts you'll need to know in order to successfully switch from a .NET mind-set to the Rails way of thinking.

Controllers

In a *Model-View-Controller* (MVC) architecture, the controllers are like first responders to an emergency. They are the first ones that get told of the call, and they are the ones that have to decide what to do about it. In Rails, controllers are normal Ruby classes that inherit from ActionController::Base or (more often) the generated ApplicationController class.

Controllers have public action methods that can be called from the dispatcher, as well as their own protected and private methods. Before

an action method returns, it is responsible for generating a response in a format that is compatible with what the client expects. For a web browser that's requesting HTML, we need to generate HTML; for an RSS reader that's expecting Atom-formatted XML, we need to generate the appropriate XML document; and so on.

Remember too that Rails is best suited for database-backed web applications. As a result, action methods typically follow this pattern:

1. Manipulate the domain model in some way. This might be selecting some data from the database, inserting data into the database based on incoming parameters, or otherwise using one or more classes from the app/models directory.
2. Look at the response format that's been requested by the client.
3. Generate a response in the correct form by rendering a view template.

Controllers generate a response by explicitly calling the render() method before they leave scope, by redirecting the browser to a different page, or by allowing Rails to implicitly render a view template that is associated with the action. For all that controllers do, they are surprisingly free of code. Controllers should delegate most of their "intellectual work" to models and do little more than provide data to be consumed by the views.

Models

Models, on the other hand, should constitute the bulk of your application code. They provide a persistent storage mechanism to your database and should also define all business logic. Any time there's a decision to be made that has nothing to with the format of the request or the response, the model should be the one making the decision. When there's a need to query the database, it's the model that should be issuing the query. Models should be the "brain" that operate on your application's database, determine and change the state of other objects, and coordinate the overall logic of your application.

There's a philosophy often referred to as "Fat Models, Skinny Controllers." As a rule of thumb, if the code in your controller actions are getting messy or unwieldy, especially if you're touching data in a database, then whatever you're doing probably belongs in your models instead.

Put another way, if you didn't have a web browser handy, could you manipulate your application logic from an irb session? If not, then some of your application logic has been put into a controller or view when it belongs in a model instead.

Views

Views are text files in app/views that can be pure HTML; HTML templates with embedded Ruby, XML, CSV; or any other kind of text file. Views are identified by MIME type and how they should be interpreted or built. Rails will automatically detect both of these characteristics if we name our view files according to a specific convention:

```
action.mimetype.builder
```

For example, show.html.erb is a template named show that will be rendered automatically after the show action method returns; it is appropriate only for HTML requests and must be processed as an embedded Ruby template file.

This MVC dance takes a little getting used to, but this forced separation of concerns pays big dividends as your application grows. The independence of views from models means that once we have an HTML representation of our data, we can create an XML view of the same data by adding a new kind of view template and allow the controller to discover it automatically. Similarly, we can be free to enhance and evolve the public interfaces of our models with little or no impact on the views but only upon the controllers that uses them.

In this chapter, we've brushed elbows with the Rails framework and taken a peek at what a Rails application is actually made of. Next, we'll take a look at some of Rails' conventions and how following them helps make application development, dare we say, enjoyable.

Chapter 5

Rails Conventions

One of the important things to know about Rails is that it's opinionated. It certainly seems strange to talk about a software development framework as having an opinion, but it is this quality that truly makes Rails what it is. If you build your app the way Rails expects you to build it, it can be a pure joy. And if you want to do things a little differently, it can sometimes be a challenge.

One of the pillars of the Rails framework is the notion of "convention over configuration." Rails strives not to overwhelm the developer with options; instead, it promotes the use of sensible defaults for the majority of cases, with the ability to override these defaults—if you really want. In this chapter, we'll closely examine what some of these conventions are and how to approach problems "the Rails way." Many of these defaults come as a result of Rails' implementation of both the MVC and REST design patterns, and we'll examine both of these implementations in detail.

MVC: Separating Responsibilities in Your Application

.NET developers should be familiar with the idea of splitting up parts of an application into different layers in order for code to make more sense and to make it easier to maintain. Most professional .NET application developers are probably accustomed to doing this by creating separate libraries for their data, business, and presentation logic. ASP.NET developers are used to the framework doing much of this for them by default, breaking a web form, for example, into ASPX/ASCX and its corresponding *code-behind*.

Rails, by convention, implements a similar separation of responsibilities. The goals are the same—to logically partition an application into smaller pieces that are much easier to understand. To accomplish this, Rails employs a very well-known software design pattern: Model-View-Controller (MVC). It goes something like this:

- Models are the classes that represent your business domain and that are responsible for communicating with your data. In Rails, this means the tables in your database.
- Views represent your presentation layer, for example, HTML and JavaScript.
- Controllers are responsible for connecting the models and views and managing the flow of the application.

If you take a peek in the app folder of any Rails application, you'll find the models, views, and controllers folders inside. Here's where the magic of Rails happens. Simply put your Ruby code inside these folders, conform to the conventions of file and class naming (that we'll discuss in a moment), and that's it—you're already following Rails' MVC pattern.

Bear in mind that you aren't required to follow every one of these conventions; Rails allows you to override most of them, if you want. But in general, it will be a lot easier and more productive to simply do what Rails expects and just ride the wave.

Model Conventions

The granddaddy of all the modules that make up Rails is ActiveRecord's built-in object-relational mapping (ORM) tool. And, as is often the case with granddaddies, it is also the most opinionated one in the family. Model classes in Rails, all of which inherit from the ActiveRecord base class ActiveRecord::Base, map one-to-one to a table in your database. They belong in the app/models directory and should follow these simple rules:

- There should be one class per file.
- The class name should be singular and camel-cased, for example, Person or InvoiceItem.
- The corresponding table name should be lowercased, plural, and underscored, for example, people or invoice_items.
- The corresponding table is always identified by an autoincrementing integer field called id.
- Column names should also be lowercased

- The filename should be the lowercased and underscored version of the class name, for example, person.rb or invoice_item.rb.

To represent the relationships between tables and columns in your application, ActiveRecord provides a set of methods collectively called *associations*. These are used to express things like "flight belongs to airline" or "flight has many passengers." Here's an example of a set of ActiveRecord classes that take advantage of the basic association methods:

Download conventions/flight.rb

```
class Flight < ActiveRecord::Base
  has_many :reservations
  has_many :passengers, :through => :reservations
  belongs_to :airline
  has_one :manifest
end

class Reservation < ActiveRecord::Base
  belongs_to :flight
  belongs_to :passenger
end

class Passenger < ActiveRecord::Base
  has_many :reservations
  has_many :flights, :through => :reservations
end

class Airline < ActiveRecord::Base
  has_many :flights
end

class Manifest < ActiveRecord::Base
  belongs_to :flight
end
```

As you can see, association methods are a lot like representing foreign keys in a database diagram—in Ruby code. Figure 5.1, on the next page, shows what the corresponding database diagram might look like.

The power of association methods stems from the power of Ruby itself. Because of the dynamic nature of the language, these methods actually add (or *mix in*) behavior to the class in which they're defined. For instance, now that we've implemented the has_many method in our Flight class, we can now create an instance of Flight and do things like flight.passengers or flight.airline and get back an array of passengers on that flight and the flight's airline, respectively. This all happens

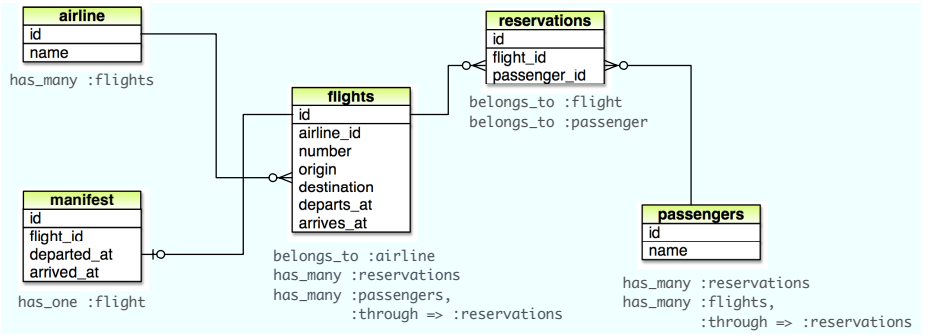


Figure 5.1: Database diagram for flight reservation example

remember with the conventions for association methods, but over time you'll learn that it's as natural as following the rules of the English language:

- has_many represents a zero-to-many relationship between the parent class in which it's declared and the child class. It must take at least one parameter, which is the pluralized version of the child class, and the child table must have a field that starts with the singular, lowercased parent class name and ends with _id. For instance, as shown in Figure 5.1, a Flighthas_many :reservations, so the reservations table must contain a flight_id column.
- has_many can take an option, :through, which then allows it to represent a many-to-many relationship. For instance, the reservations table acts as a join table between flights and passengers, so it must have both flight_id and passenger_id columns.
- has_many is often reciprocated by a belongs_to method call on the child class. In the previous example, "an airline has many flights" and "a flight belongs to an airline," so the flights table must contain an airline_id column.
- A has_one association is much like a belongs_to in that it represents a one-to-one relationship in your database. The key difference is whether the foreign key column lives on the parent or on the child table. If "a flight has one manifest" as in the example, then the manifest table must contain a flight_id column. If, instead, "a flight belongs to a manifest," then the flight table would have a manifest_id column.

Finally, keep in mind that because of the strict implementation of MVC in Rails, your models should have no knowledge of stuff like what URL a user has typed into her browser, what GET or POST parameters are being passed around, or what URL the user should go to next—these are jobs for the controller.

Controller Conventions

Controllers have all the knowledge about, and control over, the flow of request to response in a Rails application. Here are some key things to know:

- The controller's class name should be camel-cased and pluralized and be followed by the word Controller, for example, OrdersController or ProductSearchesController.
- As with models, the filename should be the lowercased and underscored version of the class name, for example, orders_controller.rb or product_searches_controller.rb.

A public method of a controller class is known as an *action*, for example, new, edit, show, and so on. Each one of these methods connects the data in your model layer to what you'd like the end user to see.

Enough talk, here's an example:

Download conventions/flights_controller.rb

```
class FlightsController < ApplicationController

  def show
    @flight = Flight.find(params[:id])
  end

end
```

Here, we see that the FlightsController has a single action, show, and that it has very little implementation other than finding a flight record based on an ID that's been passed in and assigning that record to an *instance variable* called @flight. Notice that we haven't told the action to go to a particular page or otherwise render any HTML. This is because, unless explicitly told not to, Rails will simply render the view with the same name as the action.

View Conventions

Views live in the app/views directory, and each file goes into a subdi-

every view of the OrdersController is inside the app/views/orders directory. View files themselves follow a simple and logical naming convention—the name of the action, followed by the type of file, followed by the rendering engine, all separated by a dot. For example, a view file named edit.html.erb in the orders directory tells us that this view corresponds to the edit action of the OrdersController, that it is written in HTML markup, and that it should use the ERb engine to do the rendering.

Here is an example view for the show action we implemented in the FlightsController:

Download conventions/show.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <title>Showing Flight Number <%= @flight.number %></title>
</head>
<body>
  <p>Flight Number: <%= @flight.number %></p>
  <p>Departs <%= @flight.origin %> at <%= @flight.departs_at %></p>
  <p>Arrives <%= @flight.destination %> at <%= @flight.arrives_at %></p>
</body>
</html>
```

Here, we see some simple HTML that will give us a page showing our customer's name. Since the view has access to any instance variable set in its corresponding controller method, we're able to use the @flight variable in our markup. Rails gives you access to any of the columns of a model's corresponding table as well—calling @flight.number will automatically return the value of the number column that belongs to the Flight record returned.

Generators

Luckily, you don't have to remember all this yourself. Rails comes with a set of scripts, called *generators*, that (unsurprisingly) generates code so you can skip past the mundane tasks of creating files and having to remember where to put them and get to work instead!

Let's take the model and controller generators for a spin, but remember that, by convention, models are singular and controllers are plural.

```
c:\dev\flight> ruby script\generate model flight
  exists app/models/
  exists test/unit/
  exists test/fixtures/
  create app/models/flight.rb
  create test/unit/flight_test.rb
  create test/fixtures/flights.yml
  create db/migrate
  create db/migrate/001_create_flights.rb
c:\dev\flight> ruby script\generate controller flights
  exists app/controllers/
  exists app/helpers/
  create app/views/flights
  exists test/functional/
  create app/controllers/flights_controller.rb
  create test/functional/flights_controller_test.rb
  create app/helpers/flights_helper.rb
```

Generators are a great timesaver. They also teach you a lot about where things belong and what Rails expects you to do next. For instance, in addition to the model and controller classes being created and placed in the proper directories, you'll notice that a bunch of files get created under the test directory as well. We'll learn more about what this is for in Chapter 10, *Test-Driven Development on Rails*, on page 191.

Putting It to REST

Now that you know all about how Rails implements MVC, there's one other key topic to understand before we can get to building that next great app of yours. In addition to baked-in MVC, Rails is tailor-made to allow for rapid development of RESTful web applications. To understand what that means to you, as a developer, we have to first dive into a little history lesson.

Representational State Transfer (REST) is a fancy term that simply refers to a particular style of programming. It's not something that's unique to Rails at all—the concept of REST was coined by Roy Fielding back in Internet medieval times (the year 2000, that is). If you're new to the concept of REST, the best way to get started is to think of your application as a collection of *nouns* and *verbs*.

Thinking way back to third-grade reading class, I think I can come up with a few nouns:

- Dog
- Car

- Shoe
- Truck

Nouns aka Resources

Of course, your web application is probably not made up of dogs, cars, shoes, and trucks. It's most likely made up of things like customers, orders, products, and shopping carts. In our flight reservations system, it's probably going to be comprised of, among other things, flights, reservations, and passengers. So, that's the first step in designing your application in a RESTful manner—to identify your nouns, otherwise known as *resources*. To think RESTfully is to stop thinking of your web application as a bunch of *pages* and begin to see it as a collection of resources.

Verbs

Although you're allowed to have an infinite number of nouns/resources in your application, according to the REST gospel, only a short list of verbs is allowed; you're allowed only to create, read, update, and delete (CRUD) your resources.

You might initially think that your application, in your particular business domain, won't fit into this model. But, most likely, you'll find that it's not only possible but also extremely liberating. Furthermore, HTTP, the native language of the Web, has been implemented using a REST-style pattern. We web developers are all familiar with the GET and POST methods that HTTP implements. Well, GET and POST are HTTP's way of implementing the *read* and *create* verbs. And although HTML understands only the GET and POST methods, HTTP also implements PUT and DELETE methods, which are meant to represent the *updating* and *deleting* of your resources.

OK, we know. You're not buying this yet. Let's go a step further.

A Practical Example

If you've been doing object-oriented programming at all, embracing REST is all about thinking about your application a little differently than you might be used to doing. If not REST, then you've probably been doing what's known as *RPC-style programming*. That is, you have a bunch of classes (nouns), and each of those classes has some arbitrary number of methods (verbs) on it where you take action on a particular

instance of that class. For instance, in our flight reservation system, your Flight class might implement the following methods:

- **GetFlight()**
- **AddFlight()**
- **UpdateFlight()**
- **TakeOff()**
- **Delay()**
- **Cancel()**

This approach works great in many OO programming scenarios, but wouldn't it make more sense, in a web programming context, to make your application API more web-like?

Let's think about what you'd do if your boss or client approached you and wanted you to implement the flight cancellation use case in your application. Your first thought might be to create a page for it that does the job. If you were using ASP.NET, you might create something like CancelFlight.aspx and access it via a URL like http://localhost/CancelFlight.aspx. Or, in Rails, you might perhaps create an action on your FlightsController called cancel and make it accessible via http://localhost/flights/cancel?flight_id=123. This all makes perfect sense, but the REST way takes a different approach. REST, in contrast to RPC, emphasizes the diversity of nouns, not verbs—just like HTTP does. So instead, the REST way to do this would be to create another resource—perhaps called FlightCancellations—and to perform a *create* on the FlightCancellations resource.

What Does This All Buy Us?

Why subject ourselves to these types of constraints? Going along with the spirit of Rails as a whole, the REST way is really all about not having to think about the petty implementation details of your application. Rather, it allows you to focus on your website's business domain and unique features and what you'd actually like your application to do. One of the hardest things for a developer, at times, is to just get started on a new project. With REST, again, there's not much thinking involved. Simply identify your resources, and get coding. Here are some other benefits of REST that are along the same lines:

- Consistency. You can sleep easy at night knowing that all your resources have a common API. It sure makes your application easy to maintain, because you won't be hunting for hours trying to find

that weird method that you or a teammate wrote last year. What if your boss asks you to implement the ability to "uncancel" a flight? It's easy enough to do—simply add a *delete* verb to your *flight cancellation* noun.

- If you ever decided to expose a public API, those who consume it wouldn't need to keep track of all the crazy actions you've made up—they would need to know only which resources they're allowed to CRUD.

- What if you needed to keep track of all flight cancellations? No problem. The create, read, update, and delete verbs sound an awful lot like INSERT, SELECT, UPDATE, and DELETE in SQL, if you ask us. It's a very natural mapping that requires very little work on your brain's part to understand.

REST is a design pattern. As with all design patterns in software, it is a suggested framework for how you might write and organize your code. Simply put, you don't need to conform to REST practices 100% of the time in order to build web applications with Rails. But as with everything else in Rails, you'd be going against the grain, be a lot less productive, and be running into a few more challenges.

How REST Works in Rails

There's a really easy way to get started with REST in Rails—by using the built-in *scaffold* generator. Scaffolding in Rails is lot like scaffolding in construction terms. It provides a skeleton of an application that holds everything in place while you build what's going to really become the finished product. And, just like real scaffolding, it's not meant to remain in place for production use; it's there to guide you along the way and provide something to stand on until the real thing is completed. Most of all, it's an excellent learning tool in helping you understand how a Rails application is intended to be built in a RESTful manner. To see what I mean, let's create a new Rails project for our flight reservation system:

```
c:\dev\> rails flight
```

Next, we'll use the scaffold generator to create a Flight model, as well as a FlightsController within your Rails project:

```
c:\dev\flight\> ruby script\generate scaffold flight
flight_number:integer departs_at:datetime arrives_at:datetime
origin:string destination:string
```

Rails Action	Which Represents...	URL	HTTP Verb
index	A collection of flights	/flights	GET
show	A flight	/flights/123	GET
new	A form for creating a new flight	/flights/new	GET
create	Where you submit the new flight form	/flights	POST
edit	A form for editing an existing flight	/flights/123/edit	GET
update	Where you submit the edit flight form	/flights/123	PUT
destroy	Where you destroy (delete) an existing flight	/flights/123	DELETE

Figure 5.2: The seven RESTful Rails actions

What we are telling the scaffold generator to do here is to create a model, a controller, and some views to perform basic CRUD operations for a database table named flights. The scaffold generator also takes field names and types as parameters, and we are taking advantage of that capability here to add a flight number, departure/arrival time, and origin and destination cities. Note that the ID field is always created by default, so there is no need to specify it through the generator command.

Take a look at the controller that's generated, and you'll see that Rails takes the four verbs we've been talking about (create, read, update, and delete) and maps them to seven controller actions (the public methods of the controller), as shown in Figure 5.2.

Notice that the URLs for the show, update, and destroy methods are identical. That's OK because they each use different HTTP verbs, which distinguishes them from each other.

There is one more important thing that the scaffold generator does. Rails routing, as defined in the routes.rb file, also needs to know about your resources in order to map them properly to the aforementioned HTTP verbs.

Joe Asks...

Wait a Minute...Didn't You Say HTML Doesn't Understand PUT and DELETE?

That's right, it doesn't. Even though HTTP supports the PUT and DELETE methods just fine, HTML doesn't—it does only GET and POST. So, in order for Rails to conform to the HTTP convention in the interest of being RESTful, a hidden form field is used to simulate PUT and DELETE requests. You don't have to really worry about how this happens as long as you use Rails' built-in form helpers, but it's a good thing to know anyway.

Running the scaffold generator does this for us, but let's just take a peek:

Download conventions/routes.rb

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

  map.resources :flights

end
```

That single line of code in our routes file does a whole lot of work for us. It lets Rails know that flights is a resource in our RESTful design; defines all seven URLs that we need to list, show, create, update, and delete our resource; and creates helper methods that allow for quick access to those URLs. That's not bad for one line of code!

From this chapter, you should now have a solid understanding of the main Rails conventions—MVC and REST. But this is in fact only the tip of the proverbial iceberg. From here, we're ready to get cranking on our flight reservations system and apply these concepts to writing a real-world Rails app.

Chapter 6

CRUD with ActiveRecord

ActiveRecord is the part of Rails that's responsible for talking with the database of our application. Whether that database is MySQL, SQL Server, SQLite, Postgres, or one of the myriad of other RDBMSs that Rails supports, ActiveRecord allows us to tell it what to do in a single language, Ruby.

In this chapter, we'll explore the sweet spot of Rails—the creating, reading, updating, and deleting (CRUD) of data—using ActiveRecord. We've already talked about some of the conventions that we'll need to follow to take full advantage of ActiveRecord; now, we'll put this knowledge to the test by using Rails to do a lot of the same things we're used to doing in .NET. In addition, we'll be concentrating much more on the capabilities of ActiveRecord and the model side of things and only minimally on the controller and view parts of the Rails world. We'll take a much closer look at the controller and view in the next chapter.

Displaying a Grid of Data in a Table

One of the features that you'll see in almost any web application is a collection of records in a database displayed in a human-readable grid/table format. Let's say we'd like display a simple table that shows all passengers in our flight system from our passengers table.

How You Might Approach It in .NET

There's no built-in ActiveRecord-like ORM mapper in .NET, although several open source and commercial packages are similar in style and function. For the most part, though, in out-of-the-box ASP.NET Web-Forms, you're looking at SQL (whether it's in a stored procedure or

that contains your data and then displaying it with a control like the GridView. The following illustrates such an approach:

Download crud/Passengers.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs"
    Inherits="_Default" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Passengers</title>
</head>
<body>
    <form id="form1" runat="server">
    <asp:SqlDataSource ID="SqlDataSource1" runat="server"
        ConnectionString="<%$ ConnectionStrings:ConnectionString %>"
        SelectCommand="SELECT * FROM passengers"></asp:SqlDataSource>
    <asp:GridView ID="GridView1" runat="server"
        AutoGenerateColumns="False" DataKeyNames="id"
        DataSourceID="SqlDataSource1">
        <Columns>
            <asp:BoundField DataField="name" HeaderText="name"
             SortExpression="name" />
            <asp:BoundField DataField="address" HeaderText="address"
                SortExpression="address" />
            <asp:BoundField DataField="seating_preference"
             HeaderText="seating_preference"
                SortExpression="seating_preference" />
        </Columns>
    </asp:GridView>
    </form>
</body>
</html>
```

Here, we create a simple GridView that contains our data by binding the control to a SqlDataSource object that queries for all passengers. Within the GridView definition, we declaratively let the control know which columns we want to display. When we start our application up, this code will translate into HTML for display in the web browser.

The Rails Way

It's pretty trivial to create the same interface using Rails' scaffolding, as we did in the previous chapter. But we're going to do everything by hand this time so we can dig a little deeper into how Rails works behind

As we've already learned, three parts are involved in building the same read-only view of a data collection as the ASP.NET GridView approach: the model, the view, and the controller. Let's quickly use a generator to get all the files we need before exploring our application further.

```
c:\dev\flight> ruby script\generate resource passenger
    exists  app/models/
    exists  app/controllers/
    exists  app/helpers/
    create  app/views/passengers
    exists  test/functional/
    exists  test/unit/
dependency  model
    exists    app/models/
    exists    test/unit/
    exists    test/fixtures/
    create    app/models/passenger.rb
    create    test/unit/passenger_test.rb
    create    test/fixtures/passengers.yml
    exists    db/migrate
    create    db/migrate/20080722201710_create_passengers.rb
    create  app/controllers/passengers_controller.rb
    create  test/functional/passengers_controller_test.rb
    create  app/helpers/passengers_helper.rb
     route  map.resources :passengers
```

Now that we have the files we need, let's make a few edits to the generated code to get our application working the way we want it.

The Passenger Model

We'll concentrate on the model layer first. Notice that the generator created a migration file, db/migrate/20080722201710_create_passengers.rb. Let's modify this migration to reflect the schema we'd like to create for the passengers table, where each passenger will have a name, address, and seat preference.

Download crud/20080722201710_create_passengers.rb

```
class CreatePassengers < ActiveRecord::Migration
  def self.up
    create_table :passengers do |t|
      t.string :name, :address, :seat_preference
      t.timestamps
    end
  end

  def self.down
    drop_table :passengers
  end
```

Now we'll run the migration:

```
c:\dev\flight> rake db:migrate
(in c:\dev\flight)
== 20080722201710 CreatePassengers: migrating =======
-- create_table(:passengers)
   -> 0.0026s
== 20080722201710 CreatePassengers: migrated (0.0029s) =====
```

Great. The passengers table is now created based on the information we've provided in the migration file. With no SQL. Just Ruby.

The Rails Console

Now that we've created our table, it's a good time to mention that Rails ships with a handy utility called console that is a lot like irb, except that it runs within the context of our web application. This means that, in addition to evaluating Ruby interactively, you can also do things like manipulate your application's database, make simulated requests, and execute other Rails-specific commands that wouldn't otherwise be available with vanilla irb. Let's fire it up now:

```
c:\dev\flight> ruby script\console
Loading development environment (Rails 2.1.0)
>>
```

From the console, we're able to learn a lot about the capabilities of the ActiveRecord library. Let's use it now to create some sample records in our passengers table:

```
>> Passenger.create(:name => 'John Doe', :address => '123 Main St',
:seat_preference => 'Aisle')
=> #<Passenger id: 1, name: "John Doe", address: "123 Main St",
seat_preference: "Aisle", created_at: "2008-01-15 15:49:24",
updated_at: "2008-01-15 15:49:24">
```

The create method is a *class* method of the class Passenger that accepts a single parameter—a Hash where the keys are column names (as symbols) and the corresponding values. We could just easily do it the long way and instantiate a new Passenger object, assign the values we'd like, and call the save method:

```
>> passenger = Passenger.new
=> #<Passenger id: nil, name: nil, address: nil, seat_preference:
nil, created_at: nil, updated_at: nil>
>> passenger.name = 'Jane Doe'
=> "Jane Doe"
>> passenger.address = '123 Main St'
=> "123 Main St"
```

```
>> passenger.seat_preference = 'Window'
=> "Window"
>> passenger.save
=> true
```

Both approaches yield the same result: a new record gets created in the passengers table with the values you've specified for each column. Some developers like the one-line brevity of the create method, and others enjoy the clear intention expressed by the multiline approach. It's up to you to decide which of the styles you like better.

It's Just SQL

By now, you have probably realized that ActiveRecord, as fantastic a library as it is, is not magic. All it really does is create SQL statements behind the scenes, with the added bonus of being completely database-agnostic. That is, it understands the various idiosyncrasies of various database engines and adjusts the generated SQL accordingly.

The closer your relationship with ActiveRecord, the more productive Rails developer you'll become. And the key to a deeper and more meaningful relationship with ActiveRecord is knowing exactly what SQL is being generated when you call methods like create. Fortunately, the raw SQL is exposed through the log file of a running Rails application, and furthermore, you can also examine it in the console. By default, the log output of any commands you execute in the console go straight to your development log (located at log\development.log), but you can issue a one-line command to override this and direct the log output to *standard output* instead. We'll do this now so that we can inspect the SQL that ActiveRecord creates quickly:

```
>> ActiveRecord::Base.logger = Logger.new(STDOUT)
=> #<Logger:0x105a608 @default_formatter=#<Logger::Formatter:0x105a5e0
 @datetime_format=nil>, @progname=nil,
 @logdev=#<Logger::LogDevice:0x105a590, @filename=nil,
 mutex=#<Logger::LogDevice::LogDeviceMutex:0x105a52c
 @mon_entering_queue=[], @mon_count=0, @mon_owner=nil,
 @mon_waiting_queue=[]>, @dev=#<IO:0x2e7d4>, @shift_size=nil,
 @shift_age=nil>, @level=0, @formatter=nil>
```

So when we create a new record, we see the underlying SQL right away:

```
>> passenger = Passenger.create(:name => 'Brian Eng', :address =>
'1060 West Addison', :seat_preference => 'Aisle')
  Passenger Create (0.000887)   INSERT INTO passengers ("name",
  "updated_at", "seat_preference", "address", "created_at")
  VALUES('Brian Eng', '2008-01-15 16:13:18', 'Aisle', '1060 West
  Addison', '2008-01-15 16:13:18')
```

```
=> #<Passenger id: 3, name: "Brian Eng", address: "1060 West
Addison", seat_preference: "Aisle", created_at: "2008-01-15
16:13:18", updated_at: "2008-01-15 16:13:18">
```

One thing to note is that when a record gets created, the created_at and updated_at columns of the table automatically get filled in. Don't remember creating those columns? That's because you didn't. When we used the generator to create the migration for the passengers table, it inserts the t.timestamps line in there by default. That's a special method that creates the created_at and updated_at columns for us. These are special column names that ActiveRecord recognizes and automatically fills in for us when a row is created or updated, respectively.

Also note that the value returned from the create method is a instance of the Passenger class, which contains the methods id, name, address, seat_preference, created_at, and updated_at.

```
>> passenger.address
=> "1060 West Addison"
```

You might think that these methods were added in by the generator code as well, but if you take a peek at the Passenger class, you'll see that's not the case:

Download crud/passenger.rb

```
class Passenger < ActiveRecord::Base
end
```

That's right—a completely empty class definition. ActiveRecord knows what columns you have in your database and automatically generates a method for each column under the covers. All you do is call it.

Now, let's explore a few more ActiveRecord methods from the console and see where it takes us. First we'll find and update a single record in the table using the find and save methods:

```
>> passenger = Passenger.find(1)
  Passenger Load (0.000601)   SELECT * FROM passengers WHERE
  (passengers."id" = 1)
=> #<Passenger id: 1, name: "John Doe", address: "123 Main St",
seat_preference: "Aisle", created_at: "2008-01-15 15:49:24",
updated_at: "2008-01-15 15:49:24">
>> passenger.address = '321 Main St'
=> "321 Main St"
>> passenger.save
  Passenger Update (0.000567)   UPDATE passengers SET "created_at"
  = '2008-01-15 15:49:24', "name" = 'John Doe', "seat_preference"
  = 'Aisle', "address" = '321 Main St', "updated_at" = '2008-01-15
```

We can also delete (destroy) a record with the destroy method:

```
>> Passenger.destroy(3)
  Passenger Load (0.000535)   SELECT * FROM passengers WHERE
    (passengers."id" = 3)
  Passenger Destroy (0.000573)    DELETE FROM passengers
 WHERE "id" = 3
=> #<Passenger id: 3, name: "Brian Eng", address: "1060 West
Addison", seat_preference: "Aisle", created_at: "2008-01-15
16:10:34", updated_at: "2008-01-15 16:10:34">
```

And finally, to get all the records in the table, we'll use the find method, passing in the :all option:

```
>> Passenger.find(:all)
  Passenger Load (0.000760)   SELECT * FROM passengers
=> [#<Passenger id: 1, name: "John Doe", address: "123 Main St",
seat_preference: "Aisle", created_at: "2008-01-15 15:49:24",
updated_at: "2008-01-15 15:49:24">, #<Passenger id: 2, name: "Jane
Doe", address: "123 Main St", seat_preference: "Window",
created_at: "2008-01-15 15:55:35", updated_at: "2008-01-15
16:06:10">, #<Passenger id: 3, name: "Brian Eng", address: "1060
West Addison", seat_preference: "Aisle", created_at: "2008-01-15
16:10:34", updated_at: "2008-01-15 16:10:34">]
```

Finding a data collection like this returns an Array of Passenger objects, which we will ultimately loop through to create our grid of data.

Creating the Controller and View for Our Grid of Data

Now that we've added a couple of rows to our passengers table and we know how to get the data we want in order to display our table of passengers, let's hook up the controller and view code.

First, in the controller, we're going to add a method for the index action:

Download crud/passengers_controller.rb

```
class PassengersController < ApplicationController

  def index
    @passengers = Passenger.find(:all)
  end

end
```

In the index action, we've defined an *instance variable* called @passengers that holds an Array of Passenger objects corresponding to all the records of the passengers table. Any instance variables we define in the controller are available for use in the rendered view, which by convention is the view file with the same name as the action located in the subdirectory

In this case, it's app/views/passengers/index.html.erb.

Download crud/index.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <title>Showing All Passengers</title>
</head>
<body>
  <table border="1" cellspacing="5" cellpadding="5">
    <tr>
      <th>Passenger Name</th>
      <th>Address</th>
      <th>Seat Preference</th>
    </tr>

    <% @passengers.each do |passenger| %>
    <tr>
      <td><%= passenger.name %></td>
      <td><%= passenger.address %></td>
      <td><%= passenger.seat_preference %></td>
    </tr>
    <% end %>

  </table>
</body>
</html>
```

Fire up your server using the script/server command, hop over to your browser, and you should see something like what's in Figure 6.1, on the next page.

Rails is definitely a bit "closer to metal" than ASP.NET WebForms. Instead of writing a SQL statement, dropping a control on a page, and letting the framework write the HTML for us, we're writing the HTML ourselves instead, delegating only the most granular data-driven details to the framework. This gives us the ultimate fine-grained control over the final output from the very beginning.

As shown by this example, Rails—unlike ASP.NET—doesn't really have the concept of GUI *controls* that you'd use to build a web form. Your only GUI "toolbox" in Rails is that of the native languages of the Web—HTML, JavaScript, and CSS.

