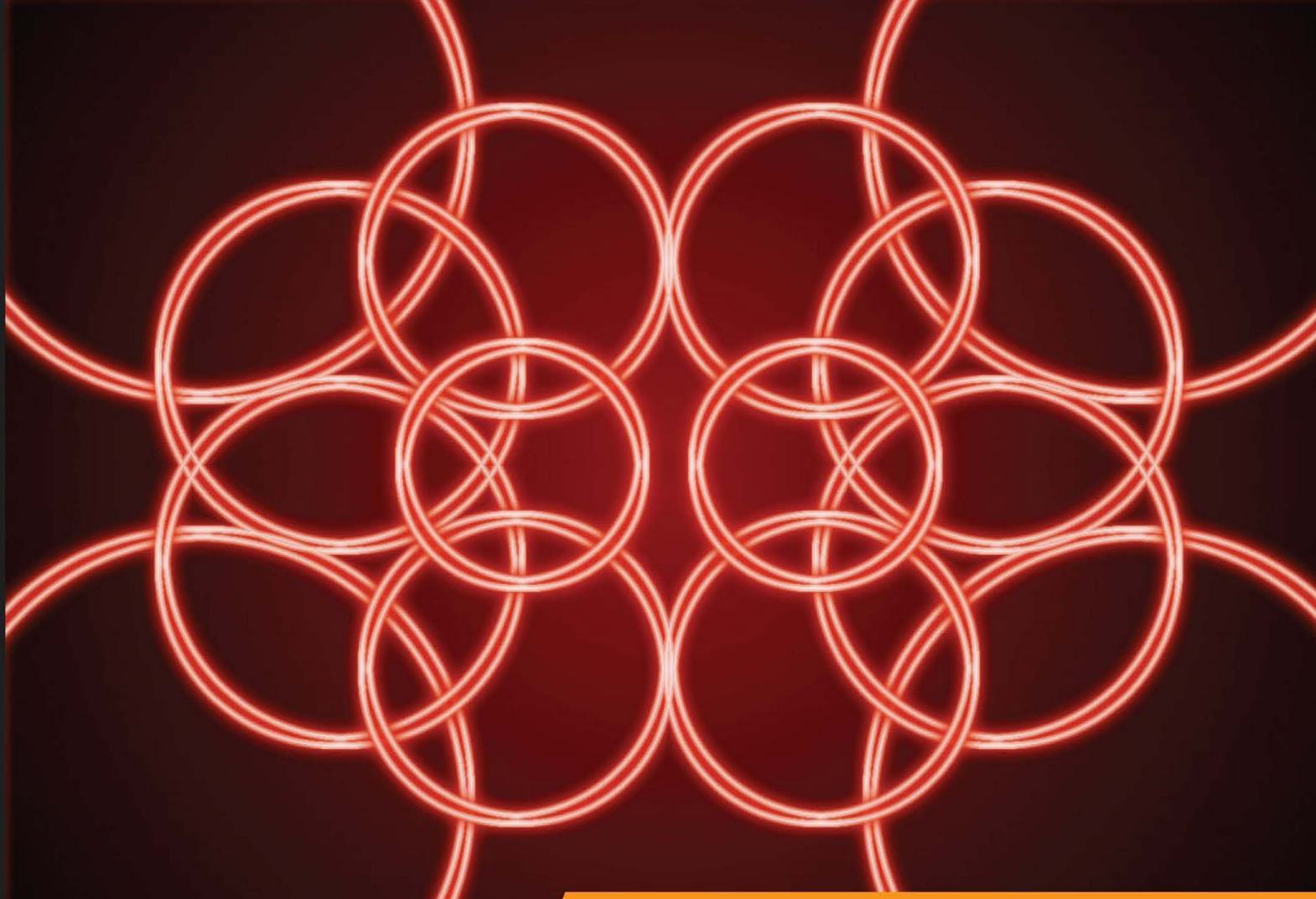


INSTANT

Short | Fast | Focused

RSpec Test-Driven Development How-to

Learn RSpec and redefine your approach toward software development

Charles Feduke

[PACKT] PUBLISHING

Instant RSpec Test-Driven Development How-to

Learn RSpec and redefine your approach towards software development

Charles Feduke

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Instant RSpec Test-Driven Development How-to

First published: June 2013

Production Reference: 1190613

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-522-4

www.packtpub.com

Credits

Author

Charles Feduke

Reviewers

Arvind Janakiram

Bhoor Meena (Raj)

Nola Stowe

Acquisition Editor

Vinay Argekar

Commissioning Editor

Shreerang Deshpande

Technical Editor

Sumedh Patil

Copy Editors

Alfida Paiva

Laxmi Subramanian

Project Coordinator

Joel Goveya

Proofreader

Lawrence A. Herman

Production Coordinator

Conidon Miranda

Cover Work

Conidon Miranda

About the Author

Charles Feduke began developing software in Perl nearly 2 decades ago. He was trapped in the Microsoft platform for far too long and spends his free time these days writing Ruby, learning Scala, and wishing he was really serious about writing C during the 90s.