A view, like this one, is simply HTML with some Ruby intermingled

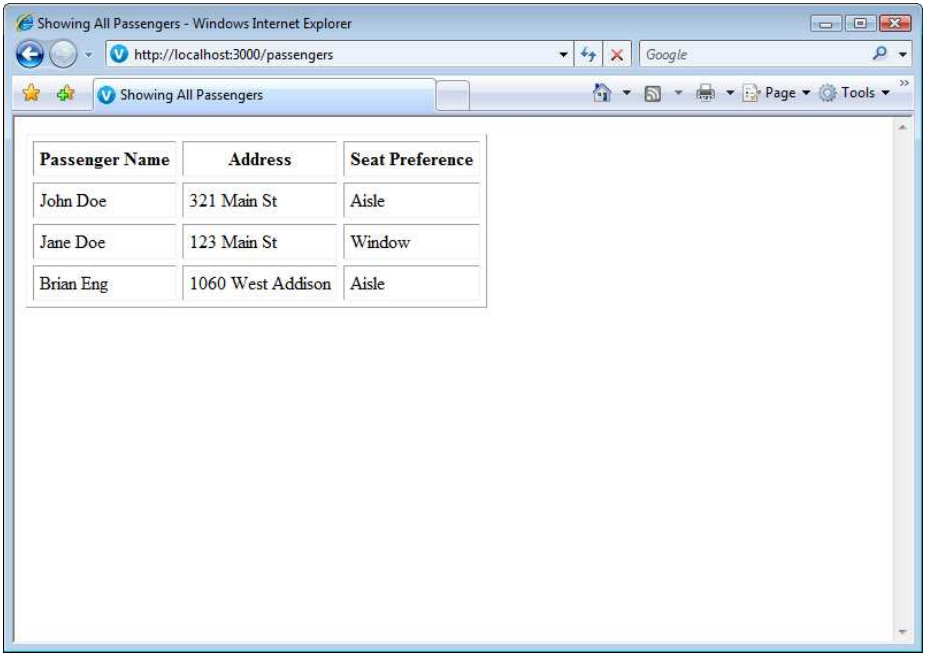


Figure 6.1: Showing a table of all passengers

a basic HTML page with markup for the table of passengers we want to display. Any Ruby code between the <% %> symbols is going to be interpreted at runtime. So, we're dynamically looping through the contents of the @passengers Array and creating a table row for each record. Within each tr tag, we have three columns represented by the td tags, and within each td tag, we're again calling on Ruby to give back a value. When Ruby code lives within <%= %> tags, we're asking Ruby to actually write the result of the code within the tags out to the resulting HTML, instead of simply running the code. From an HTML generation perspective, Rails is similar in style and spirit to traditional ASP.

Sorting, Filtering, and Paging Data

Now that we can view our data using a simple HTML table, it's time to move on creating more interesting views of that data. For a very small data set, the previous example would work just fine. Most likely, however, we will be working with larger and more complex sets of data

in our production environments. For a better end-user experience, we should provide the ability to sort, filter, and paginate.

How You Might Approach It in .NET

If we were using ASP.NET's GridView control, the ability to page through and sort data is built right in. Set the EnablePaging and EnableSorting properties to true, and that's all there is to it.

Let's say we'd also like to filter our grid of passengers by their seat preference. We would start by creating a DropDownList control that contains the values Aisle, Window, and Both. We would bind the SelectedIndexChanged event of the DropDownList with the following code:

Download crud/Passengers.aspx.cs

```
protected void DropDownList1_SelectedIndexChanged(object sender,
                                                  EventArgs e)
{
    string filter = (sender as DropDownList).SelectedItem.Value;
    if (filter == "Both")
    {
        SqlDataSource1.SelectCommand = "SELECT * FROM passengers";
    }
    else
    {
        SqlDataSource1.SelectCommand = "SELECT * FROM passengers WHERE
        seating_preference = '" + filter + "'";
    }
    GridView1.DataBind();
}
```

This well illustrates the advantage of a GUI controls-based approach, because all the hard work of coding HTML, styling it, and managing the appropriate JavaScript callback hooks have all been done for us; all we need to do is turn it on. The flip side of that is that we're stuck with the implementation the framework provides, at least without quite a bit of kicking and screaming. Customizing the way the built-in ASP.NET controls work is a much more difficult task than with handcrafted HTML and CSS.

The Rails Way

Let's take our passenger list from the previous example and give it the same functionality that the ASP.NET GridView version would give us.

Sorting

One of the really nice things about the ASP.NET GridView control is that it makes sorting simple. We can click a column header, and it automatically sorts the content by the values in that column. It is also smart enough to hang onto the current sort order so that if the column header is clicked again, the order is reversed.

Let's take a look at how ActiveRecord will help us do the same thing. Let's fire up script\console again:

```
>> Passenger.find(:all, :order => 'name')
  Passenger Load (0.001694)   SELECT * FROM passengers ORDER BY name
=> [#<Passenger id: 4, name: "Brian Eng", address: "1060 West
Addison", seat_preference: "Aisle", created_at: "2008-01-15
16:13:18", updated_at: "2008-01-15 16:13:18">, #<Passenger id: 2,
name: "Jane Doe", address: "123 Main St", seat_preference: "Window",
created_at: "2008-01-15 15:55:35", updated_at: "2008-01-15
16:06:10">, #<Passenger id: 1, name: "John Doe", address: "321 Main
St", seat_preference: "Aisle", created_at: "2008-01-15 15:49:24",
updated_at: "2008-01-15 16:18:31">]
```

The second argument of the find method is an optional hash of options. Among others, one of those options is order, which we've passed in here. This option tacks on an ORDER BY clause to the end of the resulting SQL statement, using the value of the order option as the second half of the ORDER BY clause:

```
>> Passenger.find(:all, :order => 'name DESC')
  Passenger Load (0.001242) SELECT * FROM passengers ORDER BY name DESC
=> [#<Passenger id: 1, name: "John Doe", address: "321 Main St",
seat_preference: "Aisle", created_at: "2008-01-15 15:49:24",
updated_at: "2008-01-15 16:18:31">, #<Passenger id: 2, name: "Jane
Doe", address: "123 Main St", seat_preference: "Window", created_at:
"2008-01-15 15:55:35", updated_at: "2008-01-15 16:06:10">,
#<Passenger id: 4, name: "Brian Eng", address: "1060 West Addison",
seat_preference: "Aisle", created_at: "2008-01-15 16:13:18",
updated_at: "2008-01-15 16:13:18">]
```

Since the order option of the find method will add anything we want to the ORDER BY clause of the resulting SQL statement, we've simply added the SQL to *order descending*.

Adding to the Controller and View to Support Sorting

Assuming we want the same behavior as the ASP.NET GridView, where clicking a column header a single time sorts by that column and clicking it again reverses the order, let's first change those static column headers into links.

Download crud-sorting/index.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <title>Showing All Passengers</title>
</head>
<body>
  <table border="1" cellspacing="5" cellpadding="5">
    <tr>
      <th>
        <%= link_to "Passenger Name", passengers_url(:order => "name",
        :reverse => session[:order] == "name") %>
      </th>
      <th>
        <%= link_to "Address", passengers_url(:order => "address",
        :reverse => session[:order] == "address") %>
      </th>
      <th>
        <%= link_to "Seat Preference", passengers_url(:order =>
        "seat_preference", :reverse => session[:order] ==
                                              "seat_preference") %>
      </th>
    </tr>

    <% @passengers.each do |passenger| %>
    <tr>
      <td><%= passenger.name %></td>
      <td><%= passenger.address %></td>
      <td><%= passenger.seat_preference %></td>
    </tr>
    <% end %>

  </table>
</body>
</html>
```

The link_to helper method is the built-in way to spit out a hyperlink. As with all the other helper methods, you don't have to use it—you could write out the HTML a tag instead—but using the helper is a lot easier and makes for much simpler refactoring in the long run.

Let's take the first link_to as an example. We're telling the link_to helper that the resulting link should have "Passenger Name" as its text and that the URL it should link to is passengers_url. passengers_url is another helper method that's generated by Rails' routing mechanism, which we will examine in greater depth in Chapter 7, *Directing Traffic with*

the string http://localhost:3000/passengers, which will point back to our index action.

In addition, we're passing along a parameter called order that will let the controller know the column by which we'd like to sort. This adds a GET parameter to the resulting URL and changes it from http://localhost:3000/passengers to http://localhost:3000/passengers?order=name. On the controller end of things, we can access this value (and any other GET or POST parameters) using a special hash named params. Think of params like Request.Form or Request.QueryString in ASP.NET—it's a container meant for us to pass values from the presentation layer through to the business layer, most notably HTTP GET or POST parameters. So, to get the value of the order GET parameter, we ask the params hash, using the name of the parameter we want (as a symbol) as the key. With that knowledge in hand, here's how we might change our PassengersController to account for the sort order:

Download crud-sorting/passengers_controller_simple.rb

```
class PassengersController < ApplicationController

  def index
    @passengers = Passenger.find(:all, :order => params[:order])
  end

end
```

Not bad. But not perfect. We still need to build in the reverse sorting when the header column is clicked a second time. To do this, we'll have to store which column was clicked, and for this, we'll use the session object. Just like in ASP.NET, the session object refers to the application's session store, which may be configured to be held in a variety of ways, including a cookie, a file, or the database. It's simply a way for objects to be held for later use in the application.

Download crud-sorting/passengers_controller.rb

```
class PassengersController < ApplicationController

  def index
    options = {}
    if params[:order]
      new_order = params[:order]
      new_order += " DESC" if params[:reverse]
      options.merge!(:order => new_order)
      session[:order] = new_order
    end
    @passengers = Passenger.find(:all, options)
  end
```

In our finished version of the PassengersController, we check for the existence of an order parameter and merge it into our options hash if it exists. In addition, we store the current sort order into the session, and in the view, we check whether the new sort order is the same as the current sort order—meaning the user clicked the column header a second time. If it is, we make the sort order descending by passing along an additional reverse parameter.

Filtering

In our ASP.NET example, we used a DropDownList control to filter our passenger list by seat preference. To do the same thing in Rails, we'll need to use a combination of a simple form with a select_tag and some additional code in the controller to do the actual filtering. Our approach is not unlike how it's implemented in ASP.NET; but yes, we have to think of it in a more raw HTML way, as opposed to letting a built-in control do the thinking for us.

We'll build a form that will submit to the index action (that is, the /passengers URL). Inside the form, we'll have a select_tag that will hold the values Both, Aisle, and Window. And we'll also have a hidden field that will carry over the value of any sort order we may already have. We'll also throw some JavaScript in there to automatically submit the form when the value of the select is changed.

Download crud-filtering/index.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <title>Showing All Passengers</title>
</head>
<body>
  <table border="1" cellspacing="5" cellpadding="5">
    <tr>
      <th><%= link_to "Passenger Name",
                passengers_url(:order => "name",
                :reverse => session[:order] == "name") %>
      </th>
      <th><%= link_to "Address",
                passengers_url(:order => "address",
                :reverse => session[:order] == "address") %>
      </th>
```

```
      <th><%= link_to "Seat Preference",
                passengers_url(:order => "seat_preference",
                :reverse => session[:order] == "seat_preference") %>
      </th>
    </tr>

    <% @passengers.each do |passenger| %>
    <tr>
      <td><%= passenger.name %></td>
      <td><%= passenger.address %></td>
      <td><%= passenger.seat_preference %></td>
    </tr>
    <% end %>

  </table>

  <p>
    <% form_tag passengers_url, :method => :get do %>
      <%= hidden_field_tag "order", params[:order] if params[:order] %>
      <%= select_tag "filter", options_for_select(%w(Both Aisle Window),
      params[:filter]), :onchange => "this.form.submit()" %>
    <% end %>
  </p>
</body>
</html>
```

Remember that, as with all Rails helpers, the form_tag and select_tag helpers ultimately generate raw HTML markup. Here, we're telling the form_tag method that we'd like the resulting HTML form to submit to the /passengers URL with an HTTP method of GET. We're also including a select_tag, which will generate an HTML select tag with our list of choices as options, as well as a hidden_field_tag that will help us carry over the current sort order if there is one. If we were to "view source" on the resulting page, we'd see some simple and clean HTML:

```
<form action="http://localhost:3000/passengers" method="get">
  <select id="filter" name="filter" onchange="this.form.submit()">
    <option value="Both">Both</option>
    <option value="Aisle">Aisle</option>
    <option value="Window">Window</option></select>
</form>
```

There are a couple of things to note here. We must explicitly specify a method of GET on the form, because the POST version of /passengers points to the create action. Also, if we wanted to make sure that those without JavaScript turned on had a working experience, we could add a submit button to the form, tucking it inside a noscript tag.

Adding to the Controller and View to Support Filtering

Now that our view code is in there, we can add a couple of lines of code to the controller to support the filtering we want:

Download crud-filtering/passengers_controller.rb

```
class PassengersController < ApplicationController

  def index
    options = {}

    if params[:order]
      new_order = params[:order]
      new_order += " DESC" if params[:reverse]
      options.merge!(:order => new_order)
      session[:order] = new_order
    end

    if params[:filter] && params[:filter] != "Both"
      options.merge!(:conditions => ["seat_preference = ?",
                                                    params[:filter]])
    end

    @passengers = Passenger.find(:all, options)
  end

end
```

Again, we're testing for the existence of a parameter called filter in our params hash, and we're merging a conditions option into our options hash if it exists. The conditions option allows for the addition of a WHERE clause to the resulting SQL statement. Let's look at a quick example:

```
>> Passenger.find(:all, :order => "name", :conditions =>
"seat_preference = 'Aisle'")
  Passenger Load (0.001123)   SELECT * FROM passengers WHERE
  (seat_preference = 'Aisle') ORDER BY name
=> [#<Passenger id: 4, name: "Brian Eng", address: "1060 West
Addison", seat_preference: "Aisle", created_at: "2008-01-15
16:13:18", updated_at: "2008-01-15 16:13:18">, #<Passenger id: 1,
name: "John Doe", address: "321 Main St", seat_preference: "Aisle",
created_at: "2008-01-15 15:49:24", updated_at: "2008-01-15
16:18:31">]
```

Just like the order option, the conditions option allows us to insert any arbitrary SQL into the resulting SQL statement. Whereas order corresponds with the ORDER BY clause, conditions instead maps to the WHERE clause. And, in addition to a simple string, we can use other

Special Syntax for the conditions Option

We can specify a string, an array, or a hash in the conditions option of a find. If we use a string, that string will simply be used as the WHERE clause of the resulting SQL statement.

If we choose a hash, the WHERE clause will be constructed from the keys and values of the hash. For example:

```
>> Passenger.find(:all, :conditions => {
:seat_preference => 'Aisle' })
  Passenger Load (0.000582)   SELECT * FROM passengers WHERE
   (passengers."seat_preference" = 'Aisle')
```

If we use an array, the first element of the array represents the body of the WHERE clause. Any question marks (?) in the first element will be substituted with the values from the subsequent members of the array.

In general, when building a web application where the conditions are typically dynamically set through interactions with the end user, we'll want to use either the array or hash style of building conditions. If we do this instead of specifying a plain string, Rails will automatically sanitize the input parameters, protecting against SQL injection attacks. In addition, strings will be properly quoted, so we don't have to worry about it.

special syntax to construct the WHERE clause (see the sidebar on the current page).

Paging

Rails no longer has built-in support for pagination as part of the core framework, because the Rails core team has adopted a philosophy of keeping only the most common and critical features in the core framework and leaving everything else to *plug-ins*. Plug-ins are smaller modules that add functionality to Rails when we need it, and one of the most popular is the will_paginate plug-in by Chris Wanstrath and PJ Hyett. This plug-in is distributed as a Ruby gem. We will learn more about how Rails applications can use third-party gems in Section 12.2, *Using Gems in Your Rails Applications*, on page 238.

For now, simply open up the config/environment.rb file, and add the following lines:

Download crud-paging/environment.rb

```
Rails::Initializer.run do |config|

  config.gem  'mislav-will_paginate',
              :version => '~> 2.3.2',
              :lib => 'will_paginate',
              :source => 'http://gems.github.com'

end
```

Now that we've specified that our application needs this gem, Rails can install it for us:

```
c:\dev\flight> rake gems:install
```

We'll have to restart our web server for Rails to pick up our changes. Using will_paginate is straightforward. It gives us a new method, paginate, that is a drop-in replacement for the find method on any ActiveRecord model class. The only difference between paginate and find is that paginate needs two more parameters to be set: per_page, which is the number of records to display on one page, and page, the current page number. Our revised controller code now looks like this:

Download crud-paging/passengers_controller.rb

```
class PassengersController < ApplicationController

  def index
    options = {}

    if params[:order]
      new_order = params[:order]
      new_order += " DESC" if params[:reverse]
      options.merge!(:order => new_order)
      session[:order] = new_order
    end

    if params[:filter] && params[:filter] != "Both"
      options.merge!(:conditions => ["seat_preference = ?",
                                     params[:filter]])
    end

    options.merge!(:page => params[:page], :per_page => 2)
    @passengers = Passenger.paginate(options)
  end

end
```

We've replaced the find method with the paginate method and added the per_page and page values to the default options hash. per_page is set to a hard-coded value of 2, and page is set using the will_paginate view helper, which we'll add at the bottom of our view:

Download crud-paging/index.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <title>Showing All Passengers</title>
</head>
<body>
  <table border="1" cellspacing="5" cellpadding="5">
    <tr>
      <th>
        <%= link_to "Passenger Name", passengers_url(:order => "name",
        :reverse => session[:order] == "name") %>
      </th>
      <th>
        <%= link_to "Address", passengers_url(:order => "address",
        :reverse => session[:order] == "address") %>
      </th>
      <th>
        <%= link_to "Seat Preference", passengers_url(:order =>
        "seat_preference", :reverse => session[:order] ==
                                              "seat_preference") %>
      </th>
    </tr>

    <% @passengers.each do |passenger| %>
    <tr>
      <td><%= passenger.name %></td>
      <td><%= passenger.address %></td>
      <td><%= passenger.seat_preference %></td>
    </tr>
    <% end %>

  </table>

  <p>
    <% form_tag passengers_url, :method => :get do %>
      <%= hidden_field_tag "order", params[:order] if params[:order] %>
      <%= select_tag "filter", options_for_select(%w(Both Aisle Window),
      params[:filter]), :onchange => "this.form.submit()" %>
    <% end %>
```

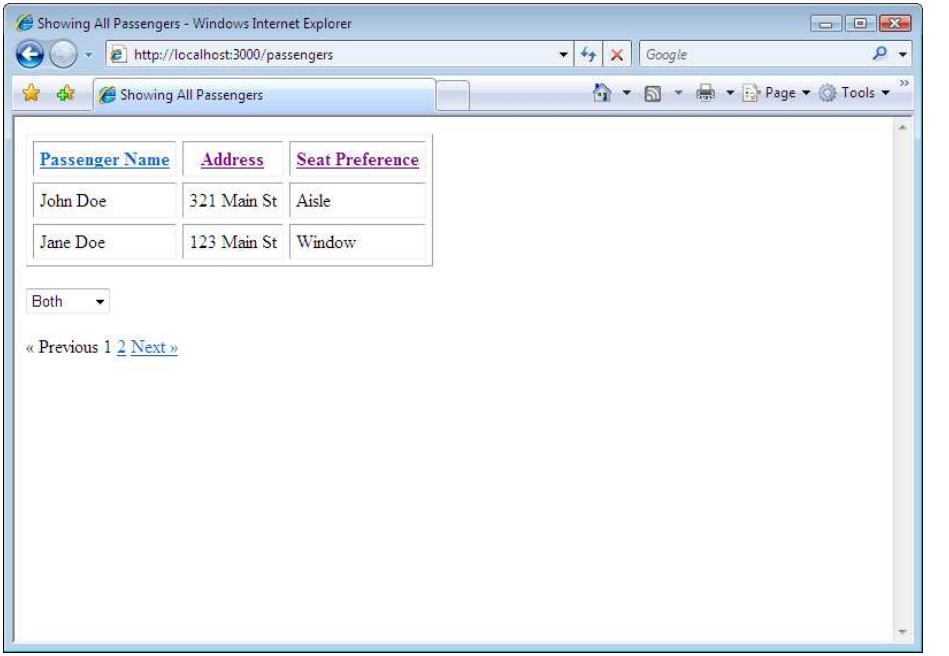


Figure 6.2: Passenger data with sorting, filtering, and pagination

```
  <p>
    <%= will_paginate @passengers %>
  </p>
</body>
</html>
```

We've added the will_paginate helper method to our view, which translates into some markup that gives us *previous* and *next* links, as well as a numbered link for each page of data in our data set. Following any of these links will send us back to the current action—the index action—with the addition of the aforementioned page parameter. Our final output is shown in Figure 6.2.

Validating User Input

In a perfect world, our end users would never make mistakes. They would fill out all the required fields on forms, never violate referential

not, happen all the time when users are using our web applications. We must have a way of validating user input and return clear and informative messages to the end user when it happens.

Although conventional wisdom says to enforce data rules in the database itself, for the purposes of this discussion, we're not going to talk about that. Rather, we're going talk about the differences in validating data on the client side between the two platforms.

How You Might Approach It in .NET

For a simple ASP.NET form, we can use the different validator controls that it provides to validate form input on the client side. For instance, to ensure that the user doesn't leave a textbox empty, we can use the RequiredFieldValidator and attach it to the form field that's required. Here's a basic example:

Download crud-validations/AddPassenger.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="AddPassenger.aspx.cs"
    Inherits="AddPassenger" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Add a Passenger</title>
</head>
<body>
    <form id="form1" runat="server">
        <p>
        Name
        <asp:TextBox ID="name" runat="server"></asp:TextBox>
        <asp:RequiredFieldValidator ID="RequiredFieldValidator1"
            runat="server"
            ErrorMessage="Name is required."
            ControlToValidate="name">
            </asp:RequiredFieldValidator>
        </p>
        <p>
        Address
        <asp:TextBox ID="address" runat="server"></asp:TextBox>
        <asp:RequiredFieldValidator ID="RequiredFieldValidator2"
            runat="server"
            ErrorMessage="Address is required."
            ControlToValidate="address">
```

```
        <p>
        Seating Preference
        <asp:TextBox ID="seating_preference" runat="server">
        </asp:TextBox>
        </p>
        <p>
        <asp:Button ID="Button1" runat="server" Text="Add Passenger"
                onclick="Button1_Click1" />
        </p>
    </form>
</body>
</html>
```

Next to each of the textboxes that are required to be filled out—the ones for the name and address—we put a RequiredFieldValidator that is attached to the respective textbox. When we try to submit the form, error messages will appear if the textboxes are empty.

The Rails Way

Here is a simple example of a form to create a new passenger:

Download crud-validations/new.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <title>Showing All Passengers</title>
</head>
<body>
  <% form_for @passenger do |f| %>
    <p>
      <%= f.label :name %>
      <%= f.text_field :name %>
    </p>
    <p>
      <%= f.label :address %>
      <%= f.text_field :address %>
    </p>
    <p><%= f.submit "Create new passenger" %></p>
  <% end %>
</body>
</html>
```

Don't worry too much about how this form works for now; we'll be going in depth with Rails forms in Chapter 8, *Exploring Forms, Layouts, and Partials*, on page 150. For now, just know that this code will result in

two HTML text input fields being shown onscreen, and when this form is submitted, it will call the create action.

In Rails, we use the same declarative-style syntax to perform validations as we've seen before. A philosophical difference between Rails and ASP.NET surfaces exists here; instead of this validation being enforced in the presentation layer as with ASP.NET, Rails puts this type of logic down into the model. To implement the same functionality as we've done in our ASP.NET add-a-passenger example, we can do something like this:

Download crud-validations/passenger.rb

```
class Passenger < ActiveRecord::Base
  validates_presence_of :name, :address

end
```

The validates_presence_of method on the Passenger model class won't allow a passenger record to be saved to the database if it's blank or nil—that is, if we call save on a Passenger object without a name or address, the record won't be saved, and the return value of the save method will be false.

Next, we should probably show a message to the end user in the event they do leave either of those fields blank. Let's see what our PassengersController looks like without the logic to show an error message in place:

Download crud-validations/passengers_controller.rb

```
class PassengersController < ApplicationController

  def new
    @passenger = Passenger.new
  end

  def create
    @passenger = Passenger.new(params[:passenger])
    @passenger.save
    redirect_to passengers_url
  end

end
```

In the create action, we're simply instantiating a new Passenger object with the values passed in from the new.html.erb form and calling its save method.

Regardless of what happens, we'll redirect to the passenger index page. Now, let's have a look at the code with error handling enabled:

Download crud-validations/passengers_controller_with_validations.rb

```
class PassengersController < ApplicationController

  def new
    @passenger = Passenger.new
  end

  def create
    @passenger = Passenger.new(params[:passenger])
    if @passenger.save
      redirect_to passengers_url
    else
      render :action => 'new'
    end
  end

end
```

We've added an `if` statement that redirects the index page if the `save` call is successful and simply re-renders the `new.html.erb` form if not. In our form, we'll also add a line of code that will display our error message:

Download crud-validations/new_with_validations.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
  <title>Showing All Passengers</title>
</head>
<body>
  <%= error_messages_for :passenger %>
  <% form_for @passenger do |f| %>
    <p>
      <%= f.label :name %>
      <%= f.text_field :name %>
    </p>
    <p>
      <%= f.label :address %>
      <%= f.text_field :address %>
    </p>
    <p><%= f.submit "Create new passenger" %></p>
  <% end %>
</body>
</html>
```

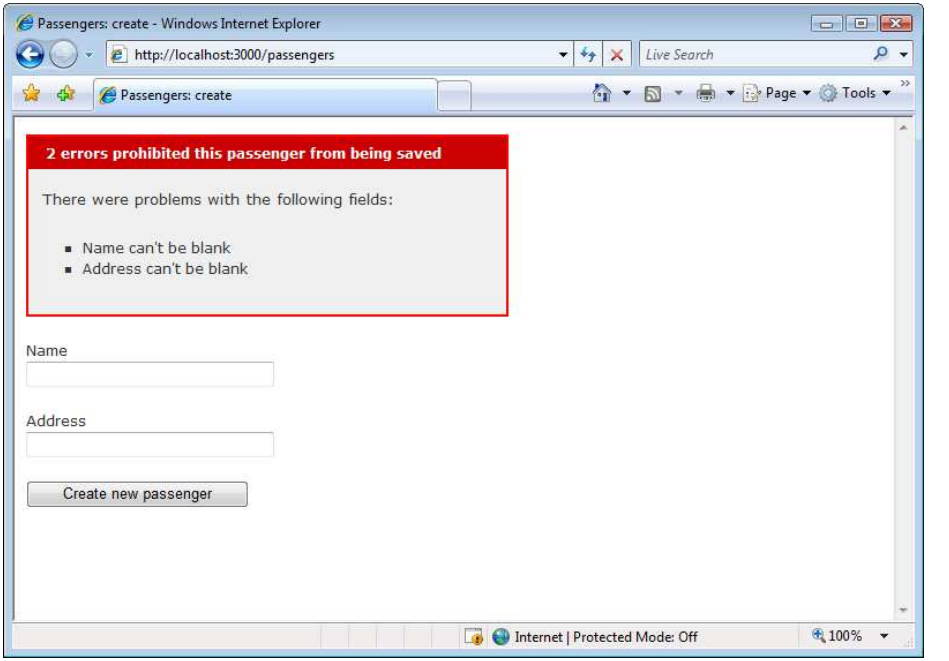


Figure 6.3: Standard Rails validation message

This additional line of code with result in a standard error message displayed inline, as shown in Figure 6.3.

Representing Relationships Between Tables

Rarely does a web application allow a user to view and manipulate a single database table. A much more common situation involves the slicing and dicing of data based on much more complex relationships between tables. For example, in our flight application, we're probably not that interested in seeing a list of all passengers; rather, it's much more interesting to see which passengers are on a particular flight.

Part of this exercise is the actual modeling of the database. Can a passenger be on more than one flight? Can a flight have more than one passenger? The relationship we're discussing is most certainly a many-to-many one. We've outlined this relationship in Figure 6.4, on the following page.

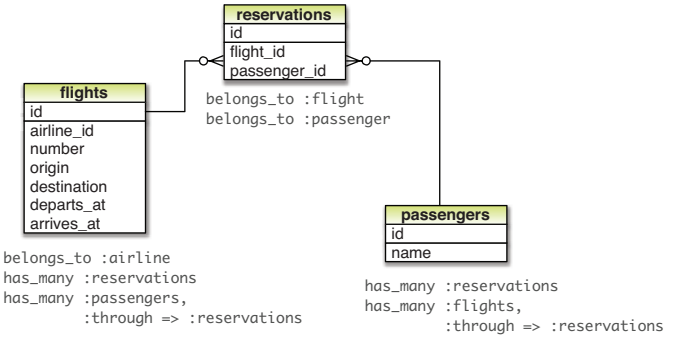


Figure 6.4: Relationship between flights and passengers

How You Might Approach It in .NET

To do this type of thing in out-of-the-box .NET, we're almost certainly talking about LINQ, or a SQL statement with an INNER JOIN, possibly saved away in a stored procedure for reusability. If we were interested in which passengers were on flight #123, for instance, we could hand-code this SQL:

```
SELECT passengers.* FROM passengers INNER JOIN reservations
  ON passengers.id = reservations.passenger_id INNER JOIN flights
  ON passengers.flight_id = flights.id
  WHERE flights.number = '123'
```

This is pretty straightforward for the experienced developer but not quite as object-oriented or elegant as the Rails solution.

The Rails Way

In Rails, we can set up the relationships between our tables in our model code, in a declarative fashion:

Download crud-associations/models.rb

```
class Flight < ActiveRecord::Base
  has_many :reservations
  has_many :passengers, :through => :reservations
end

class Reservation < ActiveRecord::Base
  belongs_to :flight
  belongs_to :passenger
end
```

```
class Passenger < ActiveRecord::Base
  has_many :reservations
  has_many :flights, :through => :reservations
end
```

We can see at a glance that a flight has_many reservations and that a passenger also has_many reservations. In addition, the has_many :through associations allow us to chain these relationships together. For example, the reservation class is used as a proxy between a flight and a passenger; a flight has_many passengers *through* the reservations association. This code reflects our database diagram directly and is extremely easy to read and understand.

We've said it before; but again, it's important to step back at this point and realize that ActiveRecord is not magic. What have we actually done with these few lines of code? Rails simply takes a look at the names of the classes we are referring to and adds a number of additional methods to each of our classes so we can do things like passenger.flights and passenger.flights.find(123) without having to write our own methods. A full list of all the methods added by Rails' associations is available via the official Rails documentation.[1]

Back to the task at hand—how do we get our passengers for flight #123, then?

```
c:\dev\flight> ruby script\console
Loading development environment (Rails 2.1.0)
>> f = Flight.find_by_number '123'
  Flight Load (0.000554) SELECT * FROM flights WHERE
  (flights."id" = 1)
=> #<Flight id: 1, flight_number: 123, departs_at:
  "2008-06-25 12:00:00", arrives_at: "2008-06-25 16:00:00", origin:
  "ORD", destination: "PDX", created_at: "2008-06-25 12:44:43",
  updated_at: "2008-06-25 12:44:43">
flight> f.passengers
  Flight Load (0.000973) SELECT * FROM flights WHERE
  (flights."id" = 1)
  Passenger Load (0.000982)   SELECT passengers.* FROM passengers
    INNER JOIN reservations ON passengers.id =
    reservations.passenger_id WHERE ((reservations.flight_id = 1))
=> [#<Passenger id: 1, name: "Brian Eng",
  address: "1060 West Addison", created_at: "2008-06-30 14:42:53",
  updated_at: "2008-06-30 14:42:53">, #<Passenger id: 2, name:
  "Jeff Cohen", address: "1901 West Madison", created_at:
  "2008-06-30 14:44:14", updated_at: "2008-06-30 14:44:14">]
```

1. http://api.rubyonrails.org/classes/ActiveRecord/Associations/ClassMethods.html

Because we have our associations set up properly, getting the passengers for a flight is pretty straightforward and intuitive. We've found the flight record with the flight number 123, and we've used the method passengers that comes along with the has_many association to return the list of passengers in which we're interested. And, as you can see from the output, Rails uses essentially the same SQL statement we hand-wrote in our .NET example. To reiterate, ActiveRecord is just SQL, but with just a thin layer of abstraction on top to make this kind of stuff a lot more pleasant.

In this chapter, we learned how to do basic CRUD operations with ActiveRecord—find, create, update, and destroy, and we saw that ActiveRecord generates plain ol' SQL that adjusts for the little differences between RDBMSs. We've also built functionality onto our flight reservations app that emulates the basic features of the ASP.NET GridView. We took a brief look at how Rails handles the job of data validation vs. the .NET approach we're used to using. And lastly, we examined the .NET vs. Rails way of dealing with relationships between database tables.

We've also had a little taste of what's to come, exploring the basics of the controller by doing some passing of data between HTTP requests using the params hash and learning how to keep data around for later using the session store. We've touched on some view basics as well by looking at some rudimentary ERb and the link_to, form_tag, and select_tag view helpers. The remainder of this section of the book will deal further with the controller and view; first, the controller.

Chapter 7

Directing Traffic with ActionController

In the previous chapter, we looked at some of the capabilities of ActiveRecord while creating our passenger list and, at the same time, we scratched the surface of controllers and views. In this chapter, we'll explore controllers in much greater detail.

ActionController is the part of Rails that we'll be discussing in depth in this chapter. ActionController is one of two modules that make up ActionPack, the other being ActionView. ActionController gives us (you guessed it) controllers in Rails, the part of the framework that's responsible for processing an incoming request and figuring out what to do with it. After the controller processes the request and we're now ready to return a response to the end user, ActionView (which we'll be discussing in the next chapter) goes to work generating HTML, XML, JavaScript, or anything else that the consuming application (for instance, a web browser) would be interested in.

We're going to touch on some of the main functionality of ActionController by going through three practical examples: using Rails routing to rewrite URLs, performing user authentication, and providing an API for our application.

Routing and Pretty URLs

At the heart of every web development framework lies some mechanism that takes a URL that a user types into the web browser and hands it

translates to our application's inner workings, giving us the freedom to be as creative (or not) as we want with our URLs.

Let's look at a practical example. We'd like each airline in our flight reservations system to have its own page, where we'll show the details for that airline along with any flights it may have for that day. Since we are using resource-based routing (that is, we have the line map.resources :airlines in our routes.rb file), our conventional resource-based URL for such a page would look something like /airlines/1, where airlines represents the name of our resource, airline, followed by the numeric ID of the resource we want. And by convention, this URL will route to the show action of the AirlinesController.

But let's say we don't want that. Suppose we'd like the URL to read /flights-for-today/united instead. There are a variety of reasons why we might want our URL to read like this, such as *search engine optimization* (SEO), backward compatibility, or simply improved readability.

How You Might Approach It in .NET

Routing in ASP.NET is largely file-based. That is, the default URLs in an application largely depend on where the .aspx files live in the file system, and where you put those files is largely up to the developer. For instance, if we have a file called ShowAirline.aspx located in a directory called Airlines, that would translate to a URL of /Airlines/ShowAirline.aspx.

There are variety of ways to do the type of URL "rewriting" we want with ASP.NET.[1] We'll look at just one of these techniques in particular.

The first technique involves only application code and no web server–level configuration. We can simply intercept the request by putting code in the Application_BeginRequest event handler in Global.asax, and using the HttpContext.RewritePath() method, we can redirect the user-inputted URL to the URL that really works. Here's a simplified example of this approach:

Download controller/Global.asax

```
void Application_BeginRequest(object sender, EventArgs e) {

    string requestedUrl = Request.Url.ToString();

    if (requestedUrl.Contains("/flights-for-today/United.aspx")) {
      Context.RewritePath("/Airlines/ShowAirline.aspx?Airline=United");
    }
}
```

In this not-so-robust example, we're simply looking for the existence of a URL that reads /flights-for-today/United.aspx and redirecting it to a URL that actually does something in our application, that is, /Airlines/ShowAirline.aspx?Airline=United. This example works only for a single URL; if we wanted to implement this application-wide, UrlRewriter[2] is a good, open source HttpModule that takes this concept and makes it a lot easier and configurable through our application's web.config.

There are also a few techniques we can employ to remove the .aspx extension entirely. In IIS 6, for example, we can ask IIS to send extension-less requests all the way to ASP.NET, and UrlRewriter can take it from there. With IIS 7, it is a bit easier, because HttpModules can be executed anywhere within the IIS pipeline. In any event, there is certainly some work involved to get it to work.

The Rails Way

ActionController gives us a very simple yet powerful way to deal with everything that has to do with routing—the config/routes.rb file. Adding a single additional line in the routes file defines our custom route:

Download controller/routes.rb

```
ActionController::Routing::Routes.draw do |map|
  map.flights_today '/flights-for-today/:name',
    :controller => 'airlines',
    :action => 'show'

  map.resources :airlines
  map.resources :flights
  map.resources :passengers
end
```

This line of code tells our Rails application to look for any request with the URL /flights-for-today/<name> and to send that request along to the show action of the FlightsController. Furthermore, whatever is in the name part of the URL gets passed into the controller and will be accessible via the params hash. An important thing to note is the order that the routes are specified. Routes are evaluated top down, so whatever comes first wins. If we had inserted our custom route at the bottom of the file, the resource-based routes would have been evaluated first. In either case, the resource-based route is still present, so a URL of /airlines/1 still works.

2. http://urlrewriter.net

Here's how the show action of the airlines controller might look:

```
def show
  @airline = Airline.find_by_name(params[:name]) ||
  Airline.find(params[:id])
end
```

The *or* condition in this line of code means that this controller action supports both the resource-based route and the new custom route we put in place; if the name parameter isn't supplied, we perform a find by ID instead.

So, now we know how to interpret this URL if it's typed in by the end user. How do we access this route in our application? Because we've used the syntax map.flights_today, we've created a *named route*. Named routes refer to any custom routes in our routes file that get defined with a specific name—a generic, non-named route would be defined using map.connect. Named routes give us a couple of helper methods that we're able to use from our controllers and views. In this case, the flights_today_url and flights_today_path methods are created for us. The only difference between the two methods is that one creates the entire fully qualified URL, such as http://localhost:3000/flights-for-today/united, whereas the other generates just the absolute path, that is, /flights-for-today/united. For example, we can create a link to an airline page in a view like this:

```
<%= link_to "United Flights for Today", flights_today_path(
:name => 'united') %>;
```

The output of this code will simply be an HTML a tag: United Flights for Today. What we're expected to pass into the flights_today_path method is a hash of the dynamic parts of the URL as specified by our custom route, in this case, just the name of the airline.

By now, you've probably figured out that the main advantage of using a named route is easy refactoring. If we ever wanted to change the way that URL reads, perhaps to /united/flights-for-today, no changes would have to be made to the individual controllers and views. One change to the routes file, and we're done.

That's just a small taste of what's possible with routing. Much more complex rules are possible; here are some short examples:

```
map.flights_by_date "/flights_for/:year/:month/:day",
  :controller => "flights",
```

```
    :year => /(19|20)\d\d/,
    :month => /[01]?\d/,
    :day => /[0-3]?\d/ }
```

Here, we want to show a list of flights by date, and we want to pass that date in via the URL. It's similar to our airline page example, with the addition of the requirements option, which specifies regular expressions under which the URL will be valid. That is, the application will route to the show action of the flights_by_date controller only if the format requirements of the year, month, and day parts of the URL are satisfied. If not, it will simply fall through to the reminder of the routes until either one is matched or a routing error exception is raised.

We can also use a regular expression for URL parameters:

```
map.connect '/:name', :id => /\A.*-flights\Z/,
:controller => 'airlines', :action => 'show'
```

Here, a URL like http://localhost:3000/united-flights will be matched. And in the controller code, we're receiving the value of the name parameter from the match result of the regular expression as specified by the route, so the controller code we already have in place still works. This can be a particularly useful technique in creating SEO-friendly URLs. This is also an example of a non-named route. By using the method connect, there will be no helper methods that will automatically produce a URL or path to this action when called.

As another example of creative things we can do with routing, we'll pass an array of parameters via the URL:

```
map.connect 'flights_for/*airlines', :controller => 'flights',
:action => 'show'
```

In the FlightsController, we'll be passed a parameter called airlines, which will be an array of values specified in the URL. For example, a URL of http://localhost:3000/flights_for/American/Delta/United will yield an array containing the values ["American", "Delta", "United"] when params[:airlines] is called in our controller. Here's a simple example of how we might use this in our controller to build up a list of all the airlines we asked for in the URL:

```
def show
  @airlines
  params[:airlines].each do |airline_name|
    if airline = Airline.find_by_name(airline_name)
      @airlines << airline
    end
```

@airlines will then become an array of the Airline objects, which we can further manipulate for displaying to the end user.

Finally, a predefined method tells Rails what to do when the root of the application is requested, for example, http://localhost:3000:

```
map.root, :controller => 'passengers'
```

The map.root method is simply an alias for map.connect '/'. In addition, for this route to work properly, we must delete the public/index.html file. Otherwise, the end user will see the "welcome" page when hitting our website's root.

User Authentication

Some kind of framework for user authentication is usually important when building a web application. In its simplest form, a user of the system supplies a login and password, and the application uses that information to compare against stored information in a database or some other data store typically used for security purposes, such as OpenID, LDAP, or NTLM. For the purposes of this exercise, we'll look at how to build a simple login form and authenticate against a database.

How You Might Approach It in .NET

There are a few ways to perform database-backed form-based authentication in ASP.NET. Let's assume we want to protect the entire site against unauthenticated users. A typical scenario might go something like this:

Download controller/web.config

```
<system.web>

  <authentication mode="Forms">
    <forms loginUrl="Login.aspx" />
  </authentication>

  <authorization>
    <deny users="?" />
  </authorization>

</system.web>
```

In the web.config file for the application, we're turning on forms authentication and denying access to all unauthenticated users. We'll also create a login form called Login.aspx, where we'll direct all unauthenticated

Then, we'll create a *User Store* database where information on the users of the application, authentication details, and roles are stored.

Finally, in the Login.aspx form, we add some code to an event handler that fires when login is attempted:

Download controller/Login.aspx.cs

```
if (Membership.ValidateUser(username, password))
{
  FormsAuthentication.RedirectFromLoginPage(username, false);
}
```

We check the validity of the login details provided by using the Membership class's ValidateUser method. If successful, redirect to the page originally requested. If not, the user gets sent back to the login form.

The Rails Way

Performing user authentication with Rails is not dissimilar to the ASP .NET way, in the sense that it involves a three-step process of denying the user access to one or more resources, displaying a login form, and storing the user's credentials away somewhere once authenticated.

The Rails community has created several plug-ins to handle this scenario, one of the most popular ones being restful_authentication.[3] In practice, we would use one of these plug-ins and be done. In this case, however, we're going to build something similar in functionality to restful_authentication to illustrate several key aspects of Rails' controllers.

Setting Up the Model

Before we can get to writing controller code, though, we'll need to do some basic setup on the model side. Let's quickly create a User model and build a method to help us authenticate against it:

```
c:\dev\flight> ruby script\generate model user
    exists  app/models/
    exists  test/functional/
    exists  test/unit/
dependency  model
    exists    app/models/
    exists    test/unit/
    exists    test/fixtures/
    create    app/models/user.rb
    create    test/unit/user_test.rb
    create    test/fixtures/users.yml
    exists    db/migrate
    create    db/migrate/003_create_users.rb
```

Now, let's update the migration, keeping it simple for the purposes of this example:

Download controller/003_create_users.rb

```
class CreateUsers < ActiveRecord::Migration
  def self.up
    create_table :users do |t|
      t.string :login, :password
      t.timestamps
    end
  end

  def self.down
    drop_table :users
  end
end
```

Execute the migration by doing a rake db:migrate. A simple users table with login and password fields is set to go.

The next step is to build a method in the User class that tests for a valid login/password combination:

Download controller/user.rb

```
class User < ActiveRecord::Base

  def self.authenticate(login, password)
    User.find_by_login_and_password(login, password)
  end

end
```

In this very simple implementation, we're simply checking the database for the existence of a record that contains that login and password combination. If it is found, we'll return the corresponding User object. If it is not found, we'll return nil.

Let's fire up the console to add a user to the database and to test the authenticate method we just wrote:

```
c:\dev\flight> ruby script\console
>> User.create(:login => 'brian', :password => 'foo')
  User Create (0.000489)   INSERT INTO users ("updated_at", "login",
  "password", "created_at") VALUES('2008-02-27 12:57:19', 'brian',
  'foo', '2008-02-27 12:57:19')
=> #<User id: 1, login: "brian", password: "foo", created_at:
  "2008-02-27 12:57:19", updated_at: "2008-02-27 12:57:19">
  >> User.authenticate('brian', 'foo')
  User Load (0.000317)   SELECT * FROM users WHERE (users."password" =
```

```
=> #<User id: 1, login: "brian", password: "foo", created_at:
  "2008-02-27 12:57:19", updated_at: "2008-02-27 12:57:19">
>> User.authenticate('brian', 'bar')
  User Load (0.000290)   SELECT * FROM users WHERE (users."password"
  = 'bar' AND users."login" = 'brian') LIMIT 1
=> nil
```

We created a user with a login and password, and we used the authenticate method we wrote to test against the user record. In the first case, we supplied a valid login/password combination, so it returned a User model object that corresponds to the database record that was found. In the second case, we supplied an invalid combination of values, so it returned nil. It works.

Now that the model layer is established, it's time to get going on the controller.

Setting Up Routing and a Login Form

Now, we'll generate a controller for a session resource. A session, in the context of our application, is a RESTful resource that we'll use exclusively for login purposes. *Creating* a new session will represent logging in, and *destroying or deleting* a session will log the user out. We can easily accomplish the same thing using a LoginController with login and logout actions, but we do want to keep our application RESTful whenever possible.

```
c:\dev\flight> ruby script\generate controller sessions
exists  app/controllers/
exists  app/helpers/
create  app/views/sessions
exists  test/functional/
create  app/controllers/sessions_controller.rb
create  test/functional/sessions_controller_test.rb
create  app/helpers/sessions_helper.rb
```

The generator creates the controller we want, along with the associated test and helper. It's time to create the login form:

Download controller/new.html.erb

```
<% form_tag sessions_url do %>
  <p>
    <label for="login">Login</label>
    <%= text_field_tag :login %>
  </p>

  <p>
    <label for="password">Password</label>
    <%= password_field_tag :password %>
```

```
  <p>
    <%= submit_tag "Submit" %>
  </p>
<% end %>
```

This will be a simple form that has two input fields, one for the login and the other for the password, and that has a submit button. These inputs will be wrapped by a form_tag. The form_tag helper method accepts, at a minimum, a URL that the form data will be submitted to—in this case, the create action. If we think back to our RESTful resource-based routes (see Figure 5.2, on page 98), we'll recall that the create action will be called with a POST to the /sessions URL. Since we've passed a URL of sessions_url to our form_tag helper and we haven't explicitly defined any other method, Rails assumes a POST, and we'll get exactly what we want when the submit button is clicked, provided that we supply a map.resources call in our routes file:

Download controller/routes_with_sessions_resource.rb

```
ActionController::Routing::Routes.draw do |map|
  map.resources :sessions
  map.resources :flights
  map.resources :passengers

  map.root :controller => 'passengers'
end
```

By calling map.resources :sessions in our routes, we are letting Rails know that sessions is a valid resource in our application, and therefore, we want the seven routes defined for us.

Using the Session and Filtering

The next thing to do is write a create action—the target for the form submission:

Download controller/sessions_controller.rb

```
class SessionsController < ApplicationController

  def create
    if User.authenticate(params[:login], params[:password])
      redirect_to root_url
    else
      render :action => 'new'
    end
  end

end
```

The login and password supplied will be in the params hash when it gets to the controller. In the create action, we're simply handing those values to the User.authenticate method we wrote earlier. If it returns anything (that is, a User model object), the login has succeeded, and we'll redirect the user to the root URL of the application. If not, we want to re-render the login form.

We're almost there. The last hurdle is that, although the login form works, all this code has nothing to do with access to the application. We can still access any controller/action regardless of whether we've logged in successfully. To lock the application down, we'll have to use a simple *filter*.

Filters allow processing to happen before and/or after an action. In our login use case, we want to check that a user is logged in before executing any action. So, we'll use a before_filter in the ApplicationController:

Download controller/application.rb

```
class ApplicationController < ActionController::Base
  helper :all
  before_filter :login_required

  private

    def login_required
      unless logged_in?
        session[:return_to] = request.request_uri
        redirect_to new_session_url
        return false
      end

      @current_user = User.find(session[:user_id])
    end

    def logged_in?
      !session[:user_id].blank?
    end

end
```

Every controller in a Rails application inherits from ApplicationController, so every filter defined in it applies to all controllers. Here, we've declared that the login_required method should be executed before all actions in all controllers. In this method, we want to check whether the user is already logged in, which we determine by examining the value of session[:user_id]. If the user is not already logged in, we do two things. First, we store the URL that the user is trying to get to in the ses-

Joe Asks...

What Is the Session Store, Exactly?

When we do something like session[:user_id] = 123, we're asking Rails to tuck the data we give it in a special hash object reserved for storing temporary, application-specific data. Whatever is stored in the session hash persists until the session ends—typically when the end user's browser is closed.

Just like in ASP.NET, we have several options as to how this data is stored. By default, it's stored in browser-based cookies, but we can configure the session store to be in memory, in files, or in the database, or to use some other server process like memcached. We can even write our own class to specify how we'd like it to work.

page, /sessions/new, using the helper method new_session_url. If the user is already logged in, we grab the user record from the database and make it available to the entire application via the @current_user variable. To hook the whole thing up, we must also make some changes to the sessions_controller:

Download controller/sessions_controller_with_login_code.rb

```
class SessionsController < ApplicationController
  skip_before_filter :login_required

  def create
    if user = User.authenticate(params[:login], params[:password])
      session[:user_id] = user.id
      redirect_to session[:return_to] || root_url
    else
      render :action => 'new'
    end
  end

end
```

The first change is an important one. We have to put a skip_before_filter in this controller, which will bypass the before_filter we've set up in the ApplicationController. If we don't do this, we'll be caught in an infinite loop where we'll try to access the login page, not be logged in, and be redirected to it over and over again.

Joe Asks...

Why Are We Storing the User ID and Not the Whole User Object in the Session?

One potential drawback to the approach we've taken here is that the login_required method makes an extra database hit with every request. We could store the whole User object instead, but if the User object changes for any reason (for example, we add a field), the session could become corrupted. So, it's a small price to pay for better stability.

Then, in the create action, we added some code that stuffs the ID of the user into the session once the authentication test passes. This will, in turn, be used in the login_required method to determine whether we're logged in. Finally, we redirect to the URL originally requested or the root of the application if the login page was accessed directly.

More Temporary Storage—Cookies and Flash

So far, we've been working on only one type of temporary data store that Rails provides, the session. Just like ASP.NET, we also have cookies, a mechanism for interacting directly with browser cookies. Rails also provides us with flash. Regardless of how any of these temporary storage engines work, we interact with them as if they were normal Ruby hashes. If we wanted to store the login name in a cookie so that the user wouldn't have to type it in every time they visited, we could make a simple change to the SessionsController:

Download controller/sessions_controller_with_cookies.rb

```
class SessionsController < ApplicationController
  skip_before_filter :login_required

  def create
    if user = User.authenticate(params[:login], params[:password])
      session[:user_id] = user.id
      cookies[:login] = { :value => user.login, :expires =>
                                                    1.week.from_now }
      redirect_to session[:return_to] || root_url
    else
      render :action => 'new'
    end
  end
```

Here, upon successful login, we are storing the value of the login name in a cookie that is set to expire in one week. We can then pick up this value in the view:

Download controller/new_with_cookies.html.erb

```
<% form_tag sessions_url do %>
  <p>
    <label for="login">Login</label>
    <%= text_field_tag :login, cookies[:login] %>
  </p>

  <p>
    <label for="password">Password</label>
    <%= password_field_tag :password %>
  </p>

  <p>
    <%= submit_tag "Submit" %>
  </p>
<% end %>
```

Now, when the user hits the login page, the login field will be populated with the cookied value, provided that it has been less than a week since the last login.

The flash storage bucket is unique to Rails; there's not really an equivalent to it in ASP.NET. Data stored in flash stays around for exactly one request—then it's deleted. This therefore makes it a convenient mechanism for conveying error messages and other short bits of information to the user.

Returning to our login scenario, we're handling a successful login pretty well, but an unsuccessful login simply re-renders the login form—not a very user-friendly experience. It would be a lot more usable if we at least displayed a message letting the user know that the login was unsuccessful. Let's modify our SessionsController and login form view to make that happen:

Download controller/sessions_controller_with_flash.rb

```
class SessionsController < ApplicationController
  skip_before_filter :login_required

  def create
    if user = User.authenticate(params[:login], params[:password])
      session[:user_id] = user.id
      cookies[:login] = { :value => user.login, :expires => 1.week.from_now
      redirect_to session[:return_to] || root_url
```

```
      flash[:notice] = "Sorry, that is not a valid login/password combinati
      render :action => 'new'
    end
  end

end
```

Download controller/new_with_flash.html.erb

```
<% if flash[:notice] %>
  <p>
    <%= flash[:notice] %>
  </p>
<% end %>

<% form_tag sessions_url do %>
  <p>
    <label for="login">Login</label>
    <%= text_field_tag :login, cookies[:login] %>
  </p>

  <p>
    <label for="password">Password</label>
    <%= password_field_tag :password %>
  </p>

  <p>
    <%= submit_tag "Submit" %>
  </p>
<% end %>
```

The interface to flash is just like session and cookies—it behaves just like an ordinary Ruby hash. We've used a key of notice here; we could have easily used any key we wanted. Then, we've modified the view to display the flash message above the login form if it is populated. Now, when the login is unsuccessful, the user will receive an informative message in addition to being shown the login form again.

Providing an API

We often want to provide our users with a way to talk with our applications without actually using them in a web browser. For example, we may want to allow our users to consume our data from a desktop application, mobile device, or another web application. The .NET and Rails approaches to doing this are quite different in both philosophy and execution.

In this example, we'll be creating an API in both .NET and Rails to retrieve flight data from our application.

How You Might Approach It in .NET

The .NET way of exposing data to the outside world is through *web services*. .NET web services use a combination of XML-based technologies—SOAP and WSDL—along with some code-generation magic that allows developers to easily create robust APIs by writing essentially ordinary methods.

To expose these methods as web services, we simply put them in a file with an .asmx extension (as opposed to .aspx for a regular web form) and tag them with a WebMethod declaration. Here's how our simple flight web service might look:

Download controller/Flights.asmx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Services;

namespace FlightMvc
{
    [WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [System.ComponentModel.ToolboxItem(false)]
    public class Flights : System.Web.Services.WebService
    {
        [WebMethod]
        public List<flight> GetAllFlights()
        {
            FlightDataContext flightData = new FlightDataContext();
            return flightData.flights.ToList();
        }
    }
}
```

We're simply using LINQ to retrieve flight data from our database and returning a List of flight objects to the consumer of our web service. The typical way for the consumer to call our web method would be to send a *SOAP message* to it via HTTP. SOAP is nothing more than plain ol' XML that follows the specifications for it laid out by the W3C.

When submitted to our web service resource (our ASMX page) via HTTP, this SOAP request:

Download controller/soap_request.xml

```
<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <GetAllFlights xmlns="http://tempuri.org/" />
  </soap12:Body>
</soap12:Envelope>
```

would return the following response:

Download controller/soap_response.xml

```
<?xml version="1.0" encoding="utf-8"?>
<soap12:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:soap12="http://www.w3.org/2003/05/soap-envelope">
  <soap12:Body>
    <GetAllFlightsResponse xmlns="http://tempuri.org/">
      <GetAllFlightsResult>
        <flight>
          <id>1</id>
          <number>123</number>
          <departs_at>2008-06-28T00:00:00</departs_at>
          <arrives_at>2008-06-28T01:30:00</arrives_at>
          <departure_airport>ORD</departure_airport>
          <arrival_airport>SFO</arrival_airport>
        </flight>
        <flight>
          <id>2</id>
          <number>369</number>
          <departs_at>2008-06-28T13:30:00</departs_at>
          <arrives_at>2008-06-28T16:30:00</arrives_at>
          <departure_airport>ORD</departure_airport>
          <arrival_airport>PDX</arrival_airport>
        </flight>
      </GetAllFlightsResult>
    </GetAllFlightsResponse>
  </soap12:Body>
</soap12:Envelope>
```

Since our web method returned a List of flight objects, .NET automatically translates that to a standard XML format that simply lists all the fields and the corresponding values in each record. In addition, .NET

web services also publish a WSDL[4] by default, providing consumers with a way to know which API methods are available, along with their expected parameters and return values. This makes it dead simple to consume a .NET web service from another .NET application, because full IntelliSense is provided.

Although the .NET-to-.NET scenario couldn't be smoother, a common criticism of this approach is its level of complexity and nonconformance to the standards of HTTP. As we'll see now, Rails takes a quite different, RESTful approach to APIs.

The Rails Way

We've already talked at length about Rails' adoption of REST principles and the advantages associated with it. Unsurprisingly, Rails also takes a RESTful approach to APIs and web services; after all, the typical use of our application's API is nothing more than a client, other than the web browser performing CRUD operations on our resources.

At the heart of Rails' web service support is ActionController's respond_to method. Without web service support, our FlightsController's index action looks like this:

```
def index
  @flights = Flight.find(:all)
end
```

As we've already seen, this simply puts all the flight data into an Array, which is then stored in an instance variable for the view to use. Without any other information given, Rails is going to look for index.html.erb and attempt to render that view as HTML to be returned to the browser. Now, let's add web service support:

```
def index
  @flights = Flight.find(:all)

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

This code (which is actually the default code generated by the scaffold generator) now supports two types of responses, HTML and XML. To

4. Web Services Description Language, an XML-based language that describes a web

determine the type of response the client wants, Rails looks at the HTTP request and inspects the value of the Accept header. If it's text/html, it will render HTML as before. If the client wants text/xml, we'll ask for the @flightsArray to be serialized into XML and returned.

Another way we can get XML back, without setting the Accept header ourselves, is to explicitly ask for it in the URL. This is also a nifty way to perform a quick test to see whether the XML response returns what we want. To do this, we simply add the .xml extension to any RESTful URL. So if we hit the URL /flights.xml in our web browser, we'll get back an XML-formatted response:

Download controller/flights.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<flights type="array">
  <flight>
    <arrives-at type="datetime">2008-06-25T16:00:00-05:00</arrives-at>
    <created-at type="datetime">2008-06-25T12:44:43-05:00</created-at>
    <departs-at type="datetime">2008-06-25T12:00:00-05:00</departs-at>
    <destination>PDX</destination>
    <flight-number type="integer">1234</flight-number>
    <id type="integer">1</id>
    <origin>ORD</origin>
    <updated-at type="datetime">2008-06-25T12:44:43-05:00</updated-at>
  </flight>
  <flight>
    <arrives-at type="datetime">2008-06-25T17:00:00-05:00</arrives-at>
    <created-at type="datetime">2008-06-25T12:45:11-05:00</created-at>
    <departs-at type="datetime">2008-06-25T16:00:00-05:00</departs-at>
    <destination>SFO</destination>
    <flight-number type="integer">3333</flight-number>
    <id type="integer">2</id>
    <origin>PDX</origin>
    <updated-at type="datetime">2008-06-25T12:45:11-05:00</updated-at>
  </flight>
</flights>
```

The :xml option in our render method automatically takes the ActiveRecord object and converts it to XML, providing all the fields in the table by default. Unlike .NET web services and SOAP, it's up to us as the developers to determine what will actually be sent back to the consumer of the service, and we can make it as simple or as robust as we want. And, although it is outside the scope of this book, Rails uses the technology we've just described to make quick work of communicating with other Rails applications through ActiveResource.

Bonus Round: Creating an iPhone Version of Our App

respond_to can be used for many other purposes other than creating an API. One of the obvious applications of it is to create separate views of our application for use on mobile devices or perhaps to shoot data out to an Excel spreadsheet.

Consider a respond_to block like this:

```
respond_to do |format|
  format.html
  format.js
  format.xml  { render :xml => @flights }
end
```

The html, js, and xml methods are saying that this action knows how to respond to HTML, JavaScript, and XML MIME types, respectively. Although these are the defaults, these aren't the only MIME types you can specify. In fact, you can add any MIME types you want, just by adding a few lines of code in the config/initializers/mime_types.rb file:

Download controller/mime_types.rb

```
Mime::Type.register "text/richtext", :rtf
Mime::Type.register "application/vnd.ms-excel", :xls
Mime::Type.register_alias "text/html", :iphone
```

We are telling the Rails environment, in addition to supporting the default HTML, JavaScript, and XML formats, to also support Rich Text (RTF), Excel, and iPhone. The first two are new MIME types that we'd like to be able to handle; the iPhone type is an alias for the text/html type we're already handling with the default HTML format but one we'll need to have in order to differentiate between the regular HTML and iPhone responses in our code. Our respond_to block might now look like this:

```
respond_to do |format|
  format.html
  format.js
  format.xml     { render :xml => @flights }
  format.xls     # render a template that creates a CSV file
  format.iphone  # render a template for the iPhone
end
```

We can either pass a block into the respond_to block for each MIME type, like we've done here with the XML type, or just let Rails' defaults take over. For instance, if the format is set to iphone, the controller will then look for a file called index.iphone.erb instead of index.html.erb in the usual, HTML-formatted case. But how do we tell Rails the format in which we'd

like to receive our response? We've already seen two different ways; let's review and talk about a third:

- Change the HTTP Accept header to the appropriate MIME type; for example, setting the Accept header to text/xml will set the format to XML.
- Tack the appropriate extension onto the end of a RESTful URL; for instance, a URL of /flights/4.iphone will ask for our show action, with an ID parameter of 4, and set the format to iphone.
- Set it explicitly in code, for example, request.format = :iphone.

Here is our FlightsController, optimized for both XML and iPhone consumption:

Download controller/flights_controller.rb

```
class FlightsController < ApplicationController
  before_filter :adjust_format_for_iphone

  def index
    @flights = Flight.find(:all)

    respond_to do |format|
      format.html    # render html
      format.xml     # render xml
      format.iphone  # render for the iPhone
    end
  end

private

  def adjust_format_for_iphone
    if request.env["HTTP_USER_AGENT"]
      && request.env["HTTP_USER_AGENT"] =~ /(Mobile)\/.+Safari)/
      request.format = :iphone
    end
  end
end
```

Here, we have our respond_to block that is set up to handle three formats—HTML, XML, and iPhone. We also have a before_filter in place that inspects the user agent string of the client and manually sets the desired format to iphone if the request is being made from the iPhone. The respond_to block then handles the rendering of the appropriate template based on the format.

In this chapter, we've explored many aspects of Rails controllers. We

of a controller and how we can do some pretty diverse application-level URL rewriting using Rails' built-in routing functionality. We've also walked through a user authentication scenario, taking a look at filters and the different types of temporary application data storage (sessions, cookies, and the flash) in the process. Finally, we created a simple API for our application using Rails' respond_to mechanism, with the added bonus of seeing how we might deliver an iPhone-friendly version of our application.

In this chapter and the previous one, we've been briefly creating views and building some basic front ends using ERb. Next, we're going to go in depth with ActionView.

Chapter 8

Exploring Forms, Layouts, and Partials

Among the three sides of the MVC triangle, it's the *V*—the view—that seems to get all the attention. Views in an MVC framework represent the ways in which we display the output of our application. To be sure, controllers are important because they receive input from the user and act as central switchboard operators. Models are also crucial because they help us represent our problem domain, they contain our business rules, and they provide us with data. However, without views, there would really be no application at all.

HTML forms are the primary way that web applications receive user input, and in this chapter we will take a closer look at how we create forms with Rails. We will also explore the concept of *layouts*, which help us refactor common HTML code out of individual view templates and enables both controller-wide and site-wide layouts. Finally, we will take a look at *partials*, which give us the ability to share reusable fragments of templated code across actions or even across controllers.

Diving Into Forms

We'll see how to create forms in Rails by revisiting our Flight model and views. When we used the scaffold generator for the flight resource, we got a lot of functionality for free. Without having to do any more work, we could simply start our local server and navigate to the form that lets us create a new flight, as shown in Figure 8.1, on the following page.

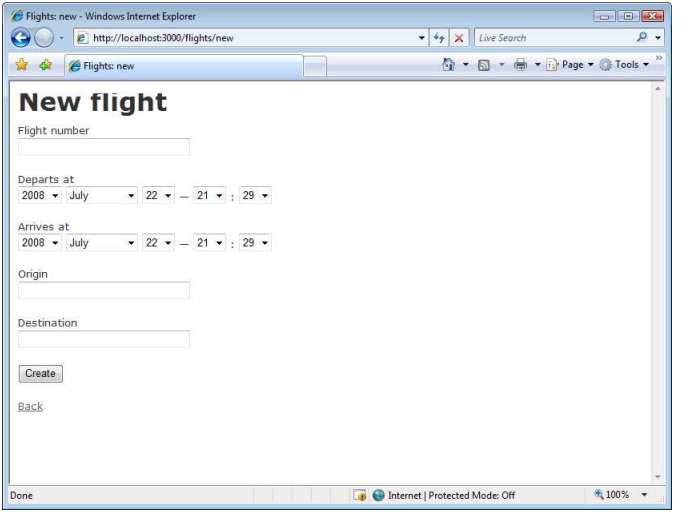


Figure 8.1: Scaffold-generated “new flight” form

The life cycle of a form on a web page generally involves two phases:

- Generating the appropriate HTML for the form, which includes the proper assignment of the form tag’s attributes so that the submit button will trigger the appropriate login on the server. It may also include setting default values for some of the controls so that the user has less work to do to complete the form.
- Processing the results of the form and updating the database as necessary.

ASP.NET forms can be difficult to build and process without aid from the Visual Studio IDE to help generate the necessary HTML. In Rails, all the development can be done “by hand” with a text editor. Let’s take a look.

How You Might Approach It in .NET

Let’s take a quick tour of how a simple new flight form might be created in ASP.NET. Creating a form in ASP.NET is fairly easy. A WYSIWYG design surface not only lets you drag and drop an ASP.NET server control onto the form, but we can also use the visual property grid to make changes without having to look at the underlying source. In practice, however, most ASP.NET developers switch to Source View for the necessary fine-tuning of the HTML and server properties that may not be

the HTML that will be returned to the browser, as we will see in a moment.

Our form is going to need a table underlying it so we can save our flight data somewhere. Visual Studio and ASP.NET's data-binding features provide a wealth of choices for binding your form elements to data sets or custom queries. For our simple example, we've chosen to use a SQL Server database .mdf file, and we added a table named flights. We then used the Project > Add New Item menu to create a TableAdapter object that wraps the entire flights table for us (see Figure 8.3, on page 154).

Creating the Form

Once we've created the database file, a flights table, and a TableAdapter class, we can start the business of creating the form (see Figure 8.2, on the following page). For our example, we simply dragged some label and textbox controls onto the designer surface and set their Name properties as desired. Switching to Source View reveals the following code:

Download view/netsource.html

```
<body>

    <h1>New Flight</h1>

    <form id="form1" runat="server">

    <p><asp:Label ID="Label1" runat="server"
        AssociatedControlID="flightNumber"
        Text="Flight Number:">
      </asp:Label>
    </p>
    <p><asp:TextBox ID="flightNumber" runat="server"></asp:TextBox></p>

    <p><asp:Label ID="Label2" runat="server"
        AssociatedControlID="origin"
        Text="Origin:"></asp:Label></p>
    <p><asp:TextBox ID="origin" runat="server"></asp:TextBox></p>

    <p><asp:Label ID="Label3" runat="server"
        AssociatedControlID="destination"
        Text="Destination:"></asp:Label></p>
    <p><asp:TextBox ID="destination" runat="server"></asp:TextBox></p>

    <p>
        <asp:Button ID="Button1" runat="server" onclick="Button1_Click"
            Text="Create" style="height: 26px" />
    </p>
    </form>
```

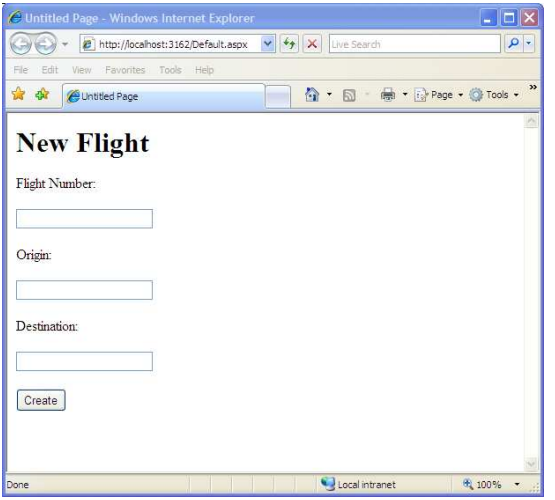


Figure 8.2: Creating a flight in ASP.NET

On line 5, we see the first clue that we're not looking at raw HTML. Browsers don't understand runat attributes of HTML elements. The runat attribute of the form element is specific to ASP.NET, and it helps to endow our form with special properties and behaviors by hooking into the HTTP request/response pipeline. When we run our application in a browser and then view the raw HTML source, we can see how ASP.NET synthesized together the actual HTML:

Download view/nethtml.html

```
<form name="form1" method="post" action="Default.aspx" id="form1">
```

ASP.NET did the work of specifying the action, method, and id attributes that the browser will use to submit the form to our application.

Similarly, we can see how the Label and TextBox controls are rendered into the HTML counterparts. Take a look at how the label and textbox pair on lines 1 to 2 became rendered into HTML:

Download view/nethtml.html