I'd like to thank my wife, Cathleen, for her patience and understanding while I embark on my endless parade of technological projects. I would also like to thank our daughter, Aleksandra, for keeping us extremely busy.

About the Reviewers

Arvind Janakiram is a software consultant and a practitioner of the TDD process. His experience encompasses a variety of platforms including mobile and the web. He has used RSpec effectively in numerous applications in order to deliver high value products to his clients.

Bhoor Meena (**Raj**) has graduated with a bachelor's and master's degree in Computer Science from the Indian Institute of Technology, Bombay, one of the best engineering colleges in India.

After postgraduation, he joined Rakuten Inc., a leading Japanese e-commerce company, as an application engineer, where he has worked on J2EE, Python, and Ruby.

Currently, he holds a Senior Engineer position in the navigation team and is handling navigation side development of global e-commerce business websites of the Rakuten group, including Rakuten Malaysia, Rakuten Indonesia, and, upcoming, Rakuten Spain, with BDD/TDD and agile methodology.

He is also a development partner in an American startup, `appointmentcare.com`, handling business requirements.

Nola Stowe has been programming since she was 13 years old, starting with a BASIC programming book for her TRS-80. A self-proclaimed "code scientist," she seeks the best tools for the job and the best solutions. She loves writing tests and using test coverage tools to find untested code.

She co-founded DevChix in 2005 when she met a handful of women developers at the ChicagoRuby group. The group's purpose is to inspire and promote women in software development. Currently, DevChix has over 1300 members all over the world and is still growing.

She spends her free time learning other programming languages and writing on her blog, `http://blog.rubygeek.com`. She currently experiments with functional programming languages such as Scala. She was a technical reviewer for *The Rails Way* (first edition).

I'd like to thank my husband, Nick, for doing the mundane things in life (cooking, cleaning, shopping, laundry, and so on) so that I can spend time on what I love doing, programming!

---

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt Publishing offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt Publishing books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt Publishing entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt Publishing
- Copy and paste, print, and bookmark content
- On demand and accessible via web browser

Free Access for Packt Publishing account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Welcome to *Instant RSpec Test-Driven Development How-to*. This short book aims to get you productive with RSpec and **Test-Driven Development** (**TDD**) as quickly as possible.

Test-Driven Development designs a system from the inside out, beginning with domain classes, expanding to controllers, and finally reaching the interface that the customer uses to work with the software.

A test-driven system is easier to maintain because the code written is designed from the ground up to be testable as small units of logic. Your code—when its design is driven by tests—has already been written for reuse (once in its actual execution path and once as the subject of tests). As your experience with writing test driven code grows and you increase the coverage of your unit tests, your confidence in deploying software that you've written will increase remarkably.

What this book covers

Installing RSpec (Simple) gets your environment set up by installing and configuring the RSpec gem.

Preparing the RSpec environment (Simple) covers how to start a new Ruby project and use RSpec for the testing frameworks. It also lays the foundation for the demonstration project written to support the rest of this book.

Refactoring specification and classes (Simple) demonstrates the techniques necessary to support code changes while maintaining high confidence that the code being changed still performs what it needs to.

Making specs more concise (Intermediate) demonstrates idiomatic RSpec code that makes good use of the RSpec Domain Specific Language (DSL).

Handling exceptions (Intermediate) covers how to write specifications that handle failure and exceptional cases.

Working with RSpec matchers (Simple) demonstrates the various matchers that ship with the RSpec library, with code examples and explanations.

Setting up Rails (Intermediate) shows the steps necessary to begin a new Rails project and use RSpec as the testing framework (instead of Test::Unit).

Writing ActiveRecord specifications (Intermediate) reviews how to install Rails, a popular Model View Controller web framework, and how to get started right away writing specifications for the model classes necessary to support most web applications.

Testing Rails routes (Intermediate) shows how to write specifications that exercise routes, an often overlooked area when it comes to testing Rails applications.

Testing Rails controllers (Intermediate) builds off of where the *Writing ActiveRecord specifications* recipe left off by moving up to the controller level.

Stubbing (Intermediate) shows how to use stubs to simulate your runtime environment at test time, ultimately helping you write idiomatic tests that are easy to maintain, fun to write, and fast to run.

Mocking (Intermediate) demonstrates the next step up from the *Stubbing* recipe, where the behavior of your mocked objects can be validated.

Working with JSON (Intermediate) teaches you how to use JavaScript Object Notation (JSON) with Rails and, more importantly, how to do so with RSpec following a Test-Driven Development approach.

Speccing file uploads (Advanced) shows you how to write tests to handle file uploads.

Integration testing with Capybara (Advanced) demonstrates how to use the Capybara integration testing framework from within RSpec to verify the behavior of your application end-to-end.

What you need for this book

This book was written using RSpec on OS X Mountain Lion with Ruby 1.9.3. The first lesson is about getting your environment set up correctly, so you won't need to worry about installing RSpec prior to starting.

Because most Ruby applications run primarily on Linux or Unix operating systems, this book assumes the reader has access to one such OS for working through the code examples. While it is certainly possible to set up a native environment on Windows, I recommend Windows users give Oracle's VirtualBox (a free, open source software project that manages and runs virtual machines), along with one of the many flavors of Linux, an honest try. Who knows, you may thank me for it in the long run!