```
<p><label for="flightNumber" id="Label1">Flight Number:</label></p>
<p><input name="flightNumber" type="text" id="flightNumber" /></p>
```

Beginning ASP.NET developers may not realize that Source View in Visual Studio is not HTML source or is it the code for our code-behind

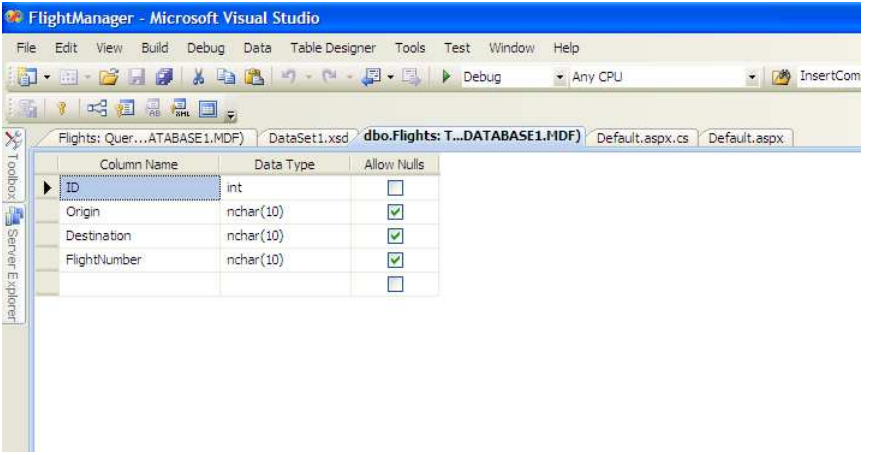


Figure 8.3: flights table definition in Visual Studio

classic ASP we called these templates *ASP script files*. In Rails, they are called *view templates*.

Saving the Flight to a Database

By using the Table Adapter Wizard, we ended up with a FlightsManager-Adapter in our code. If we ever need to change our table—say we add or drop a column—we would need to be sure to regenerate our code.

When the user clicks the submit button, we need to save our flight data to the database, and we do that by double-clicking the button in the designer to open our code-behind editor.

Download view/netcode.cs

```
protected void Button1_Click(object sender, EventArgs e)
{
    string number = this.flightNumber.Text;
    string originAirport = this.origin.Text;
    string destinationAirport = this.destination.Text;

    DataSet1TableAdapters.FlightsTableAdapter adapter =
        new FlightManager.DataSet1TableAdapters.FlightsTableAdapter();

    int flightID = adapter.Insert(originAirport,
                                  destinationAirport,
                                  number);
```

Starting on line 3, we capture the values entered into the form. On line 7, we create a new instance of our adapter, and finally on line 12, we insert a new row into the table.

We can become as elaborate with this form as we want, adding validation (see Section 6.3, *Validating User Input*, on page 119 for details), CSS support, theming and skinning, and a variety of ways to bind our form elements to the database. But the general principles we've seen here will remain the same:

- ASP.NET forms have an underlying Source View that is the "script" or "template" for the page.
- The ASP.NET machinery will transform the template into raw HTML, suitable for browser consumption.
- Code-behind partial classes can give the appearance of an event-driven model for your web applications, and ASP.NET takes care of the actual request/response processing behind the scenes.
- Data binding in .NET involves using some kind of code-generated data source control or class and using the facilities of that class to insert/update/delete data in your tables.

With our refresher of ASP.NET form basics complete, let's see how we develop forms in Rails.

The Rails Way

Developing web forms in Rails is a great way to learn about the cornerstones of Ruby on Rails.

Every form in Rails will involve these central pieces of the framework:

- Models: These are Ruby classes that are usually derived from ActiveRecord::Base, but they are any Ruby class that you write that helps model your particular problem domain. Forms in Rails tend to be for the purpose of creating or editing instances of a particular model.
- Views: These are HTML files that contain embedded Ruby code. Rails generates a pure HTML response by executing any embedded Ruby statements first and then sending the entire result to the browser. Rails provides many helper methods you can embed inside your HTML forms that help bind form controls to model attributes automatically.

- Controllers: These are your Ruby classes that are first activated by Rails and determine many things, including which models to manipulate and which view should be rendered to the browser.
- Routing: The routes.rb file is consulting for every incoming HTTP request. One of the most important results of the routing machinery is to generate a Ruby hash that contains all the data sent by the browser. This hash is the params hash. All data entered by the user into a form is stored in this hash.

If this sounds like a lot for a simple form, you may be surprised to learn that these same elements exist in ASP.NET as well. ASP.NET kept most of these gory details hidden from us to shield us from its internals.

But the invisible is also unchangeable. Rails brings these elements into the light so that we can tweak, customize, and optimize our application as we see fit. Rails gets us "close to the metal" of the HTTP request/response cycle for detailed control, while Ruby enables our code to remain at a comfortably high level of abstraction.

We can learn about how these elements fit together by looking at the code that was generated for us by the script/generate scaffold command.

Forms for an ActiveRecord Model

Rails applications spend a lot of time interacting with their underlying ActiveRecord-derived models. It should be no surprise that Rails, therefore, makes it easy to create forms that have a one-to-one mapping to an ActiveRecord object. Our embedded Ruby templates (those that end with .erb) can use the form_for method to wrap an ActiveRecord model instance with a form.

Go ahead and open the app/views/flights/new.html.erb template, and you will find a line like this:

```
<% form_for(@flight) do |f| %>
```

Here we're calling the form_for method, passing it the @flight variable we created in the new method of our controller. form_for will take care of generating the underlying HTML form tag for us with the appropriate attributes. Use your browser to view the source for the form, and you'll see that the form tag was generated like this:

```
<form action="/flights" class="new_flight" id="new_flight" method="post">
```

The action attribute is set to /flights, while the method attribute is set to post. According to REST routing rules, this will mean that the submit

Joe Asks...

What If I Need to Call the Edit Action Instead?

The form_for code in the new.html.erb template seems to "know" that we want to trigger the create action. But if we're editing an existing flight, we would want it to call the edit action instead.

It turns out that form_for is pretty smart. If the variable we pass to it (@flight in our case) is a new, unsaved model instance, then it will set the action and method attributes to values that will in turn trigger the create action.

But if the variable is already associated with an existing row in the database, it will generating HTML like this:

```
<form action="/flights/1" class="edit_flight"
  id="edit_flight_1" method="post">
```

Here, the action targets a specific flight resource with a method of POST. This combination will instead trigger the edit action of the FlightsController.

button for this form will trigger the create method of the FlightsController—exactly what we want!

Eagle-eyed readers will notice that form_for is a method that expects a user-supplied block. The block is given an object that can be used inside the block to assist in generating HTML client-side controls. The scaffold-generated code uses the common convention of naming this variable f. This variable is valid only inside the form_for block and is called a *form builder helper*. The form builder helper provided by form_for provides the equivalent functionality of control data binding in .NET.

We are now ready to take a look at how form_for enables us to easily bind HTML controls to Flight object attributes—in other words, to columns in our flights table.

Data-Bound Textbox

In .NET, textbox controls are used whenever we need an HTML text input control. In Rails, we use the text_field helper.

Look at how the first textbox is implemented in our form:

Download view/new.html.erb

```
<%= f.text_field :flight_number %>
```

Here we simply call the text_field method, passing it a symbol representing the method on our model that we want to bind to the textbox. If we view the raw HTML source in our browser, we will see this:

```
<input id="flight_flight_number" name="flight[flight_number]"
  size="30" type="text" />
```

Just as ASP.NET transforms <asp:.....> controls into raw HTML for the browser to display, Rails generates this HTML for us when the template is rendered.

The text_field method, like all the form helper methods, can accept many optional parameters that we can use to specify HTML attributes like the size of the text field, the CSS class we want applied to the control, the HTML ID, and more. See the Rails API for complete details.

How Data Binding Works in Rails

The one HTML attribute that you will not want to override is the name attribute. The form builder helper has assigned the name specially to follow Rails conventions. When this form is submitted, the browser will send the name and value of this textbox control. The name becomes a key into the params hash in our controller.

Something cool happens when the name of a control includes square brackets. When the form is submitted, Rails will break apart this name and create a new hash for us inside the params hash. params[:flight] returns this hash, in which the keys represent the controls and the values are the corresponding user-supplied values for those controls. So in our case, params[:flight][:flight_number] will return exactly what the user entered into our textbox.

That's exactly what we find inside the create action in our FlightsController. This code in our controller:

Download view/flights_controller.rb

```
@flight = Flight.new(params[:flight])
```

creates a new instance of Flight, initialized with all of the values entered by the user.

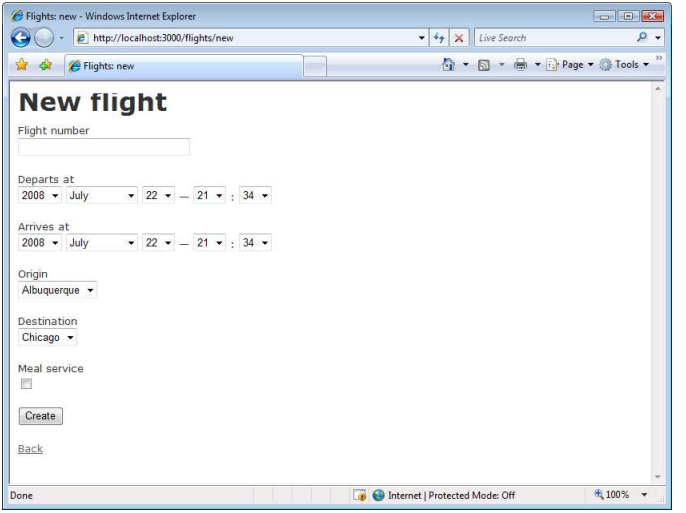


Figure 8.4: “New flight” form with enhancements

By defining conventions for HTML name attributes, Rails controllers can create a new model instance or update an existing model with one line of Ruby code.

When we use form_for and use the form builder object to create the form’s HTML controls, we get HTML that conforms to these conventions. The framework is able to automatically map the incoming HTML values into a hash that contains all the values entered into our form.

That alone would be very handy, but the story gets even better. ActiveRecord models can accept this hash of keys and values when creating new instances by using the new or create class methods, and existing rows can be updated instantly with the update_attributes method available on every ActiveRecord object.

This is another example of how powerful our code becomes when we follow conventions defined by Rails.

Combo Boxes and List Boxes in Rails

The scaffold generator created textboxes on our form for the origin and destination attributes of our model, where users would enter an airport code like ORD or ATL. It would be better if we could allow them to select the airport codes from a predefined list (see Figure 8.4).

Let's generate a new Airport resource:

```
ruby script\generate scaffold Airport code:string city:string
rake db:migrate
```

We can create our airports by manually with the Rails console or by starting up our local web server, then navigating our browser to http://localhost:3000/airports, and using the scaffold interface to create airport data.

Once we have some airports to play with, we can change the origin and destination controls into combo boxes. First we need to edit the new action in our controller to get a list of all the airports.

Download view/flights_controller.rb

```
@airports = Airpot.find(:all, :order => "city asc")
```

and then we use the collection_select helper method in our view:

Download view/new.html.erb

```
<p>
  <b>Origin</b><br />
  <%= f.collection_select :origin, airports, :code, :city %>
</p>

<p>
  <b>Destination</b><br />
  <%= f.collection_select :destination, airports, :code, :city %>
</p>
```

The collection_select takes four parameters:

- The attribute of our Flight model that we want to be bound to the combo box.
- The collection to use to populate the combo box.
- The method that should be called on the selected item when the form is submitted. The return value of this method is what gets sent to the application.
- The method that should be called on each item to be used as the displayable string in the combo box.

If we prefer to show a listbox instead, we need to use the full form of the collection_select method that takes six parameters, where the fifth and sixth are hashes. The sixth parameter is a hash of HTML options, and that's the one we need to use to specify the size of our list. Here's how we would turn the origin combo box into a listbox with a height of ten items:

```
<%= f.collection_select :origin, airports, :code, :city, {},
                        { :size => 10 } %>
```

Other Data-Bound Controls

The form_for block object can help build other controls as well:

- check_box binds a checkbox to a boolean attribute.
- password_field binds a textbox to a string attribute but masks the input.
- hidden_field creates a hidden value that will be sent to the application when the form is submitted.
- radio_button binds a radio button to a model attribute. Multiple radio buttons for the same attribute are considered to be a radio button group.
- text_area is similar to text_field but allows multiple lines of input.

Each of these helper methods "bind" HTML controls to data columns simply because they'll use generate the proper name values in the resulting HTML code. Your create or update actions in your controller perform the heavy lifting by calling the create or update_attributes method on your ActiveRecord model.

We've toured the basics of creating forms in Rails. The following sections will take us into two other aspects of view management in Rails: layouts that help us provide a consistent look and feel among all of our pages; and partials, which will help us avoid repeating the same code from one view to the next.

Using Layouts Instead of Master Pages

ASP.NET 2.0 introduced the concept of *master pages* so that site-wide logic and HTML can be applied across all pages that share the same structure and layout.

In Rails, we use *layouts* to apply a common HTML structure to our entire application or for specific controllers. Since layouts in Rails are normal view templates that can contain embedded Ruby code, layouts provide one way for us to dynamically "theme" our site. We can determine styles and page structure dynamically based on environment settings, user profiles, configuration files, time of day, data in our database, or anything else we can dream of.

Layouts also serve a second purpose. Ruby developers like to keep their

isolates faulty code to a single spot so that a fix can be performed in just one place, and helps maintain readable code as the size of the application grows larger. We want to treat our view templates as first-class code citizens in our application and strive to keep our view code as DRY as possible as well. Layouts help us refactor common code out of our views so that we don't need to repeat the same code in each view template.

Master Pages in ASP.NET

Master pages help enforce a standard structure or layout for every page on the site. Master pages have the file extension .master and use a @ Master directive instead of the usual @ Page. This placeholder would be replaced at runtime with the content of the current page.

Master pages are similar to regular .aspx files, but at some point in the code there must be a ContentPlaceHolder control. As its name suggests, the placeholder is replaced at runtime by an actual .aspx page, wrapped by the outer master layout. The page being wrapped is often referred to as the *content* page.

Connecting a content page with its master page involves setting the MasterPageFile property in the content page's @ Page directive. We usually do this from the Add Item Wizard in Visual Studio, but it can also be done by hand later. And although they are similar to regular .aspx pages, content pages do not contain HTML tags such as <html>, <body>, or even <form>.

The motivation behind master pages is to help ease the development of sites where pages share a common layout so that the common elements can be specified once in the master page. Changes to the common structure can be done in the master page, and all pages that use that master page will automatically use the new structure. At runtime, ASP.NET will merge the content page into the surrounding master page.

Rails also provides a mechanism for sharing common structure among pages by introducing the notion of a *layout template*. If you've been using master pages in your ASP.NET projects, using layouts will feel natural and easier to use than master pages.

Using Layouts in Rails

Layouts differ from master pages in a few ways:

- Layouts are either controller-wide or site-wide.
- Instead of a "placeholder" tag in the template, layouts use the Ruby keyword yield.
- Views are automatically merged into the controller or site-wide layout. No separate step is needed to connect a view to its layout.

Layouts are normal embedded Ruby templates to generate HTML, with a twist. Layouts will be "wrapped around" your more specific, action-based view templates. The layout combines with an action-specific template to generate the complete HTML for a web page. Using layout files is entirely optional, but most projects benefit from using even simple layouts.

You can incorporate layouts into your views in three ways:

- A controller-specific layout will automatically be used if it exists. Each controller can contribute a file into the app/views/layouts/ directory if it follows the specific naming convention of controller.html.erb. For example, app/views/layouts/flights.html.erb is the layout template that will wrap around all templates served by the FlightsController.
- A site-wide default layout will be used if a controller-specific layout does not exist. The site-wide layout file must be named application.html.erb. If the site-wide layout and a controller layout both exist, the controller layout takes priority. This means you can define a site-wide layout but override it for specific controllers as needed.
- Specify the layout file in your controller's Ruby code. You can explicitly call the layout class method in your controller to define the layout file you'd like to use for your controller:

```
class FlightsController < ApplicationController

  # Use app/views/layout/my_special_layout
  # for every action rendered by this controller

  layout "my_special_layout"
end
```

If needed, you can override the layout setting for a specific action:

```
class FlightsController < ApplicationController

  # Use app/views/layouts/my_special_layout.html.erb
  # only for the index action.
  # All other actions will use flights.html.erb
  # or application.html.erb if they exist

  layout "my_special_layout", :only => :index
end
```

or if you choose, specify the layout during an explicit render call:

```
class FlightsController < ApplicationController

  def index
    # render index.html.erb wrapped by
    # app/views/layouts/my_special_layout.html.erb

    render :action => :index, :layout => 'my_special_layout'
  end
end
```

Let's see how we can put layouts to practical use.

Using Layouts for Controller-wide Themes

Layout templates are just like normal action view templates, but they wrap their contents around action view templates. When no specific layout directive is found in the controller code, the controller-specific layout found in the app/views/layouts directory will automatically wrap around every action rendered by that controller.

Inside the layout, you decide exactly where the action template's code is to be inserted by calling yield, as shown in the flights.html.erb layout.

Download view/layouts/flights.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>Flights: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

<p style="color: green"><%= flash[:notice] %></p>
```

```
<%= yield  %>

</body>
</html>
```

We can modify the output rendering of every action from the FlightsController by enhancing this template. Let's add a nice header to our views:

Download view/layouts/flights_enhanced.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
       "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <meta http-equiv="content-type" content="text/html;charset=UTF-8" />
  <title>Flights: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

<h1 id="flights">Flight Administration</h1>

<p style="color: green"><%= flash[:notice] %></p>

<div id="action">
  <%= yield  %>
</div>

</body>
</html>
```

We will now tweak the scaffold.css code a bit:

Download view/layouts/scaffold.css

```
#header { border-bottom: solid 2px green; padding: 20px;}
#action { margin-left: 100px;}
```

To see the layout wrap around every action of the FlightsController, go to http://localhost:3000/flights, and use the scaffold-generated pages. You'll see that all the pages automatically inherit the same heading and margin styles because they are now rendered from within the layout.

Creating a Site-wide Layout

Often we want to enforce a site-wide structural layout. A site-wide layout is implemented by simply naming the template application.html.erb. Rails refers to this file as the *application layout*, and Rails will use it for every controller that does not specify its own layout.

To demonstrate, let's convert our newly made flights layout into a site-wide, or application, layout:

1. Rename flights.html.erb to application.html.erb in the app/views/layouts directory.
2. Edit the title tag as desired, and let's change the `h1` tag as well:

Ruby Download view/layouts/application.html.erb

```
<h1 id="header">Airline Management App</h1>
```

3. Delete the existing controller-specific layouts.

Now, every page will be rendered with our site-wide layout.

Controllers can override the site-wide layout, but you can't have it both ways: actions can be wrapped by their controller's layout or the application layout, but not both.

Creating Partials Instead of User Controls

Websites are often composed of common elements. We have already seen how layouts help promote a common structure without needlessly repeating code inside each view template and how layouts can be seen as a kind of equivalent to .NET master pages. Although layouts are useful for providing overall structure and "wrapper" content, we often want to reuse small components or self-contained sections across many pages. .NET provides this facility with user controls, which help encapsulate appearance and behavior together as a unit. In this section, we will explore *partials*, which is another facility provided by Rails to help us reuse small sections or "components" of a web page in other pages as well.

User Controls in ASP.NET

User controls and server controls are powerful features of ASP.NET. They provide several important benefits to ASP.NET projects that use them:

- They help encapsulate often-used UI and behavior in one place.
- They help factor common functionality from individual pages.
- They can be derived from DataBoundControl to bind the control's UI to the database.
- They can inherit from other user controls, leveraging existing code

Partials in Rails

Partials are reusable bits of template code that can be included in a view. If we look again at the views that were generated for us by the scaffold generator, we see that the new and edit templates are almost identical. The enhancements we made to the new template in the previous section would need to be duplicated in the edit template. Using partials, we can factor out the form code so that both templates can share the same form code. Improvements to the form will then be instantly available to both templates.

DRYing Up Views with Partials

Looking at the form_for block in the new and edit templates, we see that the only difference is the text caption assigned to the submit button. The new template needs to use the text "Create," but the edit template wants to use "Update."

Factoring code out of a template and into a partial follows a standard three-step process:

1. Cut the code out from the view, and paste into a new file in the same directory. This filename can be called anything you want but must begin with an underscore. In Rails, all partial views must begin their filenames with an underscore.
2. Insert code into the view to render the partial where the code used to be. Pass local variable values as needed into the partial.
3. There is no step 3.

In our case, we remove the entire form_for block from the new template into a file named app/views/flights/_form.html.erb. Note the underscore in front of the filename.

We then insert code to render the partial. Our new template code now looks like this:

Download view/partials/new.html.erb

```
<h1>New flight</h1>

<%= error_messages_for :flight %>

<%= render :partial => 'form' %>

<%= link_to 'Back', flights_path %>
```

Immediately we reap a prime benefit of using partials: the template code

all the details of the form, we can now see at a glance the true intent of the new template: to show validation messages, display a form, and let the user navigate back if they need.

Refreshing our browser pointed at http://localhost:3000/flights/new, we see no new apparent difference, which is exactly what we want whenever we refactor. We've improved the inside design of the code without disturbing any of its outward behavior.

Now we can change the edit.html.erb template to also include the partial:

Download view/partials/edit.html.erb

```
<h1>Editing flight</h1>

<%= error_messages_for :flight %>

<%= render :partial => 'form' %>

<%= link_to 'Show', @flight %> |
<%= link_to 'Back', flights_path %>
```

Our editing form now has the new combo boxes and any other enhancements we may have chosen to add. Even better, future enhancements will automatically be reflected in both forms.

However, we do have one problem: the edit form's submit button says "Create" instead of "Update." We want our partial to use "Update" when we are editing an existing row in the database but "Create" when we're creating a new row. We could solve this in many ways:

- In the FlightsController, we could assign an instance variable @submit_caption to be "Create" in the new action and "Edit" in the edit action. The code in our _form.html.erb for the submit button then becomes this:

  ```
  <%= f.submit @submit_caption %>
  ```

  This works because partials have access to the same controller variables as their containing templates.

- We could have the partial interrogate the @flight variable to find out whether it's a new, unsaved model. For example, here we create local variable and initially set the caption to "Edit" but change it to "Create" if necessary:

  ```
  <% caption = "Edit" %>
  <% caption = "Create" if @flight.new_record? %>
  <%= f.submit caption %>
  ```

- Instead of having the partial create its own local variables, we can have the parent template be responsible for supplying them instead. Here's how we could modify the render call in new.html.erb:

  ```
  <%= render :partial => 'form', :locals=>{ :caption => "Create" }%>
  ```

  Conversely, in edit.html.erb we adjust the code to say this:

  ```
  <%= render :partial => 'form', :locals=>{ :caption => "Update" }%>
  ```

  Now we can use this local variable code when we create the submit button in _form.html.erb:

  ```
  <%= f.submit caption %>
  ```

Being able to automate the creation of local variables for the partial helps make the partial more generic and more reusable. Our form has in many ways become our very own kind of "user control" is that we can embed our flight form into any template rendered by the FlightsController. In fact, the render method can do even more for us, as we will see in the next section.

Rendering Collections with Partials

Partials are very useful when want to display items from a collection in a table or grid. Open up index.html.erb, and you'll see code like this:

Download view/partials/index.html.erb

```
<% for flight in @flights %>
  <tr>
    <td><%=h flight.flight_number %></td>
    <td><%=h flight.departs_at %></td>
    <td><%=h flight.arrives_at %></td>
    <td><%=h flight.origin %></td>
    <td><%=h flight.destination %></td>
    <td><%= link_to 'Show', flight %></td>
    <td><%= link_to 'Edit', edit_flight_path(flight) %></td>
    <td><%= link_to 'Destroy', flight, :confirm => 'Are you sure?',
            :method => :delete %></td>
  </tr>
<% end %>
```

We can abstract the details of the table into a partial by moving this code into a separate file. Let us name this file _flight.html.erb (again, remember that all partials must begin with an underscore). Our index.html.erb gets a bit easier to read.

Download view/partials/index_2.html.erb

```
<h1>Listing flights</h1>

<table>
  <tr>
    <th>Flight number</th>
    <th>Departs at</th>
    <th>Arrives at</th>
    <th>Origin</th>
    <th>Destination</th>
  </tr>

<%= render :partial => 'flight' %>

</table>

<br />

<%= link_to 'New flight', new_flight_path %>
```

The for loop has been moved into the partial. In fact, it's so common to have a partial loop over a collection of items that Rails supports this scenario directly and allows us to remove the looping construct from our code. Our new _flight.html.erb code contains just the markup needed to represent one row in the table:

Download view/partials/_flight.html.erb

```
<tr>
  <td><%=h flight.flight_number %></td>
  <td><%=h flight.departs_at %></td>
  <td><%=h flight.arrives_at %></td>
  <td><%=h flight.origin %></td>
  <td><%=h flight.destination %></td>
  <td><%= link_to 'Show', flight %></td>
  <td><%= link_to 'Edit', edit_flight_path(flight) %></td>
  <td><%= link_to 'Destroy', flight, :confirm => 'Are you sure?',
          :method => :delete %></td>
</tr>
```

We need to modify the index.html.erb template and ask Rails to loop over the collection for us:

```
<%= render :partial => 'flight', :collection => @flights %>
```

Using the :collection specifier, render the partial once for each item in the given collection. The question arises, how does the flight variable come to exist inside the partial?

When we use the :collection option, Rails will instantiate a local variable

singular form of the variable name used for the collection. Since we used the variable @flights, Rails will construct a local variable with the singular form—flight—that can be accessed from within the partial.

In fact, if we're willing to follow another naming convention, we can use an even simpler syntax for rendering a collection via a partial. If the filename of the partial is the singular form of the collection, the render method will be able to infer which partial it should use. We named our partial _flight, which is the singular form for @flights, so we're in luck. We can replace our call to render with simply this:

```
<%= render :partial => @flights %>
```

Once again, we can see how embracing Rails conventions enabled us to avoid writing boilerplate code, focusing instead on the more interesting and important aspects of our application.

In this chapter, we took a tour of some of the options we have with our presentation layer in Rails. We saw how the MVC pattern presents a different approach than the forms-based approach we're used to with ASP.NET. We examined how HTML forms work in Rails and how to use layouts and partials to refactor our views to make them cleaner, more flexible, and easier to maintain. Next, we'll continue our exploration of the presentation layer and take a dip into creating robust user interfaces with Ajax.

Chapter 9

Creating Rich User Experiences with Ajax

Ajax is one of many buzzwords used in the web development industry today that loosely defines a collection of different technologies and methodologies for building web applications. In this chapter, we'll define what Ajax is and how we can take advantage of it to build more responsive and visually appealing web applications. Then we'll examine Rails' implementation of the various Ajax technologies and how it differs from the .NET approach, and we'll see how Rails' built-in support for Ajax is one of the things that distinguishes it as a truly powerful modern web framework.

First, a Little Background

It's important for us to fully understand what Ajax is and what choosing to use these technologies buys us. The easiest way to begin would be to take a look at the lifetime of a "traditional" (that is, non-Ajax) web request. A typical scenario goes something like this:

1. The end user types a URL in the web browser or clicks a link on a web page. A request is initiated and sent along the web server.
2. The web server takes the request, processes it, and prepares an HTML document to return in response.
3. The end user's screen flickers a bit, and the resulting HTML document is rendered onscreen.
4. Rinse and repeat.

Before Ajax, this request-response-repeat cycle is what programming for the Web was all about. It makes perfect sense, because HTTP is a *stateless* protocol. That's just a fancy way of saying that each request to the web server has no built-in way to understand state; that is, the HTTP request itself doesn't have any knowledge of what has happened previously. Of course, the modern web application demands that we know what happened before the current request, because we need to do things like manage user account information, add widgets to shopping carts, and send virtual gifts to our friends. So, web frameworks have worked around this limitation by storing session state data in a variety of places, such as in browser cookies, in server memory, in hidden form fields, embedded in the URL, or in a database.

Desktop applications, in contrast, don't have to deal with this limitation. A desktop application does just fine maintaining state between button pushes and screen transitions. As a result, desktop applications tend to seem more responsive and be more satisfying to use.

That is, they are until clever web developers began to realize that, through browser-side scripting, they could make web applications behave a lot like desktop applications in terms of usability. Not entirely, but good enough in most cases. Instead of going through the whole request-screen-flicker-response cycle over and over again, technologies like JavaScript make it possible to replace only parts of a web page instead of the whole thing.

Ajax, by the way, stands for Asynchronous JavaScript And XML. It's "asynchronous" in the sense that the request/response happens outside the realm of the traditional full page GET or POST that we're all used to. JavaScript is the technology that typically powers the ability to do so, and XML is a common data format for the data exchange that happens behind the scenes, although it doesn't necessarily have to be XML. The "XML" part is also applicable because it's the XMLHttpRequest object that allows us to make server-side calls from JavaScript.

The other aspect of using desktop applications that users are really fond of is the support for a richer UI and visual effects. The ability to react to UI events other than hyperlink clicks—such as dragging and dropping, moving the mouse, resizing a window—is also intrinsic to desktop app development. If partial-page updating is one pillar of what Ajax is all about, then the emulation of desktop-like visual effects is the other.

To summarize, we care about Ajax because it makes our web applications' user experience better. Both Rails and .NET have their answers to making partial-page updates and visual effects easy, so let's have a look.

Partial-Page Updates

The idea of partial-page updating is pretty straightforward. The way JavaScript sees an HTML document is known as the *Document Object Model* (DOM), and through the DOM, we can inspect or manipulate any part of the HTML document any way we want. Here is the most basic of examples to illustrate this in action:

Download ajax/random.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>My Great AJAX Application</title>
</head>
<body>
  <p><a href="#" onclick="document.getElementById('random').innerHTML =
    Math.random()">Random number please!</a></p>
  <p id="random"></p>
</body>
</html>
```

Here, we have one link that, when clicked, will replace the contents of a DOM element identified by the ID "random" with a random number. As simple as this example is, this is what partial-page updates are all about. In practice, we may be replacing the contents of the element with something more robust, such as nicely formatted data returned from our database, but the idea is the same.

Let's go a step further. If we take the previous example and combine it with some server-side processing, what we'll get is the quintessential example of an "Ajax-ified" web application. No longer do we have to live with the request-flicker-response cycle—now we have this:

- The end user triggers an Ajax request by clicking an element on a page.
- Without any postback or screen flicker, the server processes the request and returns a response, the elements on the page are updated, and the user gets immediate visual feedback—just like a desktop app.

Of course, with the amount of Ajax that may be going on in a modern web app, doing it all by hand like this is going to get old pretty quickly. That's why both Rails and .NET provide client-side libraries to help get it done a lot easier.

The .NET Ajax Client Library

ASP.NET Ajax, which is an add-on to ASP.NET 2.0 and included in ASP.NET 3.5, is a rich set of extensions to the core .NET libraries created to help us implement Ajax functionality in our applications. Its other goal is to help us develop client-side code in the languages we're used to, such as C# or VB. The idea, as we've seen in other parts of this book, is to reduce the amount of context switching we have do, making us more productive.

At the heart of the .NET Ajax library is the UpdatePanel control. The UpdatePanel defines a region of the page that will be replaced when certain events occur. Here is a rudimentary example—we're going to create a page that will display a totally random flight status when a link is clicked:

Download ajax/RandomFlightStatus.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeFile="Default.aspx.cs"
  Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Completely Random Flight Status Page</title>
    <script runat="server" language="C#">
      protected override void OnLoad(EventArgs e)
      {
        base.OnLoad(e);
        if (Page.IsPostBack)
        {
          string[] statuses = { "on-time", "delayed", "cancelled" };
          status.Text = statuses[new Random().Next(statuses.Length)];
        }
      }
    </script>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ScriptManager ID="ScriptManager1" runat="server">
        </asp:ScriptManager>
```

```
        <p><asp:LinkButton ID="LinkButton1" runat="server"
          onclick="LinkButton1_Click">What's the status of my flight?
        </asp:LinkButton></p>

        <asp:UpdatePanel ID="UpdatePanel1" runat="server"
          UpdateMode="Conditional">
          <ContentTemplate>
            <p><asp:Label runat="server" ID="status"></asp:Label></p>
          </ContentTemplate>
          <Triggers>
            <asp:AsyncPostBackTrigger ControlID="LinkButton1"
              EventName="Click" />
          </Triggers>
        </asp:UpdatePanel>
    </div>
    </form>
</body>
</html>
```

Breaking it down a bit, we have several controls in place that make this all work:

- A ScriptManager, which is required on any web form that uses Ajax.
- A LinkButton, which is the link we'll click to display the flight status.
- An UpdatePanel, which is the area to be updated when the link is clicked.
- Within the UpdatePanel, we define a ContentTemplate, which contains the content that will be updated and displayed when the update event occurs. Here, the ContentTemplate contains a single Label control—status.
- Also within the UpdatePanel, we have a AsyncPostBackTrigger, which tells us the control and event name that drives the updating of the UpdatePanel, in this case, the click event of LinkButton1.

We also have some C# code in the head of the document, which defines an OnLoad event. This code will fire on *postback* and will update the text of the status Label control with our totally random value.

This solution scales nicely from a developer's point of view. As we add more content that we want to update, we simply add more controls to the ContentTemplate. If we want to retrieve data from a database, we simply modify the OnLoad method to give us the data we want. And all this code is written in C# and ASP.NET, not JavaScript.

The Rails Way

In Rails, a couple of different technologies are at work that make the job of partial-page updating a lot easier. The first is the Prototype JavaScript framework, and the other is RJS templates.

The Prototype JavaScript Framework

Prototype[1] was created by Sam Stephenson, who has also been a member of the Rails core team. As with the .NET Ajax library, the main goals of Prototype are to make JavaScript less like, well, JavaScript and to make programming with JavaScript more aligned with the object-oriented patterns with which we're already comfortable. Prototype is basically a set of helper methods that extends JavaScript, making common tasks easier and makes it more, dare we say, Ruby-like.

Although it is built in to the core Rails framework, Prototype works great on its own and alongside many other web programming frameworks, including ASP.NET. Along the same lines, it is certainly not the only JavaScript framework that Rails developers can use in building their applications; libraries like jQuery, [2] MooTools,[3] and Dojo[4] are perfectly acceptable alternatives to Prototype if you are comfortable using them already.

Here is the random number example again, this time with a Prototype helper method along for the ride:

Download ajax/random_with_prototype.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>My Great AJAX Application</title>
  <script src="prototype.js" type="text/javascript"></script>
</head>
<body>
  <p><a href="#" onclick="$('random').innerHTML=Math.random()">
    Random number please!</a></p>
  <p id="random"></p>
</body>
</html>
```

The difference here is subtle but important. Instead of document.getElementById() as in the pure JavaScript example, we've replaced it

1. http://prototypejs.org/
2. http://jquery.com/

with the $ function, which essentially does the same thing. If we're going to be doing a lot of partial-page updates, it certainly saves a lot of typing! But it still feels like a bit of a hack. One of the big selling points of Prototype is that it takes hackish JavaScript DOM manipulation code and lays a nice blanket of abstraction on top, making it seem a lot more object-oriented. Here is yet another way of writing the same simple code:

Download ajax/random_with_prototype_2.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>DOM Element Updating With Prototype</title>
  <script src="prototype.js" type="text/javascript"></script>
</head>
<body>
  <p><a href="#" onclick="Element.update('random', Math.random())">
    Random number please!</a></p>
  <p id="random"></p>
</body>
</html>
```

Now, we've replaced the $ function with a call to the Element.update function, passing the DOM ID of the element we want to update, along with the new value. Nice and clean. Now, for our last bit of JavaScript refactoring, let's go crazy by removing the inline JavaScript altogether:

Download ajax/random_unobtrusive.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>DOM Element Updating With Prototype</title>
  <script src="prototype.js" type="text/javascript"></script>
  <script type="text/javascript" charset="utf-8">
    function generateRandomNumber() {
      Element.update('random', Math.random());
    }
    Event.observe(window, 'load', function() {
      Event.observe('generate', 'click', generateRandomNumber);
    });
  </script>
</head>
<body>
  <p><a href="#" id="generate">Random number please!</a></p>
  <p id="random"></p>
</body>
```

Event.observe is a Prototype function that creates an observer—that is, a bit of code that waits for something to happen in the DOM. Here, we've told the browser to wait for the content of the page to be fully loaded and then to keep an eye out for click events on the DOM element with the ID generate. And, when that event is fired, run another JavaScript function, generateRandomNumber(), which does the same DOM element replacement we did previously.

"Unobtrusive JavaScript" zealots would propose that this is the proper way to do web programming. They would argue that HTML markup should describe a document's structure, not the programmatic behavior. With this technique, the presentation (the markup) and the functionality (the JavaScript) are completely separated, and using Prototype makes it easy.

Talking with Server Data with Prototype

In a real-life scenario, we're probably going to be using Ajax to get data from a web server and display it to the user, not just executing a simple client-side function. Prototype gives us a few built-in ways to do this, the most basic being the Ajax.Request method. The Ajax.Request method handles the lifetime of an Ajax request from the browser and provides callbacks to help us evaluate the response. Here is a look at how Ajax.Request is typically used:

```
new Ajax.Request('/path/to/return/some/data', {
  method: 'post',
  onSuccess: function(transport) {
    alert(transport.responseText);
  }
});
```

The Ajax.Request method takes two parameters, a URL and a JavaScript "hash" of options. At a minimum, this hash typically includes an onSuccess callback, like we're doing here, which indicates what we'd like to happen when the Ajax request returns successfully. In this case, we're simply displaying a dialog box with the contents of the raw response.

However, if we don't specify any callbacks to react to the response of the Ajax call, Prototype will simply *eval* (execute) any JavaScript code that is returned. This is the basic idea behind RJS.

Rendering JavaScript with RJS Templates

Now that we've seen how powerful the Prototype library can be, let's look at RJS. As we've seen with other components of Rails, one of the

main goals of the framework is to have as much of our application written in Ruby as possible. Yes, even the JavaScript. RJS takes some of the most common tasks we might normally perform with pure JavaScript and gives us a Ruby layer on top of it.

In theory and spirit, RJS in Rails is not all that different from .NET Ajax—they both stem from the desire to at least try to do all our development activities in a single language—but as we'll see, it is quite different in style.

Let's now take the same functionality from our random flight status ASP .NET example and write it in Rails.

First off, we need to build a back end for our flight status randomizer. We'll use a StatusesController to do that:

```
c:\dev\flight> script/generate controller statuses
```

Since this is a RESTful controller, we'll also need to add the appropriate route to our routes file:

```
map.resources :statuses
```

Now, here's the controller code:

Download ajax/statuses_controller.rb

```
class StatusesController < ApplicationController

  def create
    statuses = %w(on-time delayed cancelled)
    @status = statuses[rand(statuses.size)]
    respond_to do |wants|
      wants.js
    end
  end

end
```

Our controller has a single action, create, which retrieves a random status message from an array of messages. The @status instance variable is then set with that value. Just like our normal HTML/ERb views, any instance variable that is set in the controller is also available in the view. The respond_to block indicates that we want this method to respond to and return a JavaScript response. We need the respond_to block because, by default, this action will look for and render a template named create.html.erb if not specifically told to do otherwise. This respond_to block says to instead look for an RJS template; and by convention, the template is named create.js.rjs.

Download ajax/create.js.rjs

```
page.replace_html :random, @status
```

Since this RJS template is only a one-liner, some folks like to skip the separate file entirely and write the controller like this instead:

Download ajax/statuses_controller_render_update.rb

```
class StatusesController < ApplicationController

  def create
    statuses = %w(on-time delayed cancelled)
    @status = statuses[rand(statuses.size)]
    render :update do |page|
      page.replace_html :random, @status
    end
  end

end
```

In general, short bits of RJS like this one can usually be written inline in the controller code, whereas more complex UI functionality should be dished off to a separate RJS file. It's up to us to decide what feels right in terms of style and code readability. Regardless of which way it's written, what we're doing here is telling the controller to return a JavaScript-typed response to the browser. The browser then executes the JavaScript code, which in this case replaces the contents of the DOM element with ID random with the contents of the instance variable @status.

The last thing we'll need to do is to create a page that calls this action and displays our flight status. We'll call it the index action:

Download ajax/index.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>Totally Random Flight Status Generator</title>
  <%= javascript_include_tag :defaults %>
</head>
<body>
  <p><%= link_to_remote "What's my flight status?",
                        :url => statuses_url %></p>
  <p id="random"></p>
</body>
</html>
```

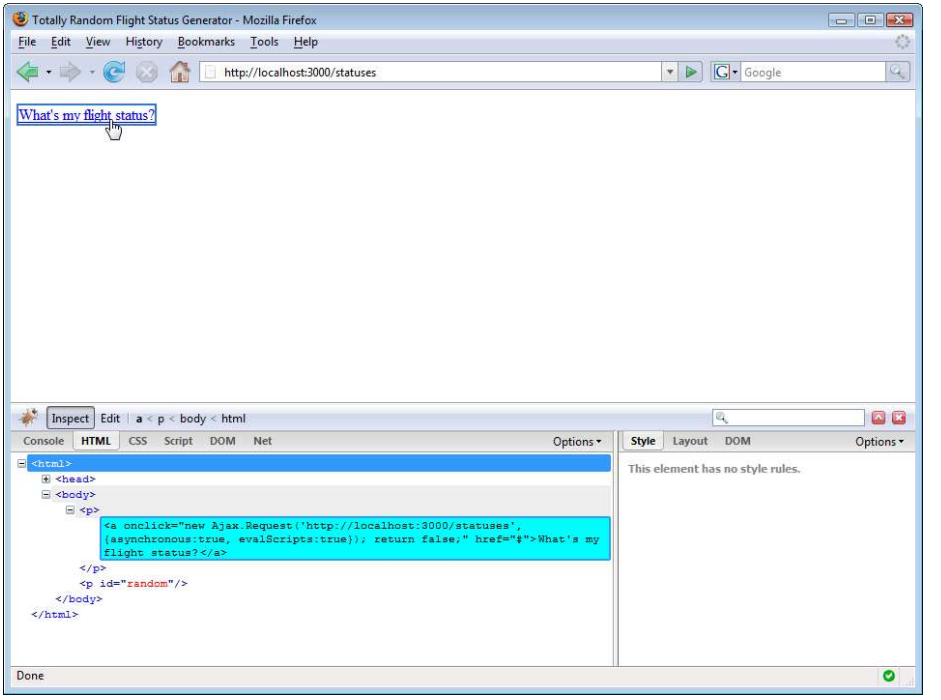


Figure 9.1: Using Firebug to inspect the DOM

Looks familiar. It's just like our random number example with a couple of notable exceptions. First, we're using the link_to_remote helper rather than the link_to helper that we'd use for a non-Ajax hyperlink. And we're also including the Prototype library with a javascript_include_tag in the head of the document.

To really see what's going on behind the scenes, we can use Firefox along with Joe Hewitt's outstanding Firebug[5] extension to inspect the source of the link we're clicking and see the JavaScript response that is returned, as shown in Figure 9.1.

The Rails code that we wrote looks like this:

```
<%= link_to_remote "What's my flight status?",
                   :url => statuses_url %>
```

5. http://getfirebug.com/

Joe Asks...

But This JavaScript Isn't Unobtrusive, Is It?

That's right. The default Rails implementation of RJS and the Prototype helpers don't generate unobtrusive JavaScript; it renders it all inline. This has been a point of contention for many developers in the Rails community; if unobtrusive JavaScript is important for your web app, it's highly recommended you check out the Unobtrusive JavaScript for Rails plug-in,* built on top of Dan Webb's excellent LowPro† extensions for Prototype.

*. http://www.ujs4rails.com/
†. http://www.danwebb.net/2006/9/3/low-pro-unobtrusive-scripting-for-prototype

Using Firebug, this is actually rendered as this markup:

```
<a onclick="new Ajax.Request('http://localhost:3000/statuses',
  {asynchronous:true, evalScripts:true}); return false;" href="#">
  What's my flight status?</a>
```

On the onclick event of this hyperlink, we are using Prototype's Ajax.Request to make an Ajax request to the /statuses URL. And since we have no callbacks in place (such as onSuccess or onFailure), the resulting JavaScript code is simply going to be *eval'd*.

Again, using Firebug, we can see what that JavaScript code is by inspecting the XMLHttpRequest's response when we click the link. So, the code that we wrote:

```
page.replace_html :random, @status
```

is rendered into this JavaScript code at runtime:

```
Element.update("random", "cancelled");
```

This code will be executed and will replace the DOM element random with our desired value when the response is successfully returned. Just to illustrate a point, we could have written the RJS like this:

```
page << "Element.update('random', '#{@status}');"
```

Using the << method sends arbitrary JavaScript to the page, so this would yield identical results. Doing it this way, of course, is more JavaScript than Ruby, so we don't want to do this. But in some cases,

RJS doesn't do the job, and we have to write our own JavaScript code and inject it using the << method.

In short, to really be a great Ajax applications developer, you have to learn JavaScript. But Prototype along with RJS makes it fairly painless to perform common tasks and gets us most of the way there without having to be a hardcore JavaScript programmer.

Now that we've looked at the basics of JavaScript in Rails vs. ASP.NET, let's move on to some visual effects.

Visual Effects on the Web

One of the things that has traditionally been missing from web user interfaces is any kind of rich, desktop-like visual response to input from the end user. Since the web was created simply to be a collection of essentially static hypertext documents, it's been a long road getting web user interfaces on par with what we're used to with client applications; computer users have long taken for granted features such as animated screen transitions, drag and drop, and autocompletion in desktop applications.

For the purposes of this discussion, let's disregard plug-in technologies like Flash and Silverlight and say that the only current way to support rich visual interfaces on the Web is by using the power of the web browser and JavaScript to, again, manipulate the DOM in such a way that the end user finds it visually appealing and responsive. And again, the average web developer would probably find that writing these types of visual effects from scratch using raw JavaScript rather painful. Luckily, both ASP.NET Ajax and Rails provide comprehensive frameworks for generating the JavaScript necessary to make it all happen.

Providing User Feedback

Back in the days when web page design was very simple, there was very little need to use Ajax for visual feedback purposes. The end user would submit a form or click a link, and the speed of the response that would come back from a full HTTP request was probably good enough to satisfy most people. But today, higher-bandwidth connections bring higher expectations; combine that with the face that modern web applications are usually quite heavy with images and other content with larger file sizes, and now we do have a need for a way to provide more immediate feedback.

To illustrate some very simple visual effects, we'll extend our random flight status example just a little more. When our end user clicks the link to ask for the flight status, what we'd like to do is give the resulting status a little more visual flair. Instead of simply replacing the contents of the DOM element with the new value, as we've done, we're going to make the DOM element "fade in," that is, begin with a completely invisible element and make it visible by gradually increasing its opacity.

How You Might Approach It in .NET

To perform animations and other rich visual effects in ASP.NET, we'll need the ASP.NET Ajax Control Toolkit.[6] Once we reference the binaries for the Ajax Control Toolkit in our project, we'll get some additional controls for building ASP.NET Ajax applications, on top of what is already provided with the core ASP.NET Ajax framework.

Updating an element and displaying a visual effect to the end user is a pretty common task in the modern Ajax web interface. So, the Ajax Control Toolkit provides the UpdatePanelAnimationExtender to help us out. Here is our ASP.NET random flight status example again, this time with fade-in effect:

Download ajax/RandomStatusWithAnimation.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"
    CodeFile="Default.aspx.cs"
    Inherits="_Default" %>
<%@ Register
    Assembly="AjaxControlToolkit"
    Namespace="AjaxControlToolkit"
    TagPrefix="ajaxToolkit" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Completely Random Flight Status Page</title>
    <script runat="server" language="C#">
      protected override void OnLoad(EventArgs e)
      {
        base.OnLoad(e);
        if (Page.IsPostBack)
        {
          string[] statuses = { "on-time", "delayed", "cancelled" };
          status.Text = statuses[new Random().Next(statuses.Length)];
        }
      }
    </script>
</head>
```

```
<body>
    <form id="form1" runat="server">
    <div>
      <asp:ScriptManager ID="ScriptManager1" runat="server">
      </asp:ScriptManager>

      <p><asp:LinkButton ID="LinkButton1" runat="server"
        onclick="LinkButton1_Click">What's the status of my flight?
        </asp:LinkButton></p>

      <asp:UpdatePanel ID="UpdatePanel1" runat="server"
        UpdateMode="Conditional">
        <ContentTemplate>
          <p style="background:#ff7f00">
            <asp:Label runat="server" ID="status"></asp:Label></p>
        </ContentTemplate>
        <Triggers>
          <asp:AsyncPostBackTrigger ControlID="LinkButton1"
                EventName="Click" />
        </Triggers>
      </asp:UpdatePanel>

      <ajaxToolkit:UpdatePanelAnimationExtender id="animationExtender"
        TargetControlID="UpdatePanel1" runat="server">
        <Animations>
          <OnUpdated>
            <Sequence duration="0.25">
              <FadeIn AnimationTarget="UpdatePanel1" />
            </Sequence>
          </OnUpdated>
        </Animations>
      </ajaxToolkit:UpdatePanelAnimationExtender>
    </div>
  </form>
</body>
</html>
```

Let's highlight the differences between this and our sample without animation:

- We've imported the AjaxControlToolkit assembly at the top of the document so we can use the controls that are included with it.
- We've added an UpdatePanelAnimationExtender control. This allows us to declaratively specify which control is to be animated, which events will trigger the animations, and which animations we want to display. In this case, we want to animate UpdatePanel1 with a FadeIn effect that lasts for 0.25 seconds whenever the panel is updated.

I Need My Controls

Web development with ASP.NET has always followed in the spirit of control-based desktop development, and ASP.NET Ajax is no exception. With the Ajax Control Toolkit, these are just a handful of the built-in pieces of functionality you get just by dragging and dropping controls onto your web form:

- Accordions
- Calendars
- Drop shadows
- Rounded corners
- Sliders
- Slideshows

The short answer to whether these types of controls are available by default in Rails is no. But that doesn't mean we can't include them in our application. It just means that we may have to utilize one of the many other JavaScript control libraries available, such as the ones included in Dojo or the Yahoo User Interface (YUI) library, look for an open source solution, or at worst write our own JavaScript. Thankfully, the open source JavaScript community is quite strong, and the Rails community does a fantastic job of finding the best open source JavaScript solutions and converting them into Rails plug-ins for all to use.

- For added effect, we've tacked on an inline style to the content that's displayed, giving it an orange background color. This will make the fade-in effect a lot easier to see.

The Rails Way

Along with Prototype, Rails ships with another JavaScript library, Scriptaculous,[7] which builds on top of Prototype to provide user interface-specific functionality such as animation, drag and drop, in-place editing, and other utility classes. And, just like it does with Prototype, Rails wraps most of Scriptaculous with helper methods in order to keep it all in Ruby.

7. http://script.aculo.us/

Here is the Rails version of the random flight status interface, complete with fade-in effect:

Download ajax/index_with_effects.html.erb

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
  <title>Totally Random Flight Status Generator</title>
  <%= javascript_include_tag :defaults %>
</head>
<body>
  <p><%= link_to_remote "What's my flight status?",
    :url => statuses_url, :loading => "Element.hide('random')",
    :complete => visual_effect(:appear, :random) %></p>
  <p id="random" style="background:#ff7f00"></p>
</body>
</html>
```

There's very little difference between this example and the one without animation; and, we've done this without any changes to the back-end controller or RJS code. Thanks to Prototype, the link_to_remote method is able to provide a few callbacks that we can take advantage of to display visual effects. The loading callback is called when we make the Ajax request. We could use this to display a progress graphic; here, we're simply hiding the DOM element random. The complete callback then calls the helper method visual_effect, which is simply a wrapper around Scriptaculous methods that display certain effects of the animation framework. In this case, we're asking for the appear effect, which makes the element random fade in.

Many other effects are available, all of which are well-documented on the Scriptaculous website. Switching from a "fade-in" to a "blind-down" (where the element appears from top to bottom) is as simple as changing our link_to_remote call:

```
<%= link_to_remote "What's my flight status?", :url => statuses_url,
    :loading => "Element.hide('random')",
    :complete => visual_effect(:blind_down, :random) %>
```

All we have to do is replace the name of the first parameter to the visual_effect method. Feeling crazy and want to make the element move from side to side when updated?

```
<%= link_to_remote "What's my flight status?", :url => statuses_url,
  :complete => visual_effect(:shake, :random) %>
```

We've removed the loading callback, since we're not making the element disappear and appear again, and simply changed the effect name to shake. It's that simple.

In this chapter, we've gone headfirst into discovering Ajax web development. We've discussed what Ajax is and why it's important in creating truly modern web applications. And, we've taken a look at two mainstays of Ajax development—partial-page updating and rich visual effects—and the fundamental differences between the implementation of these techniques in .NET and Rails. We have seen how Rails uses the Prototype JavaScript framework and the Scriptaculous extensions, while ASP.NET uses the ASP.NET Ajax client library, and we have seen how the philosophical differences between the two frameworks are reflected in the implementation of Ajax as well.

Congratulations, you've reached the end of our coverage of basic Ruby on Rails. Combined with the software development experience you already had working with .NET, you're now armed with some new knowledge on—and hopefully a new perspective of—developing web applications the *opinionated* way with Ruby on Rails. Now we'll tackle some more advanced topics, including test-driven development, integration with .NET, and fine-tuning our Rails development environment.

Part III

Advanced Topics

Chapter 10

Test-Driven Development on Rails

Test-driven development (TDD) is the agile practice of writing unit tests as you develop your application. Writing and maintaining a suite of unit tests is recognized as one of the hallmarks of quality application development. Rails not only supports the writing of unit tests, but it also actively encourages it. In this chapter, we'll first learn how write tests for preexisting Rails code to demonstrate how you can begin to develop an automated test suite for your applications. Then, we'll step into TDD, writing tests before we write code. Finally, we'll take a look at Shoulda, one of the many "specification" frameworks that provides an alternative to the standard Ruby unit testing syntax.

If you've never done TDD before, writing tests before there's any code to test might sound silly. To the contrary, writing tests first generally leads to simpler and cleaner application code. Here are a few of the reasons TDD developers typically cite when asked why they prefer to write tests before they write any code:

- Writing a unit test for a small piece of functionality is a great way to capture your requirements, so you know what to build.
- Writing a test before you write the application code forces you to design the public interfaces of your classes from the client code's point of view.
- Writing tests first generates a wonderful by-product, which is an automated test suite for your application.

Software written without unit tests is fragile software. It can be frightening to change code in a large system when you're worried that you might break something and not even know it. Ugly code that needs cleaning will tend to be ignored.

Bad code will get worse with each subsequent hack and workaround, until finally one day you just have to throw everything out so you can start over with a clean slate.

Before we can dive headlong into TDD, we will start by first learning how Ruby helps us write executable test suites, whether we write them first (which we will do in Section 10.2, *Test-Driven Development with Test/Unit*, on page 195) or last.

A First Look at Test/Unit

Ruby comes with a built-in testing framework, called Test/Unit, named after the require statement used by Ruby programmers to enable the unit testing framework inside their projects. A well-written test is generally composed of three parts:

- Some test data to play with is created. This data could be anything, from simple local variables to data from a test database filled with test data.
- The class or object to be tested is put through its paces.
- Assertions are made to ensure that the object's final state matches some expected state.

Let's first look at Ruby example that does not involve any Rails code at all. Take a look at this Car class:

Download tdd/car.rb

```
class Car

  MILES_PER_GALLON = 20

  # Represent the fuel tank of our car
  attr_reader :fuel

  def initialize
    @fuel = 0    # start with an empty tank
  end

  # Add some fuel to the tank
  def add_fuel(amount)
    @fuel += amount if amount > 0
  end

  # How far can we go?
  def range
   0
  end
```

Our car gets twenty miles to the gallon. We can add fuel to the tank by calling the add_fuel method. Once the car has some fuel, we can find out how far we can go by calling the range method.

To find out whether the range method works correctly, we could choose to write a client program that uses our Car class:

Download tdd/use_car.rb

```
require 'car'

# Create a car object
my_car = Car.new

# Add 10 gallons to the tank
my_car.add_fuel(10)

# Display the range of the car
# We expect to see "20"
puts "Range is: #{my_car.range}"
```

We reference our Car file on line 1 so that Ruby will know about the Car class. We proceed to create a Car object, fill it with ten gallons of fuel, and then display the car's range. We can run this program from our command prompt like this:

```
c:\dev> ruby use_car.rb
Range is: 0
```

We got 0 instead of the expected value of 200. There's a bug in our code somewhere. Ah, yes, we need to correctly implement the range method like this:

```
def range
 @fuel * MILES_PER_GALLON
end
```

Let's run the client code again and see what we get now:

```
c:\dev> ruby use_car.rb
Range is: 200
```

We've fixed the bug, but let's not celebrate too loudly. To find the bug in the first place, we had to manually run the program and inspect the results. In a large program—say, a web application—it may not be possible to manually exercise every feature of the application to make sure everything works. We'd also have to manually test the entire application before every release to make sure that changes to one class haven't adversely affected the expectations or behavior of any other class.

write a separate client test program or requiring us to manually verify the results. Here's our testing code:

Download tdd/car_test.rb

```
require 'test/unit'
require 'car'

class CarTest < Test::Unit::TestCase

  def test_range
    # Prepare a car for testing
    car = Car.new

    # Do something interesting with the car
    car.add_fuel 10

    # Make sure the range is calculated correctly
    assert_equal(200, car.range)
  end

end
```

We start by requiring the test/unit library as well as our Car class file. We then declare a class that derives from Test::Unit::TestCase, which is defined for us by the test/unit library.

We define our test on line 6. After creating a Car object to test with, we add some fuel on line 11. We then *assert* that the car's range now equals 200.

Now that our test is written, how do we run it? You might expect that we need to write some more code to instantiate our CarTest class and call the test_range() method. Fortunately, we don't have to do any of that work.

Our TestCase-derived class is a normal Ruby class but with one important difference. When we run our file through the Ruby interpreter, any methods that begin with test_ will automatically be executed after all the code is loaded.

Let's see what happens when we simply run our test code:

```
c:\dev> ruby car_test.rb
Loaded suite car_test
Started
.
Finished in 0.000237 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

Directory	What It's Used For
unit	Unit (model) tests
functional	Functional (single controller) tests
integration	Integration (multiple controllers) tests
fixtures	Hardcoded fixture data (.yml files)
mocks	Mock/stub classes used in development and test modes

Figure 10.1: Test subdirectories

Presto! Our test method was automatically detected and executed for us. The results show that we wrote one test method, which contained one assertion, and it passed with flying colors.

As we add features to our Car class, we could write more tests. And at any time, we can run our tests to make sure that our code still behaves as it should, no matter how large our code gets. Having an automated suite of tests is essential to writing any application, and we wouldn't think of writing a real-world Rails application without it.

It will come as no surprise that Rails embraces Ruby's built-in testing. Let's see how Rails encourages us to write unit tests for our web applications.

Test-Driven Development with Test/Unit

Let's start a new Rails application to demonstrate how to write unit tests for web applications. We will write a very simple application that lets a passenger make a flight reservation. This time around, we will start by writing the tests first.

Start by generating a new Rails application:

```
dev$ rails reservations
```

Every Rails application includes a test directory, which is where all our test classes will go. The test directory contains several subdirectories, as shown in Figure 10.1.

In this chapter, we'll be focusing on Rails unit tests and fixtures, which are the core elements of all Rails testing.

Writing Our First Test

Our application will allow a passenger to book a reservation on a flight. We've already learned a lot about creating models, views, and controllers in Rails, so we're going to keep things simple and concentrate on the testing facilities that are available to us as we develop Rails applications.

Here are the requirements we aim to achieve in our simple application:

- Passengers have a name and optionally an email address and frequent flyer number.
- We need to be able to reserve a seat on a flight for on behalf of a passenger.
- We need to cancel a reservation for a given passenger.

Let's generate our passenger and flight models:

```
reservations$ script/generate model flight number:string
                origin:string destination:string
reservations$ script/generate model passenger name:string email:string
                freq_flyer:string flight_id:integer
reservations$ rake db:create:all
reservations$ rake db:migrate
```

Whenever we generate a model, Rails automatically starts a skeleton test class for it. Model tests are called *unit tests* in Rails. Open up test/unit/passenger_test.rb, and you'll see what Rails generated for us:

Download tdd/reservations/passenger_test_1.rb

```
require File.dirname(__FILE__) + '/../test_helper'

class PassengerTest < ActiveSupport::TestCase
  # Replace this with your real tests.
  def test_truth
    assert true
  end
end
```

Rails creates a placeholder test, called test_truth, that's on line 5. Rename it, and create our first test—that passengers must have a name to be considered valid records in our database.

Download tdd/reservations/passenger_test.rb

```
def test_passenger_invalid_if_name_is_missing
  # Create a passenger without a name
  p = Passenger.new :email => 'someone@example.com'

  # Make sure the passenger object is not valid
```

Joe Asks...

What About Controller and View Tests?

In this chapter, we're exploring model tests, or what Rails refers to as *unit* tests. Rails also supports automated testing of controllers and views. For more information on how to write "functional" (single-controller) tests and "integration" (application-wide) tests, we recommend *Agile Web Development with Rails* (RTH08).

Our test code is pretty simple. On line 3, we create a new Passenger object with only an email address. On line 6, we use the built-in assert_equal method to ensure that the passenger is not valid.

Now all we have to do is run the test. Recalling our Car example earlier, we may be tempted to just run the passenger_test.rb script directly. However, that won't actually work in all circumstances. To understand why, we need to take a closer look at the meaning of the "test environment" in Rails.

The Rails Test Environment

Back in Section 4.2, *Environments in Rails*, on page 81, we learned about the different kinds of Rails environments we can define. In Section 4.3, *Configuring Data Access*, on page 83, we looked closely at the database.yml file and saw how it is used to map each Rails environment to a particular database. In development mode, the environment variable RAILS_ENV is given the value development, and the corresponding database configuration is read from the database.yml file. When we start up our local web server and use our browser to play with our application, it's the development database that's being used.

But something very different happens whenever we run our unit tests:

- The RAILS_ENV variable is set to test.
- All our model classes that derive from ActiveRecord::Base will point at the test database instead of the development database.
- All log output will be directed to log/test.log, instead of log/development.log.
- Any environment settings in config/environments/test.rb will be exe-

New Test Syntax Coming Soon?

As this book was being completed, the Rails core team had begun to introduce an alternative syntax for defining test methods. This new syntax will not be required but will be an option for those who prefer it. The new approach eschews Ruby method definitions for Ruby blocks instead:

```
test "test_passenger_invalid_if_name_is_missing" do
   # Normal test/unit code and assertions go here
end
```

Check the latest Rails documentation for the syntax that's actually available in your version of Rails.

This means that as we run our tests, new rows of data will be inserted into the test database, not the development database. This is very good news, because it means our tests can run in total isolation from our development environment.

However, it also means that before we can run any tests, we need to ensure that our test database has the same schema as our development environment. Any migrations that we've applied to our development database (by running rake db:migrate) need to first be applied to our test database, too.

In addition, we may want to prepopulate our test database with a lot of test data that's useful for our tests to use. The rows of data we provide in our test database are called *fixtures*, and we'll learn more about how to use fixtures in Section 10.4, *Providing Test Data with Fixtures*, on page 206. The important thing to know for now is that we need to ensure that the test database is reset to a known, good state before each test is run.

Lastly, Rails projects can consist of many models, each with their own accompanying test classes. Running each one by hand would become tedious, and we'd have to be careful that we run every test class every time we want to run the full suite of tests.

Since we can't dependably run a test without first ensuring that all these dependencies are taken into consideration first, we generally do not run a test class directly from the command line. Instead, we use the rake command to run our tests, as we'll learn about next.

Running Tests with rake

Rails believes that you should be writing tests and executing them frequently as you develop your application. Running your tests should be easy, leaving no room for excuses not to write and run them. It may come as a pleasant surprise (if you're already practicing test-driven development) or a bit of a shock (if you aren't) that, in the absence of any other instructions, rake will run all your tests.

We'll be talking about rake in more detail in Section 12.3, *Learning More About rake*, on page 240. For now, all we need to know is that a number of things will happen automatically for us:

- The RAILS_ENV variable is set to test.
- The test database's table schema is made to match the development database schema.
- Any test data fixtures are loaded into the database.
- All test files in the test/unit, test/functional, and test/integration directories are detected and executed.

Here's what happens when we call rake for the first time (we've removed some clutter for clarity's sake):

```
reservations$ rake
Loaded suite /../rake/rake_test_loader
Started
.F
Finished in 0.10178 seconds.

1) Failure:
test_passenger_invalid_if_name_is_missing(PassengerTest)
  [./test/unit/passenger_test.rb:10:in
       `test_passenger_invalid_if_name_is_missing'

  <false> expected but was
<true>.

2 tests, 2 assertions, 1 failures, 0 errors
Errors running test:units!
```

Whoa. That's a lot of output. Rails ran all of our tests, including the placeholder test that was created for us in flight_test.rb. The important thing to notice is that our test failed:

```
1) Failure:
test_passenger_invalid_if_name_is_missing(PassengerTest)
  [./test/unit/passenger_test.rb:10:in
       `test_passenger_invalid_if_name_is_missing'
```

This is good news! Our test fails because we haven't written any code yet that would fulfill this requirement. Now let's make this test pass, shall we?

Making the First Test Pass

Now comes the fun part: making the test pass. How do we do that?

Whenever we seek to fix a failing test, we must say to ourselves: what's the simplest thing we could possible do in our application to make the test pass?

Let's open app/models/passenger.rb and add a simple validation rule for the name attribute:

Download tdd/reservations/passenger.rb

```
class Passenger < ActiveRecord::Base

  validates_presence_of :name

end
```

Now run our test suite again:

```
reservations$ rake
Loaded suite /../rake/rake_test_loader
Started
..
Finished in 0.059897 seconds.

2 tests, 2 assertions, 0 failures, 0 errors
```

Even though it's only one test, we're actually on the road toward building a maintainable, automated test suite for our application. Before we move on, let's quickly review the steps we took to implement a test-driven feature:

1. We started by writing a test that will prove that our requirement has been fulfilled.
2. We gave our test a descriptive name and made sure to start the test method with test_.
3. We ran our tests with the rake command. We watched it fail before we wrote any application code. In this small project, this could be viewed as overkill. In larger applications, it's often critical: you want to make sure the test you wrote can clearly show that the desired requirement has not yet been implemented so that a subsequent pass is proof that you've properly implemented the new

4. We wrote the simplest application code we could that would make the test pass.
5. We ran rake to confirm that all the tests pass.

If you have that feel-good feeling by seeing our first test pass, pat yourself on the back: you're catching the idea of TDD. We're ready to finish off our application.

Making a Reservation

Now that we have a passenger, let's create a reservation. A *reservation* is an association between a passenger and a flight. Let's see how we might write our next test:

Download tdd/reservations/passenger_test.rb

```
def test_make_reservation
  # Create a passenger
  passenger = Passenger.new :name => 'John Smith'

  # Create a flight
  flight = Flight.new :number => '321',
                      :origin => 'ORD',
                      :destination => 'JFK'

  # Make reservation
  flight.reserve(passenger)

  # Make sure reservation exists
  assert flight.passengers.include?(passenger)
end
```

To make a reservation, we first create a passenger object on line 3, as well as a flight object on line 6. We decide that it would be neat to make a reservation by calling a reserve method on our flight object, so that's exactly what we do. We then check to make sure the flight has our reservation by making sure our passenger is in the list of passengers booked for the flight.

If you're thinking, "Wait a minute, we haven't written all those other methods yet," you're exactly right. What we've done is specified more of the design of our application. We just made up some method names that made sense for the situation. This is one of the great advantages of test-driven development. We design public interfaces by writing tests, making the test the first-ever client of our application code.

Without TDD, we probably would've begun by adding a bunch of methods to the Flight class. Next, we would have tried to write some appli-

were right, but maybe they'd be found to be wrong. Test-first development turns the tables, enabling us to think of our system from the client code's perspective first. We add methods to our classes only when necessary and with names that make the most sense for the code that will actually use them.

Let's rake our code again. You'll get something like this amidst the rake output:

```
 1) Error:
test_make_reservation(PassengerTest):
NoMethodError: undefined method `reserve' for #<Flight:0x6cef30>
```

We're following the classic TDD pattern: write a test, watch it fail, write the implementation code, and watch it pass. We're halfway there, so let's now write some implementation code. Here's what we need to add to the Flight class:

Download tdd/reservations/flight.rb

```
class Flight < ActiveRecord::Base

  has_many :passengers

  def reserve(passenger)
    passengers << passenger
  end

end
```

Let's rake again, and now all of our tests pass. Awesome! We've added another bit of functionality to our application. A flight now knows about its list of passengers.

But a passenger doesn't know that they've been booked on a flight. Let's add that by writing a test for it:

Download tdd/reservations/passenger_test.rb

```
def test_passenger_knows_about_reservation
  # Create a passenger
  passenger = Passenger.new :name => 'John Smith'

  # Create a flight
  flight = Flight.new :number => '321',
                      :origin => 'ORD',
                      :destination => 'JFK'

  # Make reservation
  flight.reserve(passenger)
```

```
  # Make sure passenger knows about reservation
  assert_not_nil passenger.flight
end
```

Here we're using the built-in assert_not_nil assertion, which takes one argument. The object we pass must not be nil in order for the test to succeed.

Let's take an educated guess at how to implement this code. We just need to add a belongs_to association in our Passenger class:

Download tdd/reservations/passenger_2.rb

```
class Passenger < ActiveRecord::Base

  validates_presence_of :name
  belongs_to :flight

end
```

So, rake again, and...the test still fails:

```
  1) Failure:
test_passenger_knows_about_reservation(PassengerTest)
    [./test/unit/passenger_test.rb:42:in
            `test_passenger_knows_about_reservation'
<nil> expected to not be nil.
```

That's odd, isn't it? We know that the flight has the passenger in its has_many list of passengers, so why hasn't the belongs_to association worked?

Look again at the reserve method we wrote earlier. We pushed the passenger object into the list of passengers, but we never saved the new list to the database. We need to call save:

Download tdd/reservations/flight_2.rb

```
def reserve(passenger)
  passengers << passenger
  save
end
```

Run rake again, and voila, all of our tests now pass.

Canceling a Reservation

Now that we can make a reservation, let's cancel it. You know what comes first: let's write a test.

Download tdd/reservations/passenger_test.rb

```
def test_cancel_reservation
  # Create a passenger
  passenger = Passenger.new :name => 'John Smith'

  # Create a flight
  flight = Flight.new :number => '321',
                      :origin => 'ORD',
                      :destination => 'JFK'

  # Make reservation
  flight.reserve(passenger)

  # Cancel it
  flight.cancel(passenger)
  passenger.reload

  # Make sure reservation is cancelled
  assert !flight.passengers.include?(passenger)
  assert_nil passenger.flight
end
```

This will fail as expected, since we haven't written a cancel method yet. Let's do that now:

Download tdd/reservations/flight_2.rb

```
def cancel(passenger)
  passengers.delete passenger
end
```

Let's rake again, and presto—all tests pass.

We've done it: we've fulfilled all of our requirements, and we have an automated regression test suite to boot. Any time that we change our code, we can run our tests to find out whether we've broken any of our requirements. This means we can change our implementation code as much as we want, whenever we want, and be confident that our software still works as planned.

But we're not done. Yes, our tests pass, but they are ugly. We have duplicated a bunch of code in our tests. It's time to DRY them up.

DRYing Up Tests with Setup Methods

Three of our tests need the same test data to work with, so they each have code dedicated to creating this data:

- A passenger object

- A flight object
- A reservation for the passenger on that flight

Being good agile developers, we could extract this code into a separate method. That would be an improvement, but we'd still have the ugliness of needing to call that method at the start of each test. Since this is a recurring pattern in unit test code, the Test/Unit library provides a solution: the setup method.

If we choose to define a setup method for our class, it will automatically be detected and called before each and every test. It's a great place to put code that creates data for the tests to play with. Let's refactor the passenger_test.rb file to use a setup method:

Download tdd/reservations/passenger_test_after_refactoring.rb

```
require File.dirname(__FILE__) + '/../test_helper'

class PassengerTest < ActiveSupport::TestCase

  def setup
    # Create a valid passenger
    @passenger = Passenger.new :name => 'John Smith'

    # Create a flight
    @flight = Flight.new :number => '321',
                         :origin => 'ORD',
                         :destination => 'JFK'

    # Make reservation
    @flight.reserve(@passenger)
  end

  def test_passenger_invalid_if_name_is_missing
    # Create a passenger without a name
    p = Passenger.new :email => 'someone@example.com'

    # Make sure the passenger object is not valid
    assert_equal false, p.valid?
  end

  def test_make_reservation
    assert @flight.passengers.include?(@passenger)
  end

  def test_passenger_knows_about_reservation
    assert_not_nil @passenger.flight
  end
```

```
  def test_cancel_reservation
    @flight.cancel(@passenger)

    # Make sure reservation is cancelled
    @passenger.reload
    assert !@flight.passengers.include?(@passenger)
    assert_nil @passenger.flight
  end
end
```

The tests have become eminently more readable. Without the clutter of all the setup code, the intent of each test method shines through a bit more.

It's worth noting that by removing local variables from the individual tests, we had to create instance variables so that the test methods could have access to the @flight and @passenger objects.

Providing Test Data with Fixtures

Our little reservation system doesn't need a lot of data to test with, but in larger applications the setup method can become quite large. We want our test code to be all about testing our application, not about preparing a bed of test data. Rails provides us with test *fixtures*, which are an alternative to creating test data by hand in code.[1]

When we created our flight and passenger models, Rails also generated fixture files for each model. Fixtures are a way to easily populate our test database with data, without having to ever touch the database itself. Using easily readable YML files, we can specify the data we want to have in our test database before each test is run.

Here's our test/fixtures/flights.yml file:

Download tdd/reservations/flights.yml

```
ord_to_jfk:
  number: 123
  origin: ORD
  destination: JFK
```

YML files are plain-text files that are easy to read and write:

- Key/value pairs are separated by a colon.

1. Read about fixtures at http://ar.rubyonrails.org/classes/Fixtures.html.

- Sections can be created by putting a section name on its own line ending with a colon. In Rails fixture files, each section becomes a named row in a table in the test database.

For example, the flights.yml fixture specifies the rows of data for the flights table in our test database. In line 1, we start a table row named ord_to_jfk and provide the necessary column values.

We do the same with our passengers table. Our tests use two kinds of passengers: one with just an email address and one that is a valid passenger object. We therefore specify two rows in our table:

Download tdd/reservations/passengers.yml

```
email_only:
  email: someone@example.com

john_smith:
  name: John Smith
  email: someone@example.com
  flight: ord_to_jfk
```

Notice an important feature of Rails fixtures in line 7. Our passengers table has a foreign key column, flight_id, which relates the passengers table to the flights table. In our fixture files, we don't need to worry about managing a list of foreign key identifiers. We can simply use the named section from our flights fixture. Here we are specifying the john_smith row in the passengers table should have a foreign key to the ord_to_jfk row from the flights table. Our test class has attained a new level of readability:

Download tdd/reservations/passenger_test_with_fixtures.rb

```
require File.dirname(__FILE__) + '/../test_helper'

class PassengerTest < ActiveSupport::TestCase

  def setup
    # Create a valid passenger
    @passenger = passengers(:john_smith)

    # Create a flight
    @flight = flights(:ord_to_jfk)

    # Make reservation
    @flight.reserve(@passenger)
  end

  def test_passenger_invalid_if_name_is_missing
```

Joe Asks...

When Does Rails Load the Fixture Data?

By default, fixture data is loaded only once each time you run the rake command. Rails then runs each test inside of its own database transaction, rolling back each transaction after each test runs. This ensures that the database returns to a clean state for each test.

```
  def test_make_reservation
    assert @flight.passengers.include?(@passenger)
  end

  def test_passenger_knows_about_reservation
    assert_not_nil @passenger.flight
  end

  def test_cancel_reservation
    @flight.cancel(@passenger)

    # Make sure reservation is cancelled
    @passenger.reload
    assert !@flight.passengers.include?(@passenger)
    assert_nil @passenger.flight
  end
end
```

Instead of assigning our @passenger and @flight variables by manually creating them with hardcoded data, we use the passengers and flights methods to automatically retrieve the rows by YML section name.

By using well-named names in our YML files, the intent of our tests becomes clearer. For example, :email_only is a more meaningful name than if we had used passenger_2, because it conveys something about the actual content and intent of that row in our test database.

Because the Test/Unit library is a standard Ruby library installed with Ruby, there can be no excuses for not giving it a try. We've seen how writing tests rely on very little overhead or ceremony:

1. Start with require 'test/unit' at the top of your Ruby file.
2. Declare a regular Ruby class that derives from Test::Unit::TestCase (or

The Missing Model

A limitation of our current design has become obvious: we have a one-to-many relationship between a passenger and a flight. There is no way for a passenger to book more than one flight—not even their return flight. We might as well name our application the Roach Motel Flight Reservation System: passengers can fly somewhere, but they can never come back.

We can improve our design by writing more tests. Here's one way we could proceed:

```
def test_round_trip_reservations
  @passenger.reservations.create
                :flight => flights(:go_on_vacation),
                :payment => @credit_card
  @passenger.reservations.create
                :flight => flights(:come_back_home)
                :payment => @credit_card

  assert_equal 2, @passenger.flights.count
end
```

This test will fail the first time we run it, forcing us to improve the design of our models. We've discovered a model that has gone missing until now: a Reservation that associates a passenger to a specific flight. A Reservation can take care of all the details regarding a specific reservation.

Let your tests drive the design of your application.

ActiveSupport::TestCase in Rails, which in turn derives from Test::Unit::TestCase).

3. Define methods the normal way. Methods that start with test_ are automatically identified as test methods.
4. You can optionally define a setup method, which will be called before each test is run.

Test-driven development is one of those daily practices that experienced developers come to rely on, and Test/Unit has historically been the bread and butter of Ruby development. In the next section, we will take a quick look at *behavior-driven development*, an outgrowth from and an alternative to the test/unit style of testing.

Behavior-Driven Development with Shoulda

Very recently, a new twist on TDD has begun to gain favor in some agile development circles. Behavior-driven development (BDD) intends to make writing tests easier by changing some of the terminology and the way in which test methods are created. The basic tenets of test-driven development still hold: we write tests first as a way of driving the design and implementation of our application.

One of the primary benefits of a test-oriented approach is that the tests serve as a living document of the requirements. A word processing document can specify all the application's requirements. Test suites go one better in that they not only capture requirements, but they are executable pieces of code that tell us how well our application is conforming to those requirements.

BDD takes this notion one step further by elevating the role of the test to a loftier place. Instead of thinking about writing dull, boring "tests," we instead think of them in terms of *specifications*. The intention is to provide a more comfortable segue from the language of business requirements into the language of Ruby.

To that end, there are two main differences between writing tests with Test/Unit and a BDD framework:

- We don't write test methods; we write specifications. Specifications are 100% executable Ruby code but are written using methods inherited from a BDD framework.
- The specification framework leverages the metaprogramming wonders of Ruby to provide us with a more English-like vocabulary to describe our specifications.

To think in BDD instead of classical TDD, we must begin to think as a user, not as a developer, and as a stakeholder who writes user stories, not as a programmer who thinks in terms of bits and bytes. The Ruby world is currently undergoing an early evolution of BDD frameworks. Three frameworks in particular stand out:

- RSpec,[2] a full-blown Ruby framework for behavior-driven development in Ruby and in Rails
- test/spec,[3] a lightweight alternative to RSpec

2. http://rspec.info/
3. http://test-spec.rubyforge.org/test-spec/

RSpec

RSpec is a behavior-driven development framework for Ruby. An RSpec plug-in for Rails is also available so that RSpec syntax can be used inside of Rails unit, functional, and integration tests. Unlike Shoulda and test/spec, RSpec is not an addition to or superset of Test/Unit; it is a complete replacement. The syntax for describing test contexts and specs are completely different. Some BDD developers prefer RSpec's choice of vocabulary terms over those seen in Test/Unit, test/spec, and Shoulda.

The key highlights of RSpec are as follows:

- An English-like vocabulary for application-level specifications
- A "story runner" framework for describing user acceptance tests in plain text
- Full replacement of Test/Unit-style assertions with specification-style "should" methods
- Generation of spec listings and "red-green"-style output
- rake integration (with Rails plug-in)
- A built-in mocking and stubbing framework

RSpec was one of the first widely available BDD frameworks for Ruby, and it continues to grow and evolve. Check out http://rspec.info for more information and to download the RSpec gem and Rails plug-in.

RSpec inspired projects like test/spec (which uses an "RSpec-inspired syntax"), Shoulda, and others are certain to appear that follow the spirit of RSpec but require a smaller footprint (and correspondingly fewer features). We encourage you to give RSpec a try and choose the framework that's right for you.

- Shoulda,[4] another lightweight BDD framework that integrates well with Ruby's traditional Test/Unit library

In this book we will focus on Shoulda. Covering RSpec in depth is beyond this book, but those who are curious can refer to the sidebar on this page to learn more about how to get started with RSpec.

4. http://thoughtbot.com/projects/shoulda

First, we need to install Shoulda into our Rails project. See the Shoulda home page for the latest version and installation instructions, but as of this writing we can easily install it as a Rails plugin if we have Git (see Section 1.4, *Git*, on page 19 for how to get started with Git):

```
reservations$ script/plugin install
                git://github.com/thoughtbot/shoulda.git
```

On Windows, the command looks like this:

```
c:\dev\reservations> ruby script/plugin install
                      git://github.com/thoughtbot/shoulda.git
```

When we wrote our unit tests with the Test/Unit framework, we would think in terms of writing classes and methods. BDD replaces these concepts with *contexts* and *specs* (as in, specifications).

We will write our tests for the Passenger class over again from scratch, this time with the Shoulda framework. Let's start by declaring a single context and a single test:

Download tdd/reservations/passenger_test_shoulda_1.rb

```
require File.dirname(__FILE__) + '/../test_helper'

class PassengerTest < ActiveSupport::TestCase

  context "A passenger" do

    setup do
      @passenger = passengers(:email_only)
    end

    should "have a name" do
      assert_equal false, @passenger.valid?
    end

  end

end
```

This looks and feels quite different from our old Test/Unit class. We still derive from the same base class, but there does not appear to be any Ruby methods at all! In reality, however, the context and should methods are generating test methods for us behind the scenes. We've replaced the Ruby method definition drudgery with the more airy and English-like "should" syntax that can help our tests look and feel more like real-world specifications that an end user would write. Let's walk

through it slowly to get our bearings before we add the rest of our tests (oops, I mean specs).

Contexts

On line 5 we declare a *context*. A context simply is a container around a set of related specs. Contexts are given a name and a block of code, which contain specs and (optionally) nested contexts.

Contexts can contain setup methods, just as TestCase classes do. These setups work just as you would expect, being run before each spec is executed. However, because we can have multiple contexts per test case (and even nested contexts), we can fine-tune our setups better than we could with Test/Unit. We will see an example of this shortly.

The setup for our context is quite minimal to start:

Download tdd/reservations/passenger_test_shoulda.rb

```
setup do
  @passenger = passengers(:john_smith)
end
```

Specs

Specifications (or specs, for short) always start with the word should. Like contexts, specs are also given a name and a block of code. The code block we provide serves as the bridge between Shoulda and Test/Unit. Inside the block goes regular Ruby Test/Unit-style assertion code.

We start our first spec on line 11. The name of the spec is intended to be a continuation of an English sentence that starts with the name of the context, followed by the word *should*. The full name for our spec becomes "A passenger should have a name," and that's exactly what will be displayed if this test were to ever fail.

Let's see it in action:

```
c:\dev\reservations> rake
Loaded suite /.../lib/rake/rake_test_loader
Started
..
Finished in 0.094631 seconds.

2 tests, 2 assertions, 0 failures, 0 errors
```

So far, so good. (If you're wondering why it reports two tests instead of one, recall that we still have a default test written for us in flight_test.rb.)

Let's intentionally make our test fail to see what will be displayed:

```
c:\dev\reservations> rake
Loaded suite /../rake/rake_test_loader

1) Failure:
test: A passenger should have a name. (PassengerTest)
[./test/unit/passenger_test.rb:12:in `__bind_1211299740_775702'
 /.../gem/shoulda.rb:189:in `call'
 /.../gem/shoulda.rb:189:in `test: A passenger should have a name. '
 /.../active_support/testing/default.rb:7:in `run']:
<true> expected but was
<false>.
```

Here we can see the name of the spec that failed, "A passenger should have a name," which was constructed for us. The rest of the output is standard rake output using the Test/Unit library.

More Specs

Let's add a new context for the remaining specs:

Download tdd/reservations/passenger_test_shoulda.rb

```
context "makes a reservation" do

  setup do
    @passenger = passengers(:john_smith)
    @flight = flights(:ord_to_jfk)
    @flight.reserve(@passenger)
  end

  should "be able to get a seat" do
    assert @flight.passengers.include?(@passenger)
  end

  should "have a reservation" do
    assert_not_nil @passenger.flight
  end

  should "be able to cancel it" do
    @flight.cancel(@passenger)
    @passenger.reload

    assert !@flight.passengers.include?(@passenger)
    assert_nil @passenger.flight
  end
end
```

We start the context on line 1, and it gets a setup method of its own. We then create three specifications, carrying over the same Ruby code

BDD isn't for everyone. Some feel that the vocabulary defined by their particular BDD framework makes it easier to translate business requirements into code. Others feel it's unnecessary sugar and that the Test/Unit library is more than sufficient.

Whether you stick with Test/Unit or choose one of the new BDD frameworks, the important thing is to write tests. TDD and BDD are empowering techniques that give you the freedom to change your application code at will as your code grows, without losing the elegance and simplicity it had when it first got started.

Chapter 11

Integrating with .NET

In a .NET enterprise environment, it can sometimes be hard to justify the investment in a new Rails project. It can be helpful to have some assurance that Rails applications can leverage the .NET infrastructure that has already been built. Data exposed by .NET web applications for consumption by .NET web services can also be consumed by Rails applications. Rails applications can therefore play an important role in existing .NET environments.

Perhaps more interestingly, .NET client applications—WinForms and "smart client" applications, Windows Mobile applications, Tablet PC software, and more—can all use the functionality delivered by Rails applications. The secret is in the respond_to magic that RESTful applications employ to provide tailor-made responses to incoming requests. Rails applications that conform to RESTful best practices can immediately expose the same functionality to XML-seeking .NET clients as they do for HTML-seeking web browsers.

In this chapter, we will take a tour of how you can start to integrate .NET client applications with your Rails services, as well as how we can write Ruby code to call a SOAP service exposed by a .NET 3.0 web service project.

Using a Rails Web Service from .NET

When we explored the RESTful side of Rails controllers, we learned that controllers can provide different responses for different types of clients. When a web browser comes calling with an HTTP request, we

can respond with an HTML stream. When an RSS reader makes a request, we can provide an XML feed. In fact, we can respond to any kind of request by simply examining the Accept header of the incoming HTTP request and then responding appropriately.

In other words, when you use respond_to in your controllers, you've instantly transformed your website into a real live web service. If you're working in a .NET environment that is also now working with Rails, you can use the .NET Framework to create a rich client application that connects to the web service you're built in Rails.

So, in this section, we will turn the tables. We're going to take a look at this process from the client application's point of view. We will see how to use the HTTP request and response streams to provide an entirely different kind of user interface on our flight and passenger data.

We will write a .NET client application that can access our passenger data, as provided by the Rails application we developed in Part II. We will build a standard WinForms application, but the core functionality we will develop is reusable across any other kind of .NET application you can think of: embedded applications, handwriting recognition applications for the Tablet PC, mobile applications for smartphones, or even inside other ASP.NET web applications and services.

Designing the WinForms UI

Our .NET application will be a bare-bones WinForms application. Many tomes are dedicated to writing rich client applications in .NET, so we won't be talking much about .NET UI development here. Instead, we will focus on how your .NET code can interact with a RESTful Rails web service. When you run the application, it looks like Figure 11.1, on the following page.

Our main form consists of a listview and four buttons. Each button will exercise a different kind of HTTP request against our Rails service.

One important note: our C# code expects to call a Rails service that's up and running on local port 3000. If you want to try this code on your machine, be sure to have your Rails application running on your machine on that port, or change the URL in our examples as necessary to fit your environment.

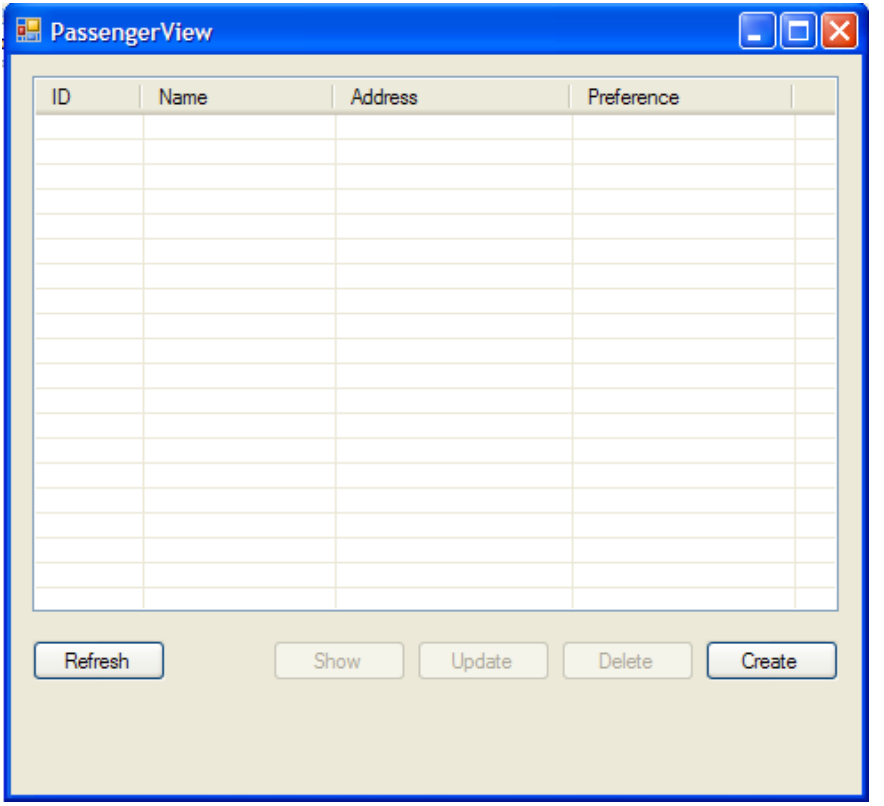


Figure 11.1: An empty passenger list

Calling the Index Action

Let's first look at how we call the index action on the PassengersController. This will return a list of all the passengers in the database. In the real world, this would probably not be wise, unless we have very unpopular airlines.

For now, we're using our local development environment with a small number of passengers, so we don't have to worry about performance or caching considerations.

Here's the event handler for the Refresh button's click event:

Download dotnetintegration/passengerview.cs

```
private void buttonRefreshList_Click(object sender, EventArgs e)
{
    // Clear listview
    passengers.Items.Clear();

    // Create the url to the index action of the Passengers controller
    string url = "http://localhost:3000/passengers";

    // Create an HTTP request
    HttpWebRequest req = (HttpWebRequest) WebRequest.Create(url);

    // Specify GET as the HTTP verb we want to use
    req.Method = "GET";

    // Specify that we can only accept XML
    req.Accept = "text/xml";

    // Call the web service and capture the returned XML
    WebResponse resp = req.GetResponse();
    StreamReader reader = new StreamReader(resp.GetResponseStream());
    string xml = reader.ReadToEnd();

    // Parse the received XML
    XmlDocument doc = new XmlDocument();
    doc.LoadXml(xml);
    XmlNodeList passengerList = doc.SelectNodes("//passenger");

    // Loop through each xml passenger
    foreach (XmlNode passenger in passengerList)
    {
        // Parse out passenger details
        string id = passenger["id"].InnerText;
        string name = passenger["name"].InnerText;
        string address = passenger["address"].InnerText;
        string preference = passenger["aisle-preference"].InnerText;

        // Add passenger to the listview
        // Save the passenger ID in the Tag property of the item
        ListViewItem item = passengers.Items.Add(id);
        item.SubItems.Add(name);
        item.SubItems.Add(address);
        item.SubItems.Add(preference);
        item.Tag = passenger["id"].InnerText;
    }
}
```

We start by generating a properly formed, RESTful URL to our index

host and port for your environment, as long as you recognize the important part, which is /passengers.

You may recall that the RESTful routing in Rails provides two possible actions for the /passengers URL: one to call the index action and one to call the create action. Rails has to decide which one to invoke, and its decision is based on the HTTP verb that the client has specified in the HTTP request header. We do that by utilizing the Method property of our HttpWebRequest object.

Now Rails understands that we're calling the index action in our controller. But our application can respond to both HTML and XML. We want to make sure we trigger the format.xml branch in our controller's respond_to block. We do that by setting the request's Accept property to text/xml.

Properly setting the Accept header in the HTML request is absolutely critical. If you are receiving HTML from your web service, it's likely you've forgotten to set the Accept header to the proper value.

In fact, changing the Accept header is the magic wand that enables you to extract data from your web service in any format you want! HTML, XML, JSON, CSV, or any other custom format you can dream of is accessible to your client applications by properly using the HTTP verb and Accept header.

All that's left for our .NET code to do is to actually send the request and receive the response, which we do with the GetResponse method. If all goes well, you'll have some tidy XML captured in the xml string variable.

Finally, we parse the XML and add a row to the listview for each passenger we can extract from the XML data.

If you're seeing something like Figure 11.2, on the next page (though probably with different passenger names), congratulations! You've just made a web service call, from .NET to Rails.

Calling the Show Action

Once we have passengers in the listview, you can select a passenger in the list, which will enable the three other buttons on the form. The Show button demonstrates how to call the show action in our controller.

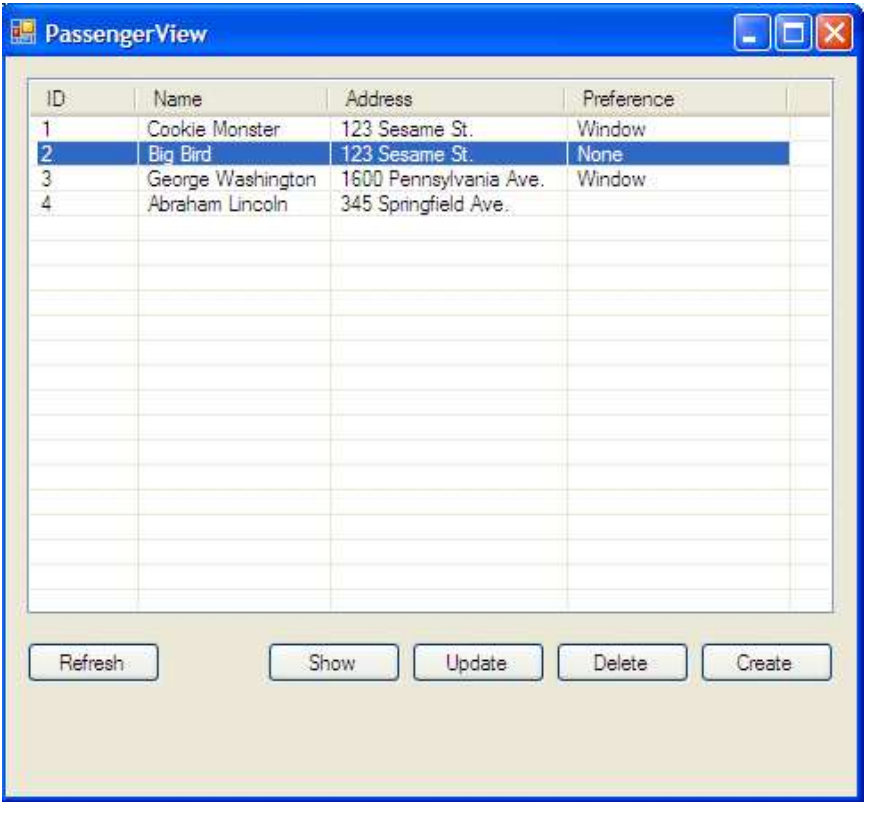


Figure 11.2: Form populated with data from a Rails web service

The code for our Show button's click event handler looks like this:

Download dotnetintegration/passengerview.cs

```
private void buttonShow_Click(object sender, EventArgs e)
{
    if (passengers.SelectedItems.Count != 1)
        return;

    ListViewItem item = passengers.SelectedItems[0];

    // Determine the url
    string url = "http://localhost:3000/passengers/" + item.Tag;

    // Create the url to the index action of the Passengers controller
    HttpWebRequest req = (HttpWebRequest) WebRequest.Create(url);
```

```
    // Specify GET as the HTTP verb we want to use
    req.Method = "GET";

     // Specify that we can only accept XML
    req.Accept = "text/xml";

    // Call the web service and capture the returned XML
    WebResponse resp = req.GetResponse();
    StreamReader reader = new StreamReader(resp.GetResponseStream());
    string xml = reader.ReadToEnd();

    // Parse the received XML
    XmlDocument doc = new XmlDocument();
    doc.LoadXml(xml);
    XmlNode passenger = doc.SelectSingleNode("//passenger");
    string id = passenger["id"].InnerText;
    string name = passenger["name"].InnerText;
    string address = passenger["address"].InnerText;
    string preference = passenger["aisle-preference"].InnerText;

    MessageBox.Show(string.Format("ID: {0}\nName: {1}\nAddress:
                    {2}\nAisle Preference: {3}", id, name,
                    address, preference));
}
```

Most of this code is identical to what we wrote when we called the index action. (Good agile developers are cringing right now, seeing the need for some refactoring.) In fact, there are only two differences:

- We construct the URL by using the Tag property from the selected item. This will generate a URL like http://localhost:3000/passengers/3 so that we can call the show action, setting params[:id] to 3 inside the Rails framework.
- We receive a single XML node, not a list, so the parsing code is a bit different.

We then show a simple message box to prove that we have indeed received passenger data from the web service.

Deleting a Passenger

So far we've used the GET verb to call the index and show actions. The Delete button demonstrates how we can manipulate the HTTP header to call the destroy action in our controller.

Here's the code for the Delete button's click event handler:

Download dotnetintegration/passengerview.cs

```
private void buttonDelete_Click(object sender, EventArgs e)
{
    foreach (ListViewItem item in passengers.SelectedItems)
    {
        if (MessageBox.Show("Delete Passenger #" + item.Tag + "?",
                  "Confirm Delete", MessageBoxButtons.YesNo)
                      == DialogResult.Yes)
        {
          string url = "http://localhost:3000/passengers/" + item.Tag;

          HttpWebRequest req = (HttpWebRequest)WebRequest.Create(url);
          req.Method = "DELETE";
          req.Accept = "text/xml";
          req.ContentType = "application/xml";

          WebResponse resp = req.GetResponse();

          MessageBox.Show(resp.StatusCode.ToString());
        }
    }
}
```

It's much shorter than our previous event handlers, because we're not expecting any data back from the web service. Instead, we display the status code we received from the web service. If all goes well, you should get a message box that says "OK."

The key elements in this code are as follows:

- For each selected passenger, we construct the passenger-specific URL.
- We set the HTTP verb to DELETE.

Try it! Select a passenger, and click the Delete button. You can then click Refresh, and the passenger will no longer show up in the list.

If you want to take a look behind the curtain, return to your command prompt, start your Rails console, and do a Passenger.find(:all) or Passenger.count—you'll find that the passenger was truly deleted from your database.

Creating a New Passenger

Creating a new Passenger object via our Rails web service is straightforward but does require a few extra housekeeping details. In the following

code, we've hardcoded the passenger data we want to create. You can enhance the code by creating a form for the user to fill out instead. As you read the code, you'll see that much of the code is the same as before. See whether you can spot what's new.

Download dotnetintegration/passengerview.cs

```
private void buttonCreate_Click(object sender, EventArgs e)
{
    // Create a new passenger
    string name = "Grover";
    string address = "123 Sesame St.";
    string preference = "Window";

    // Determine the url
    string url = "http://localhost:3000/passengers/";

    // Create the request
    HttpWebRequest req = (HttpWebRequest)WebRequest.Create(url);

    // Specify the POST verb
    req.Method = "POST";

    // Specify XML both ways
    req.Accept = "text/xml";
    req.ContentType = "application/xml";

    // Create the XML to send to the webservice
    string xml = "<passenger>";
    xml += "<name>" + name + "</name>";
    xml += "<address>" + address + "</address>";
    xml += "<aisle-preference>" + preference + "</aisle-preference>";
    xml += "</passenger>";

    // Encode the XML
    ASCIIEncoding encoding = new ASCIIEncoding();
    byte[] byte1 = encoding.GetBytes(xml);

    // Set the Content-Length header
    req.ContentLength = byte1.Length;

    // Write the XML into the request
    Stream stream = req.GetRequestStream();
    stream.Write(byte1, 0, byte1.Length);
    stream.Close();

    // Send the request and capture the result code
    HttpWebResponse resp = (HttpWebResponse)req.GetResponse();
    MessageBox.Show(resp.StatusCode.ToString());
}
```

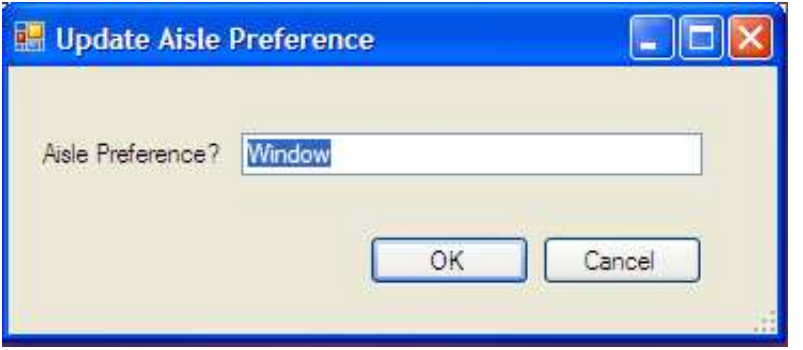


Figure 11.3: Updating the aisle preference attribute

Here are the key elements of creating a new passenger from a .NET client:

- We use the POST verb in the HTTP header instead of GET so that the create action is called instead of the index action.
- We construct a byte-encoded XML string that we can write into the request stream.

All in all, it's a rather straightforward process. If all goes well, you'll get a "CREATED" message box instead of an "OK." Surprised? Remember that our create action returns a 201 CREATED status code instead of a 200 OK.

Updating Passenger Data

Suppose we want to update an attribute of an existing passenger. Our sample application includes a small form that the user can use to change the passenger's aisle preference (see Figure 11.3).

To update the passenger resource, we must be sure to do the following:

- Use the PUT verb in the HTTP request header.
- Construct an XML fragment for the attributes we want to change.
- Use the correct passenger-specific URL.

That last point is an important one. We don't specify the passenger ID we're referring to by configuring it in the XML. Rails convention specifies that the URL will indicate the resource to be used. So, we need

to make sure we use the passenger's ID as part of the URL. Here's the code to update a passenger's aisle preference:

Download dotnetintegration/passengerview.cs

```
private void buttonUpdate_Click(object sender, EventArgs e)
{

    // Demonstrate the Update action

    ListViewItem item = passengers.SelectedItems[0];

    Form2 f = new Form2();
    f.textBox1.Text = item.SubItems[3].Text;
    if (f.ShowDialog(this) == DialogResult.OK)
    {

        // Create passenger-specific URL
        string url = "http://localhost:3000/passengers/" + item.Tag;

        HttpWebRequest req = (HttpWebRequest)WebRequest.Create(url);
        req.Method = "PUT";
        req.Accept = "text/xml";
        req.ContentType = "application/xml";

        ASCIIEncoding encoding = new ASCIIEncoding();
        byte[] byte1 = encoding.GetBytes("<passenger><aisle-preference>" +
                       f.textBox1.Text + "</aisle-preference></passenger>");
        req.ContentLength = byte1.Length;


        Stream stream = req.GetRequestStream();
        stream.Write(byte1, 0, byte1.Length);
        stream.Close();

        HttpWebResponse resp = (HttpWebResponse)req.GetResponse();
        MessageBox.Show(resp.StatusCode.ToString());
    }
}
```

There are two important aspects of this code that are worth pointing out. First, on line 14, we construct the URL that identifies the resource for this specific passenger. This is how the Rails code will know which passenger we want to update. Second, line 17 shows how we transmit the HTTP PUT verb, which Rails will route to the update action inside the controller. The HTTP verb and the URL are the most important things to remember to specify when calling your Rails code from a .NET application.

We've taken a quick tour of how to CRUD our passenger data from a .NET rich client application. Integrating your .NET applications with resources that are managed by a Rails web service is easy to do and can quickly add value to your .NET applications.

Using a SOAP Web Service from Ruby

We've seen how to write .NET code to call into a Rails web service, but what if we need to do the exact opposite? It may be that you already have web services written in .NET and you want your Rails applications to be able to call into those services, too. In this section, we will first write a very simple web service in .NET 3.0 to be our .NET guinea pig, and then we will see how to write Ruby code that calls the method we implemented in C#.

Generating a .NET Web Service

To demonstrate how a Ruby client can connect to a .NET SOAP web service, we'll have to first generate a small web service that we can use. We will generate a web service using the .NET 3.0 Web Service project type (see Figure 11.4, on the next page).

The Project Wizard generates a sample HelloWorld method. We will add another method that we will be able to call when we want to know how many frequent flyer points have been racked up by any given passenger.

Download dotnetintegration/service1.cs

```
[WebMethod]
public int GetPoints(string frequentFlyerNumber)
{
   // Lookup the points for the given
   // frequent flyer number.
   // This really should come from a file, database,
   // or any other source.

   // In this example, we will simply
   // hardcode a fictitious value.

   return 456;
}
```

In a real web service, we would likely be connecting to a SQL Server database or perhaps a legacy mainframe system, which would retrieve the number of frequent flyer points that have been accrued for the given

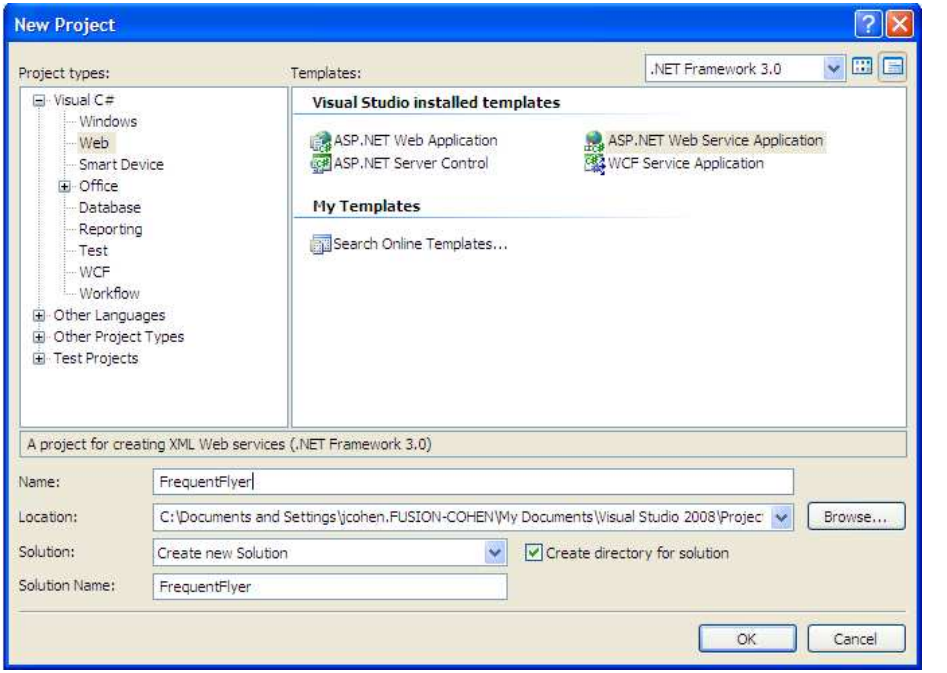


Figure 11.4: Creating a new web service in VS 2008

frequent flyer number. In our example, we're using a hardcoded value, regardless of which frequent flyer number was actually passed.

Running the Web Service

Visual Studio 2008 provides an easy way to start up your web service on a local port on your computer. Right-click the .asmx page in the Project Explorer, and select View in Web Browser. This will start a local web server (if it isn't running already) and then open your browser to the appropriate localhost address.

Generating a Ruby Proxy

Now that we have our web service up and running, we can write a Ruby program that uses it. Every client of a SOAP-based web service must start out by:

- locating a WSDL-compliant description of the web service, and

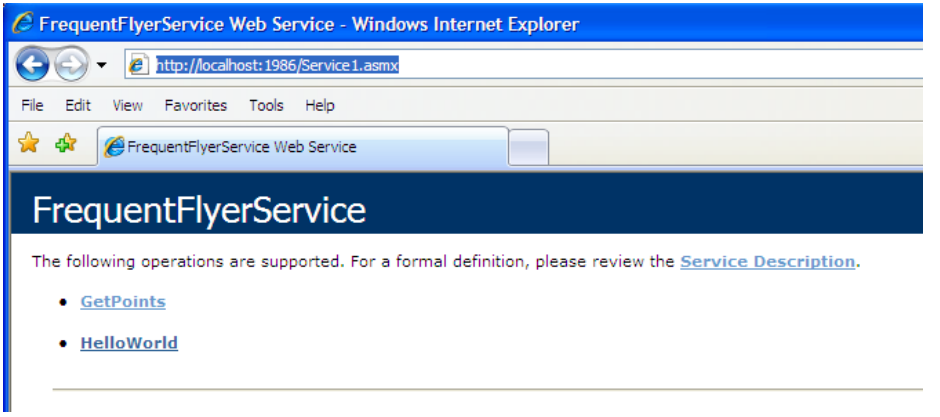


Figure 11.5: Top portion of frequent flyer .asmx page

We start by locating the WSDL for our little web service. The .asmx page in our browser contains a link to the WSDL document. Click the link, and you should see a complicated XML document in your browser. Copy and paste the URL you see in your browser. On our machine, the URL is http://localhost:1986/Service1.asmx?WSDL (see Figure 11.5).

Our next goal is to use the WSDL to generate a *proxy*, which is a Ruby class that exposes the same methods as the web service and which knows how to call the real web service methods on our behalf. To do that, we will use a publicly available Ruby utility called soap4R. Fortunately, soap4R is available as an easily installable Ruby gem. Install it now if you don't already have it:

```
gem install soap4r
```

Now that soap4R is installed, we can generate our proxy code using the wsdl2r utility. All we need to do is hand it the URL to the WSDL for our .NET web service. First, let's create a directory for ourselves to contain our Ruby web service client:

```
C:\dev>mkdir soap_example
C:\dev>cd soap_example
```

Now, we can generate the proxy:

```
wsdl2ruby --wsdl http://localhost:1986/Service1.asmx?WSDL --type client
```

You should see output as it creates four files in the current directory:

```
INFO -- app: Creating class definition.
INFO -- app: Creates file 'default.rb'.
INFO -- app: Creating mapping registry definition.
INFO -- app: Creates file 'defaultMappingRegistry.rb'.
INFO -- app: Creating driver.
INFO -- app: Creates file 'defaultDriver.rb'.
INFO -- app: Creating client skelton.
INFO -- app: Creates file 'FrequentFlyerServiceClient.rb'.
INFO -- app: End of app. (status: 0)
```

wsdl2ruby did a lot of work for us in a short amount of time:

- It downloaded the XML document found at our localhost URL.
- It generated several Ruby scripts, culminating with defaultDriver.rb, which is a full-blown proxy for our .NET web service interface.
- It generated a fourth Ruby script, FrequentFlyerServiceClient.rb, that is a complete example of how to use the generated proxy class.

We can write Ruby code that uses the generated proxy class to call our GetPoints() method like this. Create a new Ruby script called getpoints.rb with this code:

Download dotnetintegration/getpoints.rb

```
require 'rubygems'
gem 'soap4r'

require 'defaultDriver.rb'

proxy = FrequentFlyerServiceSoap.new
response = proxy.getPoints("A1234")

points = response.getPointsResult.to_s

puts "You have #{points} frequent flyer points."
```

It's important to require the defaultDriver.rb file. That defines our proxy class for us. We then call getPoints and pull out the C# return value by calling getPointsResponse from the Ruby response object. Let's just take it for a spin and see what we get:

```
c:\dev\soap_example> ruby getpoints.rb
You have 456 frequent flyer points.
```

Presto! Our Ruby code called into our .NET web service and got the correct value back.

In this chapter, we took a quick look at how Ruby and Rails applications can fit nicely into an existing .NET environment. One of the most challenging aspects of software engineering is being able to choose the right tool for the right job. We don't need to use Rails for everything, just like we don't need to use .NET for everything. They each have strengths and weaknesses that make them more or less suitable for any given situation. The goal is to use each according to their strength and then be able to seamlessly integrate them together.

Chapter 12

Finishing Touches

We've learned a lot about both the Ruby language and the Rails framework. Creating great web applications requires more than just coding skills, however. It's time for us to take a step beyond the nuts and bolts of Rails development and to invest some time in developing some other skills that seasoned Rails developers attain.

To that end, we will explore three key areas that every Rails developer must feel comfortable with: RubyGems, rake, and deployment.

Getting to Know RubyGems

Most software systems have a notion of *package management*. Especially useful for distributing prewritten code libraries, package management refers to some standardized way for users to share source code or executable binaries with each other. Windows has a long tradition of using the Add/Remove Programs facility. Microsoft has published a standard package format for third-party software called the Windows Installer format. These standards make it easy for software creators to distribute their works to a large audience and also provides a standard way for users to manage (that is, install, upgrade, and uninstall) the software they own.

Unlike .NET languages that can be compiled to binary assemblies to make distribution easy, Ruby programs don't have a binary representation. To share your Ruby code with someone else, you can simply give them your code. However, distributing a Ruby library that consists of a large set of Ruby source files can be unwieldy. The user who chooses to use such a library is faced with the unenviable task of managing their

RubyGems is the solution to this problem—a kind of Add/Remove Programs equivalent for Ruby developers. It is the official package management system for Ruby code libraries, and it enables us to consume Ruby code written by someone else in a simple manner, allowing us to install, upgrade, and remove Ruby libraries from our system at will.

A Standard Package Format and Experience

In the same way that the standard Windows Installer file format worked in concert with the Add/Remove Programs applet to help solve software distribution problems, Ruby has its own standard packaging format for Ruby libraries that works in concert with a standard utility program for installing, upgrading, and removing third-party Ruby code.

A Ruby code library packaged in this standard format is called a Ruby *gem*. Smart pun notwithstanding, a gem is a single-file encapsulation of all the Ruby files you want to distribute. But a gem is more than a simple ZIP file of your Ruby code. It also describes certain metadata of the code library contained within it: who created it, the version number, the operating system or platform that it was built for, and more.

RubyGems is the system we use to interact with gems. The `gem` utility provides us with a consistent experience for installing, upgrading, and removing third-party code libraries. RubyGems automatically stores all gems in a central location on your hard drive. It is easy to install new gems from both official and user-supplied web locations ("gem repositories") and even makes it easy to manage multiple versions of a gem (this is the Ruby equivalent of .NET "side-by-side" installations).

Let's learn how to use RubyGems to fine-tune and enhance our Ruby environment.

Exploring the RubyGems Commands

We can get a brief summary of how to use the RubyGems system by using the `help` command:

```
c:\dev> gem help

  RubyGems is a sophisticated package manager for Ruby.  This is a
  basic help message containing pointers to more information.

    Usage:
      gem -h/--help
      gem -v/--version
      gem command [arguments...] [options...]
```

```
    Examples:
      gem install rake
      gem list --local
      gem build package.gemspec
      gem help install

    Further help:
      gem help commands            list all 'gem' commands
      gem help examples            show some examples of usage
      gem help platforms           show information about platforms
      gem help &lt;COMMAND&gt;     show help on COMMAND
                                   (e.g. 'gem help install')
    Further information:
      http://rubygems.rubyforge.org
```

It is also important to know how to determine what version of the RubyGems system you have installed on your machine by using the -v option. Here we can see that we have version 1.2.0 installed:

```
c:\dev> gem -v
1.2.0
```

To upgrade your RubyGems system to the latest version available, you should use this command:

```
c:\dev> gem update --system
```

For the rest of this chapter, we will assume you have version 1.1 or higher installed.

It's easy to view a list of all possible gem commands:

```
c:\dev> gem help commands
```

We're now ready to walk through the gem commands Rails developers use most.

Finding Out What Gems Are Installed

When you first installed Ruby, you installed not only the Ruby interpreter and the RubyGems system, but several gems as well. You can find out what gems you have installed on your machine by using the list command:

```
c:\dev> gem list

*** LOCAL GEMS ***

actionmailer (2.1.0, 2.0.2)
actionpack (2.1.0, 2.0.2)
activerecord (2.1.0, 2.0.2)
```

```
activesupport (2.1.0, 2.0.2)
cgi_multipart_eof_fix (2.5.0, 2.1)
fastthread (1.0)
gem_plugin (0.2.3, 0.2.2)
mongrel (1.0.1)
rails (2.1.0, 2.0.2)
rake (0.7.3)
RedCloth (3.0.4)
rubygems-update (0.9.5)
sources (0.0.1)
sqlite3-ruby (1.2.1)
tzinfo (0.3.5)
```

For each installed gem, you will see the following:

- The name of the gem
- A list of version numbers for the gem

That second point implies a lot of functionality: you can have more than one version of a particular gem installed at the same time. When your code uses a gem's classes or methods, you can choose to always use the latest version of the gem or select a specific version (or even specify a range of allowable versions). For details on how to use gems from within your own non-Rails applications, see *Programming Ruby* [TFH05].

Installing a New Gem

When you want to install a new gem onto your computer, RubyGems can install it for you directly from the Internet. Let's suppose we want to install the win32-sound gem, which provides a wrapper around the Windows sound API:

```
c:\dev> gem install win32-sound

Successfully installed win32-sound-0.4.1
1 gem installed
Installing ri documentation for win32-sound-0.4.1...
Installing RDoc documentation for win32-sound-0.4.1...
```

That was so easy that it bears some explanation of what really happened under the hood. The RubyGems system actually did a lot of work for us in a short amount of time:

- It searched well-known, sanctioned Internet locations, known as *remote sources* for a gem named win32-sound.
- It identified our operating system platform (Windows or Mac, for example).

- It downloaded the latest version of the win32-sound gem that has been made available for our platform.
- It installed the gem into our local gem repository.
- By default, it will download and installed any dependent gems.
- It generated the RDoc documentation for the gem.

Uninstalling a Gem

Getting rid of a gem is just as easy:

```
c:\dev> gem uninstall win32-sound
Successfully uninstalled win32-sound-0.4.1
```

Notice how it reported the full name of the gem, including the version number. If you have multiple versions of a gem installed, you will be prompted to select which version (or all) you'd like to remove. Alternatively, you can use the --version option to explicitly state which version you'd like to uninstall.

Environment Settings

Sometimes it's helpful to have RubyGems report what it believes to be our platform, local gem repository directory, and other elements of our local environment. As you may have guessed, RubyGems provides that information to us with the environment command. We will also take this chance to demonstrate how we can abbreviate gem command names (so long as they aren't so short as to conflict with another command):

```
c:\dev>gem env
RubyGems Environment:
  - RUBYGEMS VERSION: 0.9.5 (0.9.5)
  - RUBY VERSION: 1.8.6 (2007-03-13 patchlevel 0) [i386-mswin32]
  - INSTALLATION DIRECTORY: c:/ruby/lib/ruby/gems/1.8
  - RUBY EXECUTABLE: c:/ruby/bin/ruby.exe
  - RUBYGEMS PLATFORMS:
    - ruby
    - x86-mswin32-60
  - GEM PATHS:
     - c:/ruby/lib/ruby/gems/1.8
  - GEM CONFIGURATION:
     - :update_sources => true
     - :verbose => true
     - :benchmark => false
     - :backtrace => false
     - :bulk_threshold => 1000
  - REMOTE SOURCES:
     - http://gems.rubyforge.org
```

Use of sudo required on Mac/Linux

The command shell sessions we're showing here demonstrate typical usage on Windows. However, on *nix platforms, you sometimes need to wrap your gem commands with sudo to temporary elevate your permissions environment. This is generally required only when installing, updating, or removing gems. If you see an error like this:

```
$ gem install activerecord
ERROR:  While executing gem ... (Gem::FilePermissionError)
    You don't have write permissions into the
      /usr/local/lib/ruby/gems/1.8 directory.
```

then you need to use the sudo command:

```
$ sudo gem install activerecord
```

The GEM PATHS folder is the directory where all gems are located on your computer. If you open that folder, you can (after drilling down into the appropriate subdirectories) find the win32-sound.gem file that we just installed.

Getting Help for a Specific Command

This would be a good time to learn how to get detailed information on a specific command. Let's say we wanted to learn more about the options available for the uninstall command:

```
c:\dev> gem help uninstall
Usage: gem uninstall GEMNAME [GEMNAME ...] [options]

Options:
  -a, --[no-]all                  Uninstall all matching versions
  -i, --[no-]ignore-dependencies  Ignore dependency requirements
                                   while uninstalling
  -x, --[no-]executables          Uninstall applicable executables
                                   without confirmation
  -v, --version VERSION           Specify version of gem to uninstall
      --platform PLATFORM         Specify the platform of gem to
                                   uninstall

Common Options:
  -h, --help                      Get help on this command
  -V, --[no-]verbose              Set the verbose level of output
  -q, --quiet                     Silence commands
      --config-file FILE          Use this config file
```

```
  Arguments:
    GEMNAME       name of gem to uninstall

  Summary:
    Uninstall gems from the local repository

  Defaults:
    --version '>= 0' --no-force
```

Upgrading a Gem

Sometimes we want to upgrade a gem to a new version. Upgrading a gem is similar to installing a gem from scratch:

```
c:\dev> gem update win32-sound
```

To find out whether we have any gems that could be updated, without actually performing the update, we can use the outdated command:

```
c:\dev> gem outdated
actionmailer (2.0.1 < 2.0.2)
actionpack (2.0.1 < 2.0.2)
activerecord (2.0.1 < 2.0.2)
activeresource (2.0.1 < 2.0.2)
activesupport (2.0.1 < 2.0.2)
hpricot (0.4 < 0.6)
rails (2.0.1 < 2.0.2)
win32-clipboard (0.4.1 < 0.4.3)
win32-dir (0.3.1 < 0.3.2)
win32-eventlog (0.4.3 < 0.4.7)
win32-file (0.5.3 < 0.5.5)
win32-file-stat (1.2.3 < 1.2.7)
win32-process (0.5.1 < 0.5.5)
win32-sapi (0.1.3 < 0.1.4)
windows-pr (0.6.2 < 0.7.4)
```

You can see that for each outdated gem, it reports the name of the gem that's outdated, indicated the version currently installed vs. the latest version that's available.

Next, let's find out how we can use a gem's functionality from inside a Rails application.

Using Gems in Your Rails Applications

Using a third-party gem from within your Rails code is dirt simple. Rails knows to activate the RubyGems environment when it starts up, so all you have to do is tell Rails which Rails your application depends upon. You'll then need to decide whether you want to bundle a copy of each

gem with your application or whether you'll be installing the gems on the server.

Adding Gem Specifications

First, locate your config/environment.rb file. Open it up, and inside you'll see a block that begins like this:

```
Rails::Initializer.run do |config|
```

Inside this block, we can add a variety of configuration settings and options. For example, here's how we tell Rails that we need to load version 1.1 or higher of the pdf-writer gem:[1]

```
Rails::Initializer.run do |config|

  # Require version 1.1 or higher
  config.gem "pdf-writer", :version => '1.1'

end
```

You can specify as many gems as you need in this way. The version parameter is optional, and we recommend you check the latest Rails documentation for details on this and other optional parameters you can use.

Including Gems in Your Application

Although we've installed the gem on our development machine, our code won't work when we deploy it to our server unless we either install the same gems on the server or include a copy of the gems with our application.

Including a copy of the gems you need is called *unpacking* them. Now that we've included our specification in the previous Initializer block, we can ask Rails to unpack all of our gems for us:

```
c:\dev\flight> rake gems:unpack
```

This will unpack all the gems your application needs into the vendor/gems directory. You can now deploy your app anywhere you want, without worrying about whether the server has your gems.

Installing Gems on the Server

Sometimes you may choose instead to install a gem once onto your server, instead of copying into each and every Rails app that you deploy there.

Rails can make this task easier:

```
rake gems:install
```

This command will search your Initializer block for all your gem specifications and install fresh copies as if you ran the gem install command yourself for each one.

Knowing how to use and manage gems in your Ruby environment is a skill that all advanced Ruby developers possess. Next, we will learn about another essential utility that Ruby developers can't live without: rake. We've actually been using it throughout this book, but it's time we took a closer look to understand the central role it plays in every Rails application.

Learning More About rake

rake is one of the most commonly used gems in the Ruby community. Originally developed as a Ruby version of make, rake is a first-class citizen in the Rails environment. The rake gem is actually listed as a dependency of Rails: you can't install Rails without also installing rake.

rake provides a simple command-line syntax for running tasks defined in *rakefiles*. Rakefiles, make files, and the Visual Studio build system are all examples of a build system founded on concepts of *tasks* and *dependency chaining*. Building and maintaining software systems often utilize a set of tasks that have some relationship or dependency structure among them.

If you're a Visual Studio user, you know that pressing F5 will start the build cycle. Every .NET project has a natural dependency chain structure. Source code files can depend upon other files. Your project depends upon other projects and ultimately upon the .NET Framework itself. Building a .NET executable is the process of determining what must be done (compiling assemblies, creating help files, building manifests, and so on) according to your project's dependency chain and then performing those tasks so that the right components can be built in the right order.

rake is the Ruby analogy to MSBuild, the Visual Studio build system. rake reads a text file (a rakefile) to understand what needs to be done and in what order. Unlike static languages like C#, Ruby does not have any compile-time dependencies to worry about. So, why would we need a task-dependency build tool like rake?

It turns out that although we don't need to compile our Ruby code into any kind of native executable format, there are in fact many chores that must be done during Rails development that need a task-dependency tool. These chores:

- often rely on another chore or task having been done first, and
- need to be done repeatedly during development and can be automated.

Let's get some concrete examples of how rake is used during Rails development.

Built-in rake Tasks

rake comes with a bunch of prewritten rake tasks to make Rails development easier. To get a listing of all the rake tasks that come with Rails, go to your "root" directory for your application, and enter the following:

```
c:\dev\flight> rake -T
```

You'll see a rather lengthy list of tasks, complete with a brief comment or description of each one.

Tasks are invoked by simply specifying their name on the command line. For example, rake stats will execute the stats task, which provides rudimentary statistics about your code.

More commonly, tasks are grouped into various namespaces. For example, db:create refers to the create task that's in the db task namespace. rake namespaces help avoid name collisions in the same way that .NET namespaces and Ruby module names do.

It is in fact also allowable to call rake without specifying any task explicitly. In this case, the *default task* will be executed. Every rake file designates one task as the default task. We don't have to provide any command-line arguments to execute the default task; thus it is the easiest task to run.

It would therefore behoove a large Ruby project like Rails to think carefully about which task should be the default task. We learned in Section 10.2, *Running Tests with rake*, on page 199 that the default rake task in a Rails project is to run all the tests.

Creating Local Databases

Unless you're using a file-based database system like SQLite, you will need to create your development and test databases before attempting

to run any migrations. You can use a GUI or command-line tool to create the databases for you—or you can use rake!

```
c:\dev> rails -d mysql flight
c:\dev> cd flight
c:\dev\flight> rake db:create:all
```

The db:create:all task reads your config\database.yml file, determines which databases are local databases, and creates them with the given name. Databases are considered local only when:

- The host: line is omitted.
- The host: value is localhost.
- The host: value is 127.0.0.1.

In other words, rake db:create:all will generally create your development and test databases, but you'll have to create the production database on your production server yourself (unless you first deploy your application to the server and then run rake db:create from there).

If you want, you can choose to only create the database for a given Rails environment. For example, to just create the test database, you can use the db:create task and pass the desired RAILS_ENV:

```
rake db:create RAILS_ENV=test
```

Migrating the Database

In Part II we learned how to "migrate" the database whenever we added new models or needed to adjust tables or indexes in some fashion:

```
rake db:migrate
```

You can also specify VERSION= to instruct the migration mechanism to migrate up (or down) to a specific version. For example, this will roll back all migrations, leaving you with an empty database:

```
rake db:migrate VERSION=0
```

Sometimes you'll want to simply rerun the most recent migration:

```
rake db:migrate:redo
```

Sometimes you'll want to roll back all migrations and then run all the migrations again. This can happen toward the end of development, if you choose to consolidate your migrations or rewrite them in some way.

```
rake db:migrate:reset
```

Displaying the Current Database Version

Here's an easy way to display the current database version, in case you're trying to modify your migrations or in case you're wondering whether the database is due for another migration:

```
rake db:version
```

Routes Cheat Sheet

As we learned in Part II, the routes.rb file employs a powerful syntax for describing the mapping between incoming URL structures and your controller actions. When resources are nested inside one another, or the number of resources in your application simply grows too large, it can be hard to keep track of all the named routes and the HTTP verbs that go with them.

rake to the rescue! To see a quick cheat sheet of all your routes, use rake routes:

```
rake routes
```

Cleaning Up the Log File

Finally, here's a task that seems trivial but can be very handy. Your development and test log files can grow out of control if you don't truncate them from time to time. You guessed it, there's even a rake task for doing just that. Here's how we can clear the test.log file:

```
rake log:clear RAILS_ENV=test
```

RAILS_ENV always defaults to "development," so to clear your development log, you can simply run rake log:clear.

Writing Custom rake Tasks

Much in the same way that Visual Studio allows us to insert custom build steps into a project (see Figure 12.1, on the following page), we can also write our own rake tasks. They will be listed in the rake -T task list and are executed like any other task.

Custom rake tasks are useful when any of the following are true:

- You want to write a Ruby script that has easy access to your Rails models, controllers, and views.
- You want to augment or replace one of the built-in rake tasks.
- Your task can be accomplished, at least in part, by calling upon other rake tasks.

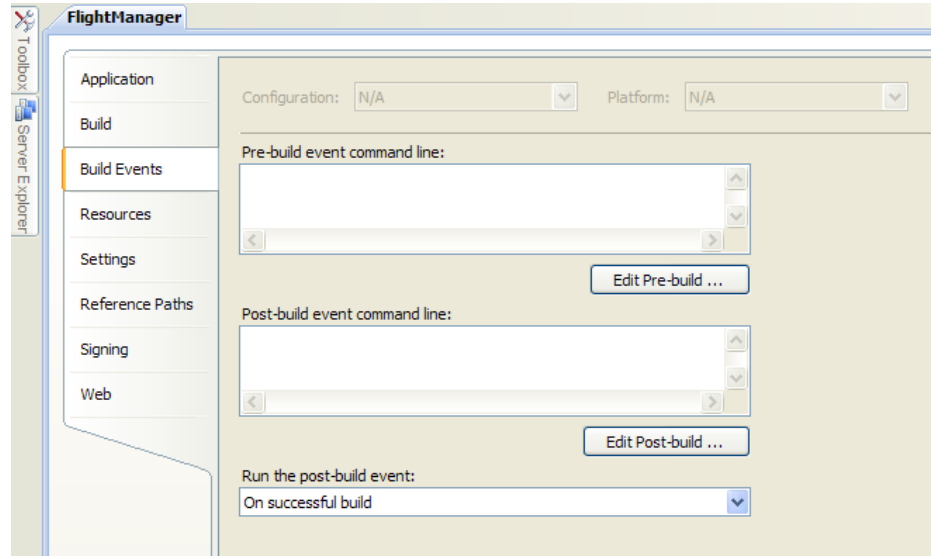


Figure 12.1: Custom build steps for a .NET project

Suppose we'd like to submit our airline on-time performance reports to the FAA on a daily basis. We can choose to use a system utility such as cron on Linux systems or the Scheduler on Windows systems to run our task once a day.

To create our own rake tasks, we simply create a file in the lib/tasks folder and give it an extension of .rake. For example, we could name our file faa.rake. In the file, we define our task like this:

Download finetuning/custom_rake.rb

```
desc "Submit on-time performance to the FAA"
task :ontime_report => :environment do
  compliance_helper = FaaCompliance.new
  compliance_helper.ontime_performance(Date.yesterday)
end
```

This code should look familiar, since it's just Ruby code. However, it may also look a little odd, since we don't seem to be defining any classes or methods. Rakefiles are just Ruby source code. But your rakefile will be called from the rake utility itself, and rake provides a set of predefined methods we can use such as desc() and task(). These methods allow us to define our tasks as simply as possible. The rake system will find the Ruby code inside your task definition when your task is invoked by the rake command line.

rake tasks start by calling the desc() method, which allows us to provide a short description of our task. It is this description that will be shown next to the task name when rake -T is used to get a task list.

We define our task named ontime_report. The important thing to notice is the hash-like syntax we can use to declare that our task depends upon the environment task. The environment task is a built-in Rails task that loads our Rails environment for us—essentially doing the same thing that script/console would do. It loads all our model classes, establishes database connections, and does everything else that Rails application needs to operate.

We then provide a code block for our task. Our task is just two lines in this example, but your task code can have as much or as little Ruby code as you need.

rake tasks can be as simple as the one we've shown here or arbitrarily complex as needed. For more about rake, consult the project's home page at http://rake.rubyforge.org/.

Now that we've peered into the depths of both RubyGems and rake, we are prepared to discuss the strategies available for packaging the Rails framework into a Rails application, a step that's often necessary prior to deployment to a production server.

Distributing Rails with Your Application

The Rails framework code is itself just a handful of Ruby gems installed on your machine. Your application will, by default, always look for the same version of Rails that was present when you first generated your application.

When you need to copy your application code to another workstation or server—perhaps the time has come to deploy your application to a live production web server—you must ensure that the expected version of Rails has been installed on the server and that no one has modified that version in any way, shape, or form.

There are only two ways to accomplish this:

- Install the Rails gems that you need on the server using the gem install command. Place a pack of hungry wolves outside the door to the server room. Finally, revoke administrative privileges from every user account on the box except your own.

Freezing a Rails Application

We will assume that you've agreed that option #1 isn't feasible. Then what do we mean by freezing the application?

Freezing an application binds your application to a specific version of Rails. This line in your config/environment.rb file appears to magically do this:

Download finetuning/gems.rb

```
# Specifies gem version of Rails to use when vendor/rails is not present
RAILS_GEM_VERSION = '2.1.0' unless defined? RAILS_GEM_VERSION
```

This code is generated for you when you use the rails command to create your application. Unless you change the version number specified here, your application must be able to locate that exact version of the Rails gems when it starts up.

Though this might seem sufficient, it is not. Web servers typically serve more than just one application. Since gems are system-wide, servers would have to keep multiple versions of Rails installed indefinitely to support legacy applications. This may not be desirable or even possible, particularly in shared hosting environments.

Freezing an application places an entire copy of the Rails gems into your application's vendor/rails directory. More precisely, it *unpacks* each gem so that the complete source code for Rails will end up in vendor/rails. So when you deploy your application, a copy of whatever Rails version you were using during development will be shipped along with your code. When you start up your application on the web server, there will actually be two choices available: the gems that are installed on the server and the gems that have been copied into your application's vendor/rails directory. Which one will Rails use?

The Rails startup code first looks in the vendor/rails directory for the framework source code. If it finds it, it will use it. Otherwise, your application will instead look for and use the system-wide Rails gems that match the version specification hard-coded in your environment.rb file.

Freezing your application is highly recommended. It insulates your application from unexpected changes in the system-wide Rails gems and guarantees that your production code will be the same as your development code, framework and all.

Enough talk. Open a command prompt, go the root directory of the application, and enter the following:

Edge Rails

The phrase "freezing to edge" means something similar, but distinctly different from, freezing to the latest version of your system-installed gems. Instead, "edge" refers to the latest Rails code that's under active development.

Freezing your application to the most up-to-the-minute version of Rails that resides in the official Rails source code repository can be a great way to learn about the changes are going into Rails. To freeze your code to the edge, run these two rake tasks:

```
rake rails:freeze:edge
```

Living "on the edge" is fun and often instructive. Just remember, you do this at your own risk! Don't ship production code with it unless you're confident of your test coverage.

Note: if you're using a version of Rails prior to 2.1, you'll need to run this command as well:

```
rake rails:update
```

If you now go to c:\dev\flight\vendor\rails, you will see that a copy of the entire Rails framework has been placed there. Open the code, and take a look! It's open source, you know!

Thawing It Out Again

After freezing your application to your current Rails version, you may want to unfreeze it. Perhaps you have updated your Rails gems on your system, and you want to bind your application to the new version. First you'll need to thaw it out:

```
rake rails:unfreeze
```

If you check your vendor directory, you'll see that the rails subfolder has melted away. You're back to "floating with gems." The application will now look for system-installed Rails gems that match the version number specified in your environment.rb file.

Now that we're familiar with how to distribute the framework with our application, we can now embark on an overview of the bigger picture surrounding Rails application deployments.

Deployment Considerations

Way back in Chapter 4, *A Bird's Eye View of Rails*, on page 75, we explained the concept of a "web stack." Rails applications today are, by and large, deployed on Linux servers of one flavor or another. Chances are that the cool new Rails application you build today will need to be deployed into a Linux environment. Although a full tutorial on Rails deployments is beyond the scope of this book, we can at least get a rough idea of what's involved so that you can feel confident about taking the next step even if you haven't used Linux before. For a complete guide to deploying Rails applications on both Linux and Windows servers, see *Deploying Rails Applications: A Step-by-Step Guide* [ZT08].

Choosing a Deployment Host

Regardless of your choice of operating system, there are three choices as to where you can host your application:

- You can own the servers yourself, with your own Internet connection, security setup, power supply backup, and data backup services. This is the best scenario since you have complete control over the hardware and software. But it's also the most expensive, so this choice tends to be a bit unusual for budding Rails entrepreneurs.

- You can get a *virtual private server* with a company that will let you rent an entire server, under some kind of virtualization technology like Xen or VMware. A VPS lets you believe you have full control over a physical server somewhere, when in fact you're managing a virtual machine image of an OS that you choose. Many users might be using your physical machine at the same time you are, but the virtualization ensures fault isolation and some degree of guaranteed performance.

- You can get a *shared hosting* account with a company that will sell you an account on one of their servers. This is the cheapest option, but usually the worst, because you won't have full control over the server and typically have no control over how many other applications are running on your server along with yours. This means memory and bandwidth can be limited. But it can also be a great way to get your feet wet with deploying Rails applications to a real production environment.

Once you've decided on where to host your application, you need to choose an operating system. Let's take a quick tour of the popular choices available today to Rails applications.

Deploying to a Linux Environment

Let's start with the top of the stack, the web server. Most Rails applications are currently deployed under Apache, with nginx (pronounced "engine-x") gaining momentum. Apache 2.2 provides a compelling combination of speed and configurability, making it the runaway leader in web server installations. Litespeed[2] is also a viable alternative to Apache.

Next up, the application server. The winner at the moment here is Mongrel, but there are some others that are proving to be good competition: lighttpd, Litespeed, and mod_rails for Apache.

- Thin[3] is a lightweight, Ruby-specific application server that claims to have slightly better stability under heavy loads.
- mod_rails, aka Passenger,[4] is an Apache module that eliminates the need for a separate application server. Its simple configuration and ease of management make it an appealing alternative in Apache environments.
- Lighttpd[5] (pronounced "lighty") was a forerunner to Mongrel and is still a viable Rails application server.
- FastCGI was the original technique for connecting Apache to a Rails application, but it's no longer considered a reasonable choice for production environments.
- WEBrick comes built into every Rails application...but don't even think about it. WEBrick is fine for your development environment but is not suitable for real-world production environments.

It's important to realize that you'll likely need to deploy more than one instance of your application server. Your web server should also act as load balancer, handing out requests to your application servers. Application servers in this kind of configuration are often referred to as a *cluster*, as in a "Mongrel cluster." Part of the configuration that your web server needs to be concerned with is making sure that requests

2. http://litespeedtech.com/
3. http://code.macournoyer.com/thin/
4. http://www.modrails.com/
5. http://www.lighttpd.net/

are forwarded to the cluster in a manner that evenly distributes the load among them. You'll need to install the mongrel_cluster gem:

```
gem install mongrel_cluster
```

For more information about how to use mongrel_cluster to manage a group of Mongrel servers as a single unit, see the Mongrel home page.[6]

When it comes time to actually deploy your code to the server, we highly recommend using a tool like Capistrano.[7] Capistrano is a free Ruby gem that enables you to automate the nitty-gritty tasks of deployment. Much of the work involved in deploying a web application is error-prone and tedious: getting the latest code from your source repository, deploying the code to one or more physical application server, migrating your database, and finally restarting the application servers. Capistrano is easy to configure and uses simple Ruby scripts to control its operation.

Deploying to a Windows Environment

Deploying to a Windows server is not as straightforward or trouble-free as it is with Linux. However, with some patience and little trial and error, it can be achieved. We'll just try to provide an overview here; refer to *Deploying Rails: A Step-by-Step Guide* [ZT08] for all the gory details on how to get a production-quality Rails environment setup on Windows servers. Your first decision—the choice of web server—will be the biggest. There are mainly three choices here:

- IIS 6 (or higher), with the ISAPI Rewrite module[8]
- Apache 2.2 (or higher) for Windows[9]
- Pen[10] for Windows

IIS 7 is most common in Microsoft-centric organizations but is perhaps the most problematic in getting set up for Rails. However, recent enhancements including a FastCGI handler should make Rails applications easier to deploy. The advent of IronRuby[11] is sure to make the Rails deployment story much easier in the near future (see Section 13.1, *IronRuby*, on page 253). In the meantime, one big hurdle must be overcome: the proxying of load-balanced requests back and forth to a separate application server (and mongrel_service is the only real option currently).

6. http://mongrel.rubyforge.org
7. http:///www.capify.org/
8. http://www.isapirewrite.com/
9. http://www.apache.org/

The IIS Rewrite module will enable you to forward requests from IIS to your application server. But the HTTP responses must go back to IIS for eventual delivery back to the client. This is the job of a "reverse proxy," which is something the IIS Rewrite module cannot do. However, Brian Hogan[12] has written an excellent Rails plug-in, named Reverse Proxy Fix, that you can install into your Rails application that will suffice. You can install this plug-in into your Rails app like this:

```
ruby script/plugin install
    http://svn.napcsweb.com/public/reverse_proxy_fix
```

Apache 2.2 is a good choice for Windows and comes complete with an installation wizard to help ensure a trouble-free installation experience. It can be installed as a Windows service. Perhaps the best reason to choose Apache is the sheer wealth of configuration tutorials and examples that can be found all over the Internet because of its popularity in the Unix/Linux communities.

Finally, Pen is a straightforward web server that is best suited to lower-traffic environments. It is easier to set up and configure than Apache or IIS, but it has fewer features and is not quite as flexible in terms of SSL capabilities and URL rewriting.

Once you have decided on a web server, then choosing the application server is easy: mongrel_service. It's currently the only viable choice as a production-worthy application server on Windows. Just as on Linux, it's recommended that you install the mongrel_cluster gem so that you can set up multiple instances of Mongrel, each listening on different ports, and have your web server spread the requests among them.

Finally, to deploy your application, Capistrano won't work, since it cannot deploy to servers that don't implement the standard Secure Shell (SSH) protocol. It's up to you to determine the best means of deploying your Rails code from your source code repository given the constraints and aspects of your particular server environment.

12. http://www.napcsweb.com/blog/

Chapter 13

Inspired by Rails

When David Heinemeier Hansson released Rails back in 2004, it looked like a promising "little" web application framework to us. We certainly had no idea of the kind of impact Rails would ultimately make on the software development industry. It has transformed many people's impressions of what is possible within web development, given just a little creative (and opinionated) thinking.

On the surface, Rails is simply a framework for creating web applications quickly. But it's much more than that. It also represents a set of ideals for what makes a web application great and a blueprint for how web development "should be done." By not trying to be all things to all people, and instead succeeding at being most things for most people, Rails has carved out a very substantial place in the software development industry.

And because of Rails, many other technologies have been created in the same spirit. Even existing technologies that are part of the Rails ecosystem have experienced an incredible resurgence because of its popularity. Most notably, the Ruby language has certainly been propelled into mainstream recognition because of the hard work done by David and the Rails core team.

In this chapter, we're going to look at a few technologies in the Microsoft universe that have been heavily influenced by Rails, including IronRuby and ASP.NET MVC. Each of these technologies is different in purpose and in style, but undoubtedly, they were all created because their authors were inspired by Rails.

IronRuby

IronRuby[1] is an implementation of Ruby that is written entirely using the .NET Framework. This project runs under Microsoft's Dynamic Language Runtime (DLR), the purpose of which is to allow developers to develop .NET platform applications in their programming language of choice—even if that language is a dynamic one (Ruby, Python, JavaScript) rather than a static one (C#, Java). The addition of the Ruby language to the core .NET Framework speaks volumes about Microsoft and its continuing commitment to both dynamic languages and open source technologies.

IronRuby means Ruby developers can leverage the full power of the .NET Framework and still write code in Ruby. And for .NET developers, it means being able to introduce the flexibility of the Ruby language and frameworks into the everyday workflow. In fact, Rails is able to run—unmodified—on top of IronRuby today, and in the near future, IronRuby on Rails for production-quality apps will be a reality.

Benefits of IronRuby

Why go through all this trouble? Isn't plain ol' Ruby good enough as is? In fact, the official implementation of Ruby, known as Matz's Ruby Interpreter (MRI), is written in C and has always been the subject of criticism by some, usually in regard to performance, threading, and the use of modern programming techniques. Fortunately for us, within the scope of the average web application, these things won't matter to most people. Still, several alternative implementations, in addition to IronRuby, have sprung up, among them JRuby (implemented in Java), Rubinius (a Ruby virtual machine written in Ruby), and YARV (Yet Another Ruby Virtual machine—the official Ruby interpreter in the next version of Ruby, version 1.9).

IronRuby is another attempt to build a better Ruby, but with the goal of interoperability with Microsoft systems. IronRuby may ultimately mean the following:

- It will be easy to talk with .NET assemblies from a Ruby program.
- We will be able to use first-class TDD/BDD tools from the Ruby world like test/unit, shoulda, and RSpec to test our .NET applications.

- When combining IronRuby with ASP.NET MVC (which we'll look at in Section 13.2, *ASP.NET MVC*, on page 257), we will be able to write Rails-like MVC applications with the power of the .NET Framework under the hood.
- We will be able to write Windows Presentation Foundation (WPF) and Silverlight applications with Ruby.
- We will be able to evaluate Ruby code directly in the web client, without the need to go back to the server, when IronRuby runs within Silverlight.
- Ruby will be a first-class citizen within the .NET Framework, and someday, we'll be able to run applications like Ruby on Rails on Windows servers out of the box.

IronRuby in Action

Let's look at a quick example of how IronRuby can talk to a .NET assembly. First, we'll create a couple of simple .NET classes:

Download inspired/Person.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Test
{
    public class Person
    {
        public Person(string name)
        {
            this.Name = name;
        }

        public string Name;
    }
}
```

Download inspired/DemoClass.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Test
{
```

```
    public class DemoClass
    {
        public Int32 Two()
        {
            return 2;
        }

        public static string Marco()
        {
            return "Polo";
        }

        public static List<Int32> SomeNumbers()
        {
            List<Int32> numbers = new List<Int32>();
            numbers.Add(1);
            numbers.Add(2);
            numbers.Add(3);
            return numbers;
        }

        public static List<Person> People()
        {
            List<Person> people = new List<Person>();
            people.Add(new Person("Brian"));
            return people;
        }
    }
}
```

Here, we have a very simple Person class that has a constructor and a single property, Name. Then, we have a DemoClass that contains just one instance method and three class methods. For the purposes of this example, we'll build this assembly and stick it in c:\dev\Test.dll.

After building IronRuby,[2] we'll have access to the ir command, which acts as the IronRuby equivalent of both the ruby and irb commands. Running ir by itself will open an interactive session where you can run IronRuby code interactively, just like irb, as shown in Figure 13.1, on the following page. We can use ir now to exercise our .NET assembly:

```
IronRuby 1.0.0.0 on .NET 2.0.50727.3031
Copyright (c) Microsoft Corporation. All rights reserved.

Note that local variables do not work today in the console.
As a workaround, use globals instead (eg $x = 42 instead of x = 42).
```

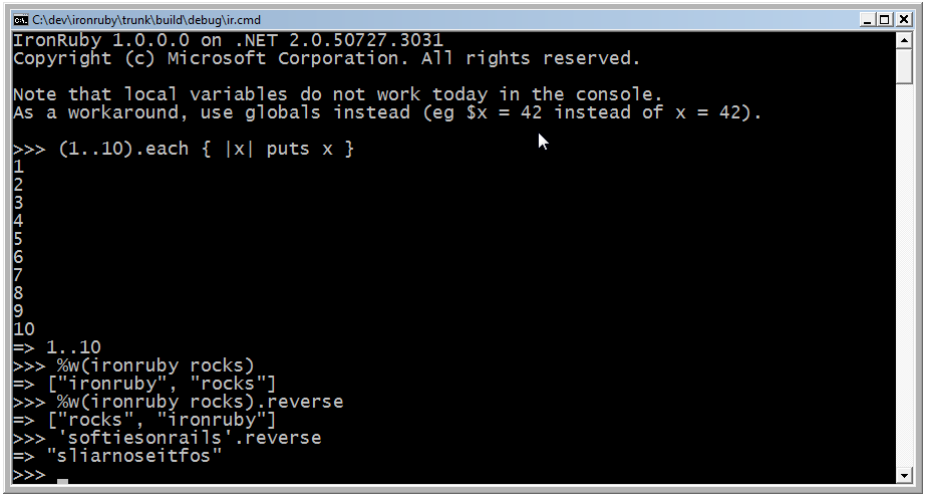


Figure 13.1: Running IronRuby interactively

```
>>> require 'c:\dev\Test.dll'
=> true
>>> Test::DemoClass.new
=> #<Test::DemoClass:0x000005c>
>>> Test::DemoClass.new.Two
=> 2
>>> Test::DemoClass.Marco
=> "Polo"
>>> Test::DemoClass.SomeNumbers
=> [1, 2, 3]
>>> Test::DemoClass.People
=> [#<Test::Person:0x0000060>]
>>> Test::DemoClass.People[0]
=> #<Test::Person:0x0000062>
>>> Test::DemoClass.People[0].name
=> "Brian"
>>>
```

Several interesting things are happening here:

- We can easily reference our .NET assembly from IronRuby simply by require'ing it.
- Once referenced, we can call both instance and class methods in our assembly as if it were any other Ruby class.
- Even though the SomeNumbers and People methods have a return value of a strongly typed generic List in our .NET code, these types are correctly marshaled into the appropriate Ruby type (Array)

Even from this simple example, we can see how IronRuby opens up a world of possibilities for developers wanting to build applications targeting the .NET Framework in Ruby.

ASP.NET MVC

ASP.NET MVC is Microsoft's answer to bringing the MVC pattern that's popular with Rails and other LAMP[3] stack web frameworks to ASP.NET. Currently available as a preview release that runs on top of ASP.NET 3.5, Microsoft hopes to include ASP.NET MVC as part of the standard ASP.NET development environment with its next release. In addition to the separation of responsibilities that the MVC pattern usually follows, ASP.NET MVC also includes the following features:

- Built-in support for testing and mocking.
- A URL rewriting/mapping component that allows for the "pretty" URLs that we're used to in Rails.
- No more postback. Instead of postback, all requests will be routed through controllers that then direct traffic to the appropriate place (sound familiar?).

ASP.NET MVC takes a similar approach to development as Rails, relying on a lot of the same conventions and folder structure we've seen throughout this book. Creating a new ASP.NET MVC project, as shown in Figure 13.2, on the next page, yields a project broken into the subfolders Models, Views, and Controllers, containing code that (unsurprisingly) does essentially the same job as Rails' models, views, and controllers. The directory structure for our project is shown in Figure 13.3, on page 259.

We're going to create a small application that simply lists all the flights. We have an SQL Server table—flights—containing fields for flight number, departure and arrival airports, and departure and arrival times.

The Model and Controller

We're going to use a simple LINQ query to get the flight data we want to display, so the model is going to be a LINQ to SQL object that we're adding to our project; this will give us a FlightDataContext class that we can use to instantiate our data access.

3. Acronym for Linux, Apache, MySQL, and Perl/PHP/Python, typically used to describe any open source bundle of software used to run a website.

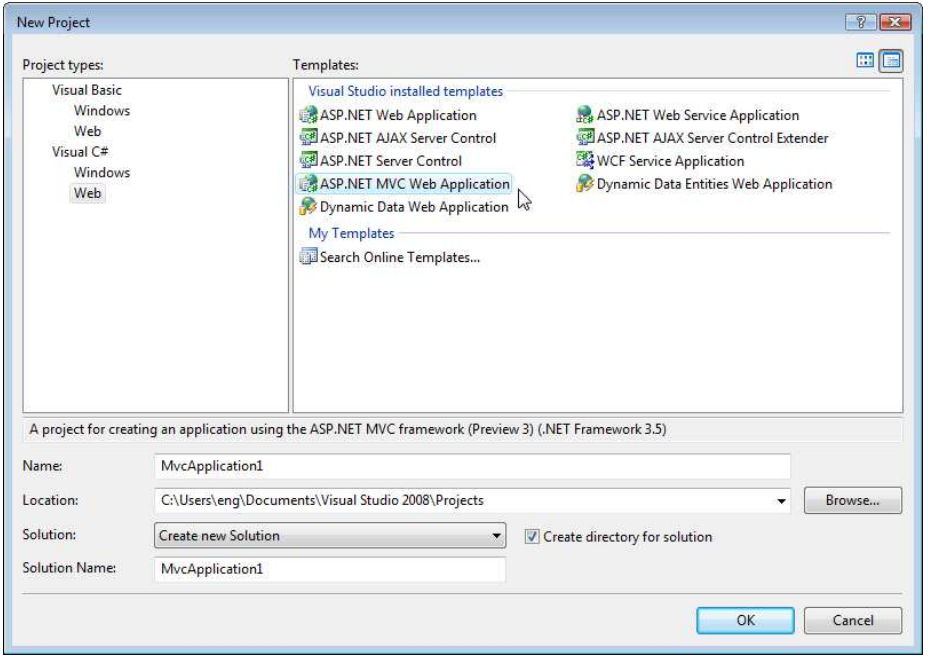


Figure 13.2: Creating a new ASP.NET MVC project

We then use that data in our controller class:

Download inspired/FlightsController.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace FlightMvc
{
    public class FlightsController : Controller
    {
        FlightDataContext flightData = new FlightDataContext();

        public ActionResult Index()
        {
            var flights = flightData.flights.ToList();
            ViewData["Title"] = "All Flights";
            return View(flights);
        }
    }
```

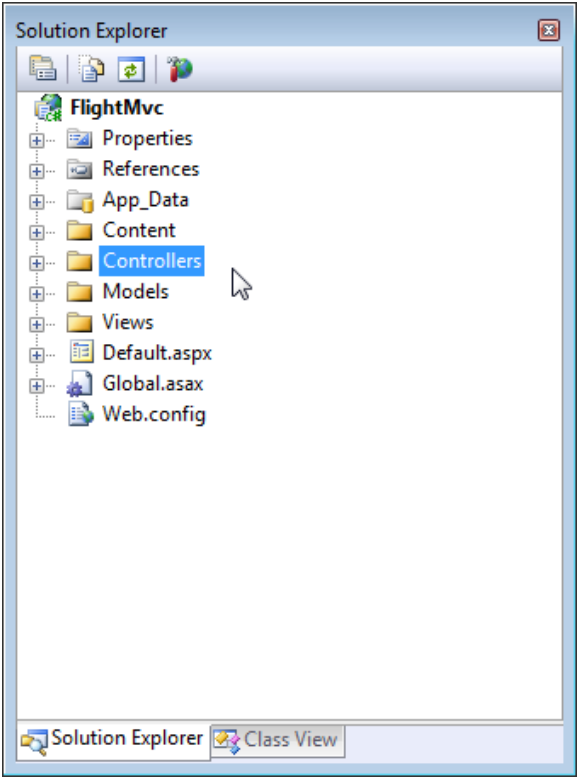


Figure 13.3: ASP.NET MVC project structure

ASP.NET MVC has some conventions of its own going on here. Like Rails, the name of the method, Index(), corresponds to the view that will be rendered—that is, it's expected that an Index.aspx page be present in the Views directory. The Index() method is also expected to return an object of type ActionResult. This, combined with the View method, is ASP.NET MVC's way of exchanging data between the controller and view. The View method indicates the primary object the view is to act upon; we can also pass additional data by populating the ViewData collection.

The View

The view, Index.aspx, then takes this data and, through some templating magic that's similar in style to what we've seen with ERb, displays the end result to the user.

Download inspired/Index.aspx

```
<%@ Page Language="C#" AutoEventWireup="true"
  CodeBehind="Index.aspx.cs"
  Inherits="FlightMvc.Views.Flights.Index" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title></title>
</head>
<body>
    <table border="1">
        <tr>
            <th>Flight Number</th>
            <th>Departs</th>
            <th>To</th>
            <th>Departure Time</th>
            <th>Arrival Time</th>
        </tr>
        <% foreach (var f in ViewData.Model) { %>
            <tr>
                <td><%= f.number %></td>
                <td><%= f.departure_airport %></td>
                <td><%= f.arrival_airport %></td>
                <td><%= f.departs_at %></td>
                <td><%= f.arrives_at %></td>
        <% } %>
    </table>
</body>
</html>
```

Download inspired/Index.aspx.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace FlightMvc.Views.Flights
{
    public partial class Index : ViewPage<List<flight>>
    {
    }
}
```

The view has a code-behind that, in this case, derives from the type of the ActionResult (a List of Flight objects). This gives us a pretty nice feature—the ability to have full IntelliSense on that object in the view.

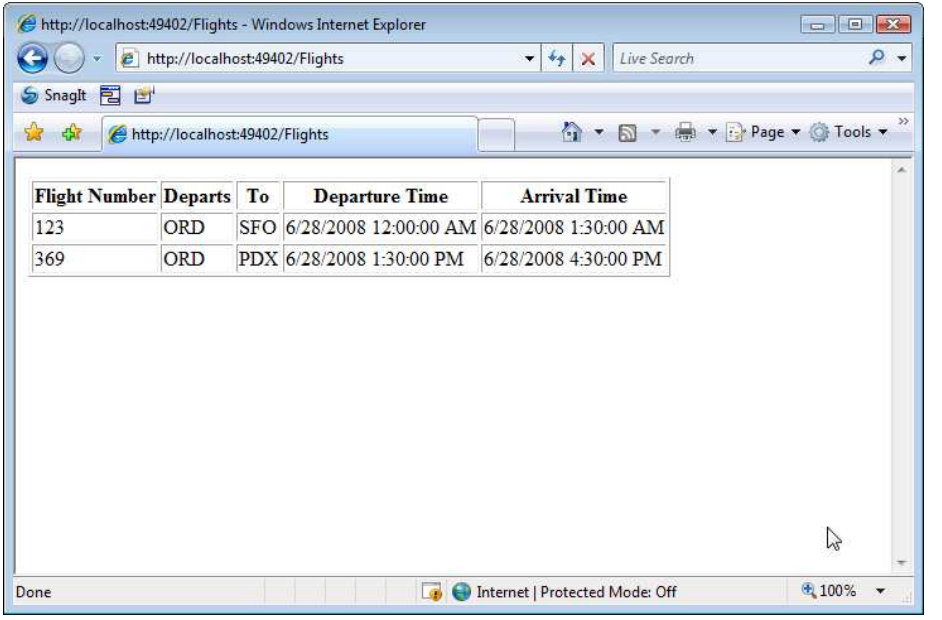


Figure 13.4: ASP.NET MVC application showing flights

The view template itself is pretty straightforward—we loop through each Flight object in the collection and display each record in an HTML table. Our finished product is shown in Figure 13.4.

ASP.NET MVC is clearly Microsoft's response to developers wanting simpler and cleaner solutions for web development than the WebForms approach currently gives us. It's also exciting to think about ASP.NET MVC and IronRuby working together, which essentially gives us a full Ruby-based web development framework that is as easy to use as Rails, but with a powerful .NET back end that is able to talk with existing .NET-based enterprise systems.

Other Open Source Projects

Two other Rails-inspired projects are worth mentioning, namely the Castle Project and Subsonic. Both have been around for some time and continue to have active communities driving further development.

The Castle Project

The Castle Project[4] also brings the MVC pattern to ASP.NET development. The MVC component of the Castle Project (known as *MonoRail*) works alongside Castle's own ActiveRecord module to provide a very Rails-like development experience to .NET web applications. It predates ASP.NET MVC and is very similar in style and spirit.

Subsonic

Subsonic[5] concentrates more on the data access side of things, bringing Rails ActiveRecord-like features to the .NET party. One of the great things about Subsonic is that it's not web-app-specific, so we can easily have an ActiveRecord-like experience within our WinForms applications as well. This includes the ORM mapping, transactions, and migrations that we love in Rails, plus IntelliSense.

The authors of Castle and Subsonic are able to achieve these things by being opinionated and adhering to the "convention over configuration" and DRY mantras that Rails have popularized. Here are some examples of Subsonic's conventions:

- Autoincrementing integers as primary keys
- Singular table names
- Columns named CreatedOn and ModifiedOn gives you the same autoauditing capabilities as the created_at and updated_at fields give us in Rails

These conventions aren't necessarily the same as they are in Rails, but the point is that increased developer productivity has less to do with what the conventions are and more to do with that there are conventions in the first place; it's about deciding upon a set of best practices and then sticking to them, as opposed to forcing the developer to make all the choices.

How About You?

In this chapter, we've taken a look at just a handful of technologies that have come from the Microsoft development community that have been influenced by Rails. It would be safe to say that the invention, and subsequent popularity, of Rails has caused those involved with other

4. http://castleproject.org/

development communities to take a long, hard look at new technologies and methodologies that can increase developer productivity, improve speed to market, and make developing for the Web fun again. In this regard, Rails has certainly succeeded.

Now, how about you? Are you inspired to become proficient in Ruby on Rails?

After getting their feet wet with some Rails programming, some budding Ruby developers get a bit stuck at this point. They're not sure how to get beyond the basics and begin to really excel at Ruby programming, RESTful web design, and real-world deployment, to name just a few areas that are the hallmarks of experienced Rails developers.

Our answer is threefold. First, *just do it*. That sounds obvious, but it's really true. Following on with the examples in this book is a good start. Not sure what a gem command will do? Try it. Wondering how to write a rake task for your project? Start writing one and see what happens. Getting an error page you can't understand? Try the Ruby or Rails Google Groups to get an answer. Hesitant to try a Linux server? Get a cheap shared hosting account and try it. Dive in!

Second, seek to understand everything you do. Reading someone else's Ruby code is one of the best ways to learn. Read the best Ruby and Rails blogs[6] you can find. And don't lift code from the Internet, blindly pasting it into your project hoping it will work. Understand it before you use it.

Finally, give back to the community. Submit a Rails patch, and contribute to the ongoing development of the framework. Everything from big enhancements to tiny documentation fixes are all welcome. Help someone else learn Ruby for the first time, or answer questions on the public Ruby and Rails Google groups. These are just a few of the many meaningful ways you can contribute.

We hope you've been inspired to continue your journey into Rails. Your adventure is just beginning.

Bibliography

[Ful06] Hal Fulton. *The Ruby Way*. Addison-Wesley, Reading, MA, second edition, 2006.

[HT00] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.

[RTH08] Sam Ruby, David Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.

[TFH05] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, second edition, 2005.

[ZT08] Ezra Zygmuntowicz and Bruce Tate. *Deploying Rails Applications: A Step-by-Step Guide*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2008.

Index

Symbols

A

B

C

D

E

F

G

H

O

P

Q

R

S

T

U

V

W

X

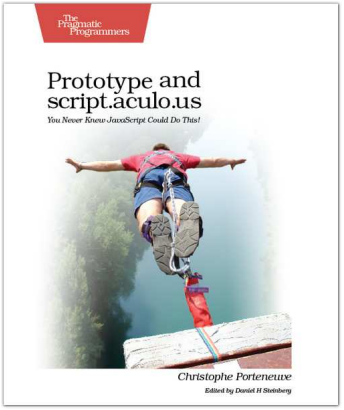
Y

Web 2.0

Welcome to the Web, version 2.0. You need some help to tame the wild technologies out there.

Prototype and script.aculo.us

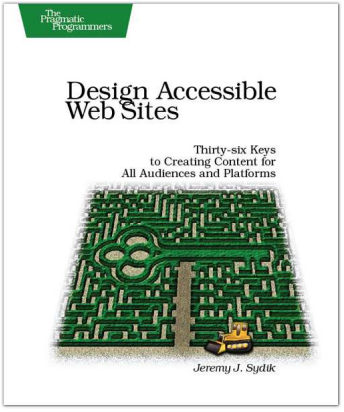
Tired of getting swamped in the nitty-gritty of cross-browser, Web 2.0–grade JavaScript? Get back in the game with Prototype and script.aculo.us, two extremely popular JavaScript libraries that make it a walk in the park. Be it Ajax, drag and drop, autocompletion, advanced visual effects, or many other great features, all you need is to write one or two lines of script that look so good they could almost pass for Ruby code!



Prototype and script.aculo.us: You Never Knew JavaScript Could Do This!
Christophe Porteneuve
(330 pages) ISBN: 1-934356-01-8. $34.95
http://pragprog.com/titles/cppsu

Design Accessible Web Sites

The 2000 U.S. Census revealed that 12% of the population is severely disabled. Sometime in the next two decades, one in five Americans will be older than 65. Section 508 of the Americans with Disabilities Act requires your website to provide *equivalent access* to all potential users. But beyond the law, it is both good manners and good business to make your site accessible to everyone. This book shows you how to design sites that excel for all audiences.



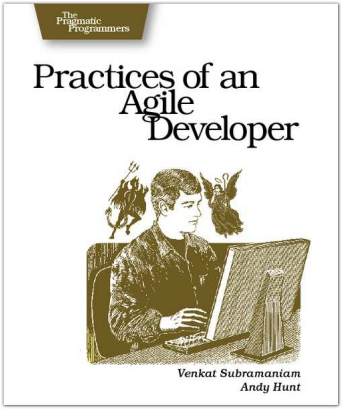
Design Accessible Web Sites: 36 Keys to Creating Content for All Audiences and Platforms
Jeremy Sydik
(304 pages) ISBN: 978-1-9343560-2-9. $34.95
http://pragprog.com/titles/jsaccess

Getting It Done

Start with the habits of an agile developer and use the team practices of successful agile teams, and your project will fly over the finish line.

Practices of an Agile Developer

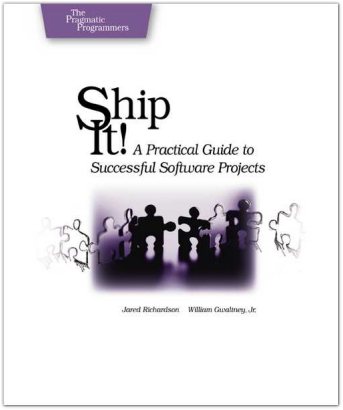
Agility is all about using feedback to respond to change. Learn how to • apply the principles of agility throughout the software development process • establish and maintain an agile working environment • deliver what users really want • use personal agile techniques for better coding and debugging • use effective collaborative techniques for better teamwork • move to an agile approach



Practices of an Agile Developer: Working in the Real World
Venkat Subramaniam and Andy Hunt
(189 pages) ISBN: 0-9745140-8-X. $29.95
http://pragprog.com/titles/pad

Ship It!

Page after page of solid advice, all tried and tested in the real world. This book offers a collection of tips that show you what tools a successful team has to use, and how to use them well. You'll get quick, easy-to-follow advice on modern techniques and when they should be applied. **You need this book if:** • you're frustrated at lack of progress on your project. • you want to make yourself and your team more valuable. • you've looked at methodologies such as Extreme Programming (XP) and felt they were too, well, extreme. • you've looked at the Rational Unified Process (RUP) or CMM/I methods and cringed at the learning curve and costs. • **you need to get software out the door without excuses.**



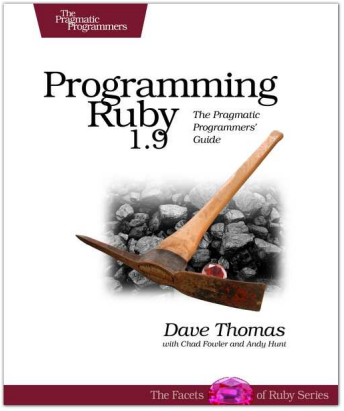
Ship It! A Practical Guide to Successful Software Projects
Jared Richardson and Will Gwaltney
(200 pages) ISBN: 0-9745140-4-7. $29.95
http://pragprog.com/titles/prj

It All Starts Here

If you're programming in Ruby, you need the PickAxe Book: the definitive reference to the Ruby Programming language, now in the revised 3rd Edition for Ruby 1.9.

Programming Ruby 1.9 (The Pickaxe for 1.9)

The Pickaxe book, named for the tool on the cover, is the definitive reference to this highly-regarded language.

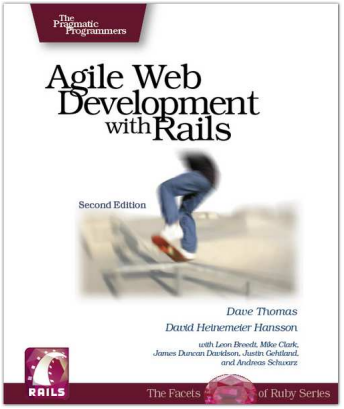


• Up-to-date and expanded for Ruby version 1.9
• Complete documentation of all the built-in classes, modules, and methods • Complete descriptions of all standard libraries • Learn more about Ruby's web tools, unit testing, and programming philosophy

Programming Ruby 1.9: The Pragmatic Programmers' Guide
Dave Thomas with Chad Fowler and Andy Hunt
(992 pages) ISBN: 978-1-9343560-8-1. $49.95
http://pragprog.com/titles/ruby3

Agile Web Development with Rails

Rails is a full-stack, open-source web framework, with integrated support for unit, functional, and integration testing. It enforces good design principles, consistency of code across your team (and across your organization), and proper release management. This is the newly updated Second Edition, which goes beyond the Jolt-award winning first edition with new material on:



• Migrations • RJS templates • Respond_to
• Integration Tests • Additional ActiveRecord features • Another year's worth of Rails best practices

Agile Web Development with Rails: Second Edition
Dave Thomas and David Heinemeier Hansson with Leon Breedt, Mike Clark, James Duncan Davidson, Justin Gehtland, and Andreas Schwarz
(750 pages) ISBN: 0-9776166-3-0. $39.95
http://pragprog.com/titles/rails2

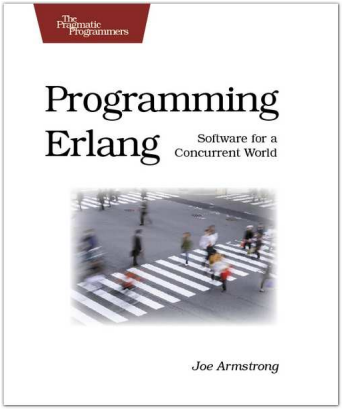
Stuff You Need to Know

From massively concurrent systems to the basics of Ajax, we've got the stuff you need to know.

Programming Erlang

Learn how to write truly concurrent programs—programs that run on dozens or even hundreds of local and remote processors. See how to write high-reliability applications—even in the face of network and hardware failure—using the Erlang programming language.

Programming Erlang: Software for a Concurrent World
Joe Armstrong
(536 pages) ISBN: 1-934356-00-X. $36.95
http://pragprog.com/titles/jaerlang

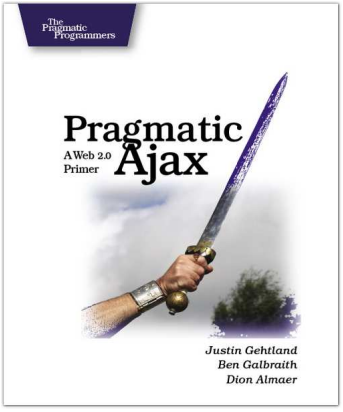


Pragmatic Ajax

Ajax redefines the user experience for web applications, providing compelling user interfaces. Now you can dig deeper into Ajax itself as this book shows you how to make Ajax magic. Explore both the fundamental technologies and the emerging frameworks that make it easy.

From Google Maps to Ajaxified Java, .NET, and Ruby on Rails applications, this Pragmatic guide strips away the mystery and shows you the easy way to make Ajax work for you.

Pragmatic Ajax: A Web 2.0 Primer
Justin Gehtland, Ben Galbraith, Dion Almaer
(296 pages) ISBN: 0-9766940-8-5. $29.95
http://pragprog.com/titles/ajax



The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Rails for .NET Developers' Home Page
http://pragprog.com/titles/cerailn
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/cerailn.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com