Who this book is for

This book is for novice or experienced developers seeking to learn how to perform idiomatic Test-Driven Development using Ruby and RSpec. Rails experience is not necessary. In fact if the reader possesses no preexisting Rails knowledge, he or she may find this book a worthwhile primer on getting started with development of Rails applications.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text are shown as follows: "This `Location` class isn't terribly useful."

A block of code is set as follows:

```
describe "Example" do
  subject { { :key1 => "value1", :key2 => "value2" } }
  it "should have a size of 2" do
    subject.size.should == 2
  end
end
```

Any command-line input or output is written as follows:

```
$ mkdir spec/lib
$ touch spec/lib/location_spec.rb
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking on the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt Publishing book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt Publishing books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt Publishing, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Instant RSpec Test-Driven Development How-to

Welcome to *Instant RSpec Test-Driven Development How-to*. This short book aims to get you productive with RSpec and **Test-Driven Development** (**TDD**) as quickly as possible.

In TDD, we follow a simple but important tenet, which is **red**, **green**, **refactor**. Red represents writing test code that exercises code you wish you had and then seeing the test code fail when executed. (It gets the name red from a unit testing software that almost always represents failing tests with the red color.) Green is the stage in the cycle in which you actually write the code to satisfy the test in the simplest possible way. (If you guessed that the unit testing software shows passing tests in green, you're right!) Refactor is the final stage, during which the code written to satisfy the test can be moved, broken apart into smaller units, decomposed into other classes, or subjected to similar techniques in such a way that the tests that drove the development of the code in the first place do not fail. A passing test suite for code that is being refactored is a necessity.

A test-driven system is easier to maintain because the code written is designed from the ground up in order to be testable as small units of logic. Your code—when its design is driven by tests—has already been written for reuse, once in its actual execution path and once as the subject of the tests. As your experience with writing test-driven code grows and you increase the coverage of your unit tests, your confidence in deploying software that you've written will increase remarkably.

Installing RSpec (Simple)

Assuming you already have a Linux or Unix-based system and the Ruby programming language installed, you'll need to install **RSpec**. There are ways to isolate third-party libraries (called gems in Ruby parlance) from one another with tools such as **Ruby Version Manager** (**RVM**), but for the sake of brevity that complexity has been omitted from this book. The exercises herein should work just fine with Ruby 1.8.7 (the default on OS X Mountain Lion) or Ruby 1.9.3 and later.

If you are using Windows, there are options for getting Ruby running in your environment too. Usually you will deploy to a Linux server, so it is often a good idea to develop on a Linux system. Oracle provides **VirtualBox**, which is an open source virtual machine environment and is free of charge, and with it you can install and run any of the mainstream Linux distributions concurrently within Windows.

If you want to remain in a Windows environment while learning RSpec, you can use **RailsInstaller** (`http://railsinstaller.org`). While this isn't a book entirely about Ruby on Rails, RailsInstaller provides the prerequisites for getting Ruby running on your computer.

Getting ready

As most of the commands presented in this book assume a Unix or Linux command line, it's advised to develop on a Windows substitute for the appropriate Command Prompt/ PowerShell commands or install **Cygwin** (`http://cygwin.com`).

How to do it...

1. Install RSpec:

   ```
   $ gem install rspec
   ```

2. Next, prepare the directory structure:

   ```
   $ mkdir lib
   ```

3. Now run RSpec:

   ```
   $ rspec --init
   ```

4. To show what we're testing, we'll change the `.rspec` file generated for us, replacing `progress` with `doc`:

   ```
   --color
   --format doc
   ```

5. We can now run RSpec on our empty `specs` directory and verify we have the gem installed:

```
$ rspec spec
No examples found.


Finished in 0.00004 seconds
0 examples, 0 failures
```

How it works...

We'll work with a sample code base that we will later integrate into a Rails 3 application. The code used for demonstration purposes is a very simple and fictitious system that works with geographical coordinates.

While not strictly necessary at this stage, the `lib` directory is where the code that is written to satisfy the tests will reside.

RSpec creates a `spec` directory, a `spec_helper.rb` file within that directory, and a `.rspec` file in the current directory with sensible defaults.

If you have a problem locating the `.rspec` file, it's because the file is hidden. A command-line editor such as **Vim** has no problem opening a hidden file, for example `vim .rspec`, but using a common dialog box to select a hidden file can be difficult. In OS X, while the **Open** dialog box is shown, you can press *command* + *shift* + . (period) to temporarily show these hidden files.

There's more...

The previous `.rspec` file contains default configuration options, which are applied while executing the `rspec` command-line program. You must execute `rspec` from the directory containing the `.rspec` file if you want the options contained within to be applied. Each line in the `.rspec` file contains a different option. Other options include:

- `--format progress`: Displays progress dots for each executed spec
- `--format doc`: Renders a wordy documentation
- `--format html`: Displays HTML-formatted output, which can be redirected to a file and that file can then be viewed in a web browser like Firefox (see `-o` in the next bullet item)
- `-o`, `--out`: Redirects output to the specified file
- `-c`, `--color`: Use a color in the terminal output (green for passing and red for failing)
- `--fail-fast`: Stops execution when the first failing spec is encountered

These are not all the options. For a complete list check the `rspec` command's help:

```
$ rspec --help
```

Writing a specification

In TDD, we write tests for the code we wish we had, verify that the tests fail, and then implement the code to satisfy the tests. This leads us to a well-designed testable system where monolithic classes are reduced to smaller supporting classes.

This paradigm shift takes some getting used to, but once you've experienced it you'll not want to program without it. Often, you'll find yourself writing tests after you've used TDD to evolve a program. This is encouraged as it is a natural part of increasing test coverage. You'll rarely fall back to your old habits and write code first—when you do, make sure you write accompanying tests to verify the new untested code!

In compiled languages such as Java and C#, the compiler will catch a missing class and refuse to compile—this becomes your very first "verify it fails" test. In a language such as Ruby, it is important to execute the "verify it fails" step for new code to ensure you're not a monkey patching an existing class and altering its behavior unexpectedly.

Preparing the RSpec environment (Simple)

In this section, we'll setup a new project that will be used with RSpec and lay some of the foundation source code for the later sections in this book.

How to do it...

1. First, create a `lib` subdirectory under `spec` and create the `location_spec.rb` file:

   ```
   $ mkdir spec/lib
   $ touch spec/lib/location_spec.rb
   ```

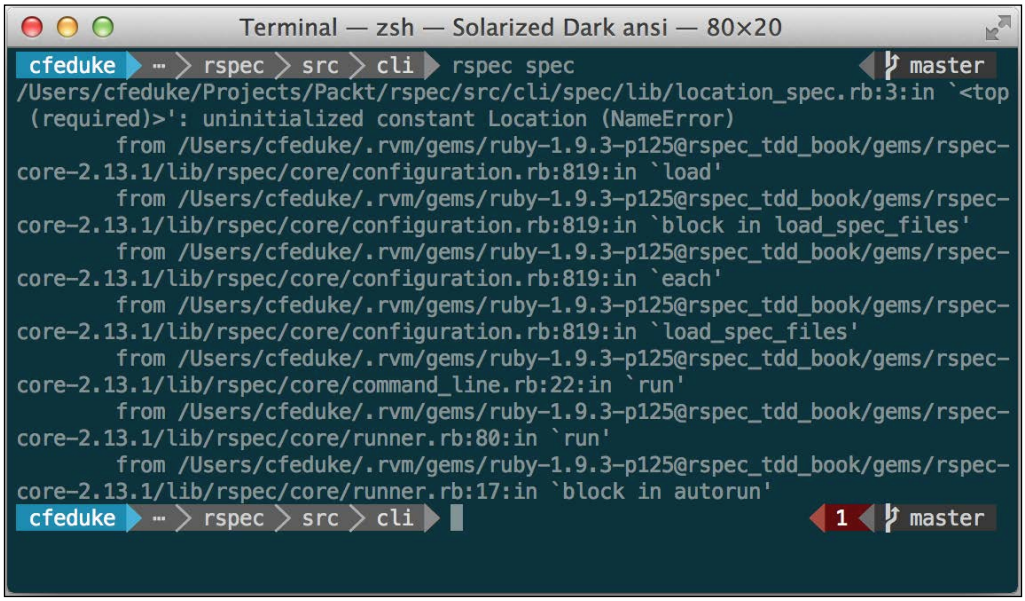
2. We'll begin this process by authoring a specification in `spec/lib/location_spec.rb`:

   ```
   require "spec_helper"
   describe Location do
   end
   ```

3. Save the file and run it from your terminal:

   ```
   $ rspec spec
   ```

4. You'll see a stack trace alerting that there is an uninitialized constant named `Location`:



This is to verify that it fails for the new code and that we're not accidentally trampling on another class provided by Ruby. It is important to understand why a test fails and ensure that it fails for the right reason. If it fails, but you ignore the failure reason, you may be unintentionally introducing a bug that could prove difficult to find.

5. Now, write the code that satisfies the specification in the same `spec/lib/location_spec.rb` file, making sure we define the `Location` class preceding the `describe` block:

```
require "spec_helper"

class Location; end

describe Location do
end
```

6. At this stage, it's often acceptable to define and work on your class in the same file as the spec. The source code won't be delivered to a production environment mixed with specs, and we only take this liberty here for the sake of convenience. Later, we'll refactor the source code into its own file.

7. The next step will be initializing our `Location` class with values for latitude and longitude:

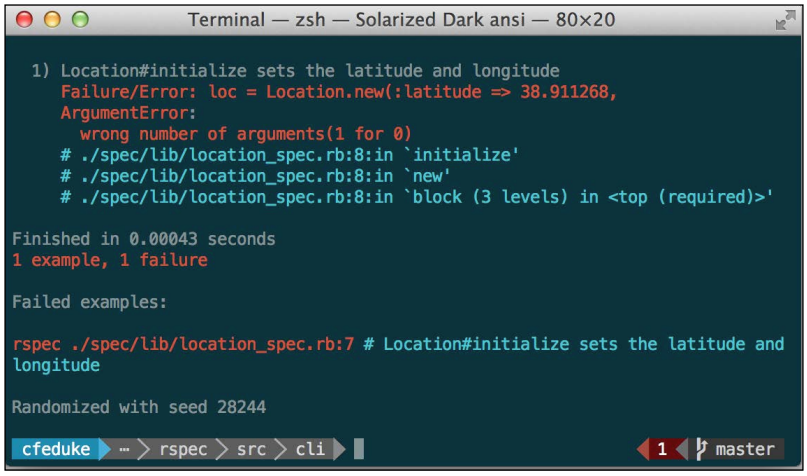
```
describe Location do
  describe "#initialize" do
    it "sets the latitude and longitude" do
      loc = Location.new(:latitude  => 38.911268,
        :longitude => -77.444243)
      loc.latitude.should == 38.911268
      loc.longitude.should == -77.444243
    end
  end
end
```

Downloading the example code

You can download the example code files for all Packt Publishing books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

8. Now run Rspec:

```
$ rspec spec
```

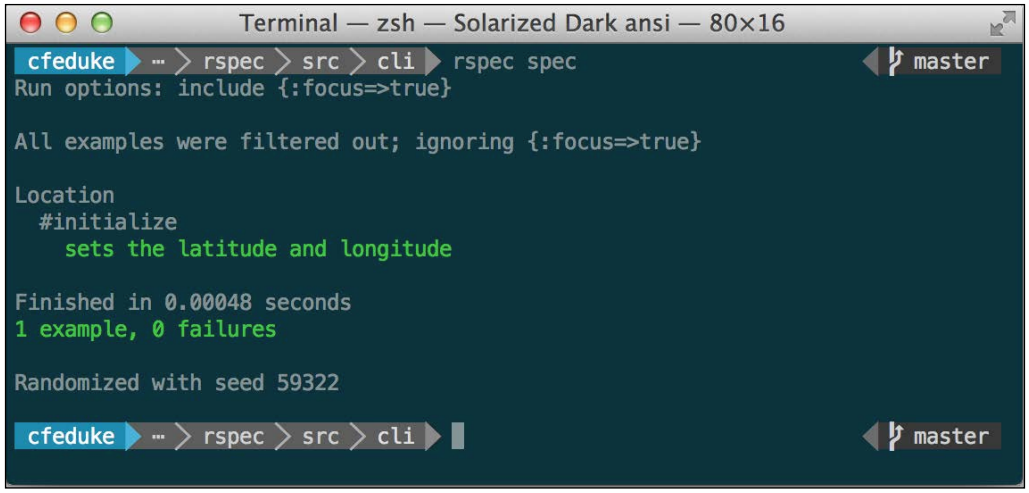


This time the test will fail and you'll see a wordy description as to why it failed—`ArgumentError: wrong number of arguments(1 for 0)`. Next, we'll write the code we wish we had when we wrote the spec, but it is important to only write the minimum code necessary to pass the spec.

9. This next example will be illustrative of a testing extreme—we take the rule for minimum code necessary to heart. Expand the definition of the empty `Location` class that we created within `spec/lib/location_spec.rb` to look like the following code:

```
class Location
  def initialize(args = {}); end
  def latitude
    38.911268
  end
  def longitude
    -77.444243
  end
end
```

10. Running Rspec, we see the specs pass, and our job is done:



11. This `Location` class isn't terribly useful. But, we have followed the rule and only written the code necessary to pass the test.

When you are comfortable with TDD, you can sometimes skip this degenerate step as long as you think about the results of performing such a step. There may be times when you don't need a variable at all (or some other, shorter implementation will work). But if you skip this step you may miss these decisions that reduce implementation complexity. In this particular case, we'll need to author a specification that forces a change in the implementation so that it's more appropriate.

12. Add the following code to the `describe "#initialize"` block:

```
it "sets the latitude to 0 and longitude to 1" do
  loc = Location.new(:latitude => 0, :longitude => 1)
  loc.latitude.should == 0
  loc.longitude.should == 1
end
```

13. When we execute `rspec` again, it fails:

```
1) Location#initialize sets the latitude to 0 and longitude ...
  Failure/Error: loc.latitude.should == 0
    expected: 0
      got: 38.911268 (using ==)
```

14. Now, we update the implementation by performing some small refactoring to the `Location` class itself:

```
class Location
  attr_accessor :latitude, :longitude
  def initialize(args = {})
    self.latitude = args[:latitude]
    self.longitude = args[:longitude]
  end
end
```

15. Make sure to eliminate both the `latitude` and `longitude` methods.

Our code passes, but our specs aren't pretty. They aren't concise; they perform multiple assertions in each specification—we'll fix these problems in a later section—and there are two tests asserting the *same* behavior but with different values. Redundant tests create more code to maintain, so they should be removed whenever possible.

16. Remove the associated spec's `it` block entirely from the file:

```
it "sets the latitude to 0 and longitude to 1" do... end
```

There's more...

Anytime you make a change to a specification or a class under test, you must execute `rspec` to run the test suite. Performing this provides immediate feedback—whether your test fails (you always want it to fail the first time for new code), passes, or you've broken something you weren't expecting to break.

The Ruby community provides tools such as **Guard** for automatically running your specs every time it detects a file system change. It is a good idea to get into the habit of manually running the specs yourself at first, until you've gotten enough experience to know when to save your files in such a way that Guard's output won't interrupt you.

Refactoring specifications and classes (Simple)

Throughout this book, we'll be refactoring the specifications and classes under test to achieve a more concise result. The mantra of TDD is "red, green, refactor," referring to failing tests as red, passing tests as green, and refactoring happening after you have passing tests.

Refactoring is the act of rearranging the code internally without disturbing its external behavior. Having good test coverage is imperative to refactor properly. Without it, you cannot be certain that you have not broken the external behavior, except in the most trivial source code—and even in trivial code this is dangerous!

We've already seen refactoring illustrated in the previous example when we had to refactor our code to pass the specifications, and we then eliminated the redundant specification. There are other reasons to refactor—to follow the "Don't Repeat Yourself" (DRY) principle, improve testability, and help organize source code, among others.

In this section, we'll safely refactor our existing code from the specification into its own class file and verify everything works by running Rspec.

How to do it...

1. Remove the `Location` class from `spec/lib/location_spec.rb` and place it into its own file under `lib/location.rb`
2. Run RSpec:

```
$ rspec
```

You should see the resulting output indicate a failure. During refactoring, we don't often expect to see our tests fail, but it does sometimes happen. In this particular case, it's because Rspec doesn't know about the `lib` directory or where to find the `Resource` class.

3. Remember the empty `spec/spec_helper.rb` file that `rspec --init` generated? It's time to incorporate this file (which is already included at the top of `location_spec.rb`) into our process. You may add the following contents above or below the `Rspec.configure` block found in the file:

```
$:.unshift 'lib'
require 'location'
```

4. Execute `rspec` and you should see that the specs now pass. A successful refactor!

How it works...

`spec_helper.rb` is a file that includes all the code to be executed before running each suite of specifications. For our purposes, we add the `lib` directory to the current load path (`$:` is the symbol for `$LOAD_PATH`, a global variable, which is an array of paths that Ruby searches for any files specified with the `require` keyword) and we then require a file named `location`.

As you'll see later, Rails makes a lot of use of the `spec_helper.rb` file to prepare its environment. Be aware that the more code executed in `spec_helper.rb`, the slower your spec suites will run.

If a spec file doesn't explicitly require `spec_helper.rb`, it will not be automatically required.

Making specs more concise (Intermediate)

So far, we've written specifications that work in the spirit of unit testing, but we're not yet taking advantage of any of the important features of RSpec to make writing tests more fluid. The specs illustrated so far closely resemble unit testing patterns and have multiple assertions in each spec.

How to do it...

1. Refactor our specs in `spec/lib/location_spec.rb` to make them more concise:

```
require "spec_helper"
describe Location do
  describe "#initialize" do
    subject { Location.new(:latitude => 38.911268,
                           :longitude => -77.444243) }
    its (:latitude) { should == 38.911268 }
    its (:longitude) { should == -77.444243 }
  end
end
```

2. While running the spec, you see a clean output because we've separated multiple assertions into their own specifications:

```
Location
  #initialize
    latitude
      should == 38.911268
    longitude
      should == -77.444243

Finished in 0.00058 seconds
2 examples, 0 failures
```

The preceding output requires either the `.rspec` file to contain the `--format doc` line, or when executing rspec in the command line, the `--format doc` argument must be passed. The default output format will print dots (`.`) for passing tests, asterisks (`*`) for pending tests, `E` for errors, and `F` for failures.

3. It is time to add something meatier. As part of our project, we'll want to determine if `Location` is within a certain mile radius of another point.
4. In `spec/lib/location_spec.rb`, we'll write some tests, starting with a new block called `context`. The first spec we want to write is the *happy path* test. Then, we'll write tests to drive out other states. I am going to re-use our `Location` instance for multiple examples, so I'll refactor that into another new construct, a `let` block:

```
require "spec_helper"
describe Location do
  let(:latitude) { 38.911268 }
  let(:longitude) { -77.444243 }
  let(:air_space) { Location.new(:latitude => 38.911268,
    :longitude => -77.444243) }
  describe "#initialize" do
    subject { air_space }
    its (:latitude) { should == latitude }
    its (:longitude) { should == longitude }
  end
end
```

5. Because we've just refactored, we'll execute `rspec` and see the specs pass.

6. Now, let's spec out a `Location#near?` method by writing the code we wish we had:

```
describe "#near?" do
  context "when within the specified radius" do
    subject { air_space.near?(latitude, longitude, 1) }
      it { should be_true }
    end
  end
end
```

7. Running `rspec` now results in failure because there's no `Location#near?` method defined.
8. The following is the naive implementation that passes the test (`in lib/location.rb`):

```
def near?(latitude, longitude, mile_radius)
  true
end
```

9. Now, we can drive a failure case, which will force a real implementation in `spec/lib/location_spec.rb` within the `describe "#near?"` block:

```
context "when outside the specified radius" do
  subject { air_space.near?(latitude * 10, longitude * 10, 1) }
  it { should be_false }
end
```

10. Running the specs now results in the expected failure.
11. The following is a passing implementation of the haversine formula in `lib/location.rb` that satisfies both cases:

```
R = 3_959 # Earth's radius in miles, approx
def near?(lat, long, mile_radius)
  to_radians = Proc.new { |d| d * Math::PI / 180 }
  dist_lat = to_radians.call(lat - self.latitude)
  dist_long = to_radians.call(long - self.longitude)
  lat1 = to_radians.call(self.latitude)
  lat2 = to_radians.call(lat)
  a = Math.sin(dist_lat/2) * Math.sin(dist_lat/2) +
    Math.sin(dist_long/2) * Math.sin(dist_long/2) *
    Math.cos(lat1) * Math.cos(lat2)
  c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a))
  (R * c) <= mile_radius
end
```

12. Refactor both of the previous tests to be more expressive by utilizing predicate matchers:

```
describe "#near?" do
  context "when within the specified radius" do
    subject { air_space }
    it { should be_near(latitude, longitude, 1) }
  end
  context "when outside the specified radius" do
    subject { air_space }
    it { should_not be_near(latitude * 10, longitude * 10, 1) }
  end
end
```

13. Now that we have a passing spec for `#near?`, we can alleviate a problem with our implementation. The `#near?` method is too complicated. It could be a pain to try and maintain this code in future. Refactor for ease of maintenance while ensuring that the specs still pass:

```
R = 3_959 # Earth's radius in miles, approx
def near?(lat, long, mile_radius)
  loc = Location.new(:latitude => lat,
    :longitude => long)
  R * haversine_distance(loc) <= mile_radius
end

private
def to_radians(degrees)
  degrees * Math::PI / 180
end

def haversine_distance(loc)
  dist_lat = to_radians(loc.latitude - self.latitude)
  dist_long = to_radians(loc.longitude - self.longitude)
  lat1 = to_radians(self.latitude)
  lat2 = to_radians(loc.latitude)
  a = Math.sin(dist_lat/2) * Math.sin(dist_lat/2) +
    Math.sin(dist_long/2) * Math.sin(dist_long/2) *
    Math.cos(lat1) * Math.cos(lat2)
  2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a))
end
```

14. Finally, run `rspec` again and see that the tests continue to pass. A successful refactor!

How it works...

The `subject` block takes the return statement of the block—a new instance of `Location` in the previous example—and binds it to a locally scoped variable named `subject`. Subsequent `it` and `its` blocks can refer to that `subject` variable. Furthermore, the `its` blocks implicitly operate on the `subject` variable to produce more concise tests.

Here is an example illustrating how `subject` is used to produce easier-to-read tests:

```
describe "Example" do
  subject { { :key1 => "value1", :key2 => "value2" } }
  it "should have a size of 2" do
    subject.size.should == 2
  end
end
```

We can use `subject` from within the `it` block and this will refer to the anonymous hash returned by the `subject` block. In the preceding test, we could have been more concise with an `its` block:

```
its (:size) { should == 2 }
```

We're not limited to just sending symbols to an `its` block—we can use strings too:

```
its ('size') { should == 2 }
```

When there is an attribute of `subject` you want to assert but the value cannot easily be turned into a valid Ruby symbol, you'll need to use a string. This string is not evaluated as Ruby code; it's only evaluated against the subject under test as a method of that class.

Hashes, in particular, allow you to define an anonymous array with the key value to assert the value for that key:

```
its ([:key1]) { should == "value1" }
```

There's more...

In the previous code examples, another block known as the `context` block was presented. The `context` block is a grouping mechanism for associating tests. For example, you may have a conditional branch in your code that changes the outputs of a method. Here, you may use two `context` blocks, one for a value and the second for another value. In our example, we're separating the happy path (when a given point is within the specified mile radius) from the alternative (when a given point is outside the specified mile radius). `context` is a useful construct that allows you to declare `let` and other blocks within it, and those blocks apply only for the scope of the containing `context`.

Handling exceptions (Intermediate)

Sometimes, when code doesn't follow the prescribed happy path, we encounter exceptional situations that must be handled as special cases, whereas not every non-happy path execution is exceptional. We should, however, specify and verify execution paths that do lead to exceptional situations so we are sure that our code handles them properly.

What would happen if a negative value was passed for the mile radius in the `near` method? It would never return `true` for one, but we shouldn't pass that sort of value in the first place; Instead, we should signal back to the call site that a negative radius is never permitted.

How to do it...

1. In `spec/lib/location_spec.rb`, run `rspec` and verify that the specification fails:

```
context "when a negative radius is used" do
  it "raises an error" do
    expect { air_space.near?(latitude, longitude, -1) }
      .to raise_error ArgumentError
  end
end
```

2. Implement the specified behavior in `lib/location.rb`:

```
def near?(lat, long, mile_radius)
  raise ArgumentError unless mile_radius >= 0
  # remainder of method omitted
end
```

3. Run `rspec` and verify that the specification now passes.

There's more...

In other programming languages, a common anti-pattern is to use exceptions to manage control flow. In Ruby, the `throw/catch` keywords serve this purpose by throwing a symbol that the code further up the stack can catch, albeit without a stack trace. Although not strictly used to replace exception raising and handling, `throw/catch` is a useful control structure construct that can be verified by RSpec.

Working with RSpec matchers (Simple)

RSpec matchers can be combined with either `should` or `should_not` as a part of validation. We'll step back from the sample application for this section and review each matcher class and how to use it.

How to do it...

1. The various ways to determine equality and inequality are:

```
1  describe "Equal" do
2    let(:address) { "123 Main Street" }
3    subject { "123 Main Street" }
4    it { should eq '123 Main Street' }
5    it { should == "123 Main Street" }
6    it { should_not == "789 Any Circle" }
7    it { should_not be(address) } # object equality
8    it { should_not equal(address) } # object equality too
9    it { should eql(address) }
10   it { should == address }
11 end
```

Lines five and six in the preceding code snippet should not be a surprise; they compare the value of `subject` with string literals that are a match (in terms of string comparison) or not a match, respectively. Lines seven and eight may surprise you if you're not accustomed to Ruby's object equality comparisons. Although the values are indeed the same, the actual references to those values are not the same, and `be` and `equal` test for object equality. Lines eight and nine bring us back home by using string comparison for `eql` and `==` (such as line four) with a variable reference.

2. Comparisons allow us to verify greater than and less than conditions:

```
1  describe "Comparisons" do
2    subject { 42 }
3    it { should be > 41 }
4    it { should be >= 42 }
5    it { should be <= 42 }
6    it { should be < 43 }
7  end
```

3. In RSpec, there is no restriction specifying that only numbers may be compared; many other types are candidates for comparison. You would find yourself comparing floating point numbers or checking whether a value is within an acceptable threshold:

```
1  describe "Floating Comparison" do
2    subject { 3.141_592_653_5 }
3    it { should be_within(0.000_2).of(3.141_590) }
4  end
```

4. Regular expression comparisons are a convenient and powerful way of validating portions of text, and are especially noteworthy for their use in validating Rails view specs:

```
1  describe "Regular Expression Comparison" do
2    subject { "this is a block of text" }
3    it { should match(/text$/) }
4    it { should =~ /\bblock\b/ }
5  end
```

5. Boolean tests determine the truthiness of a variable or statement:

```
1  describe "Boolean" do
2    subject { "non-nil is true" }
3    it { should be_true }
4    it { should_not be_false }
5  end
```

It should be noted that in Ruby any non-`nil` value is `true` and any `nil` or false value is `false`.

6. RSpec performs some magic by dynamically creating matchers for any methods on a class that either begin with the word `has` or end with a question mark. These dynamically created methods are named `have_method_name` or `be_method_name` respectively and are called predicates:

```
1  describe "Predicate" do
2    subject { { :a => 1, :b => 2 } }
3    it { should have_key(:a) } # has_key?(:a)
4    it { should_not be_empty } # empty?
5  end
```

7. Determining whether a given value is contained by a collection is done with the `include` matcher. Remember that a string is a collection of characters, so `include` may be used with substrings, shown as follows:

```
1  describe "Collections" do
2    subject { ["text one", "text two"] }
3    it { should include "text two" }
4    its (:first) { should include "ext" }
5  end
```

8. Testing for a particular class or superclass has limited applicability in Ruby, but it can be done as shown in the following code:

```
1  describe "Class" do
2    subject { 42 }
3    it { should be_instance_of Fixnum }
4    it { should be_kind_of Integer } # Fixnum > Integer
5  end
```

9. Because Ruby doesn't have interfaces or abstract classes, it can become important to verify that a given class adheres to a specific contract:

```
1  describe "Contract Validation" do
2    subject { Resource.new }
3    it { should respond_to :near? }
4  end
```

10. Unlike `throw` in other languages, Ruby's `throw` and `catch` are used as control structures and may have an associated symbol and optional payload.

```
1  describe "Throws" do
2    subject { Proc.new { throw :some_symbol, "x" } }
3    it "should throw some_symbol" do
4      expect { subject.call }.to throw_symbol
5      expect { subject.call }.to throw_symbol(:some_symbol)
6      expect { subject.call }.to throw_symbol(:some_symbol, "x")
7    end
8  end
```

11. Raising errors is similar to the `throw` statements, but are used for error situations and not for the control flow.

```
1  describe "Errors" do
2    subject { Proc.new { raise RuntimeError.new("x") } }
3    it "should raise an exception" do
4      expect { subject.call }.to raise_error
5      expect { subject.call }.to raise_error(RuntimeError)
6      expect { subject.call }.to raise_error(RuntimeError, 'x')
7      expect { subject.call }.to raise_error('x')
8    end
9  end
```

Setting up Rails (Intermediate)

In this section you will install and set up rails for `ActiveRecord` specifications.

How to do it...

1. To install Rails, if you do not already have it installed, execute:

   ```
   $ gem install rails
   ```

This will download and install the most recent stable version of Rails into your system. The proceeding examples use Rails 3.2.13, though Rails Version 3.0 and later should work too.

2. To create a new Rails project for integrating our existing code into, execute the following code:

   ```
   $ rails new geo_pictures --skip-test-unit
   ```

3. By passing the `--skip-test-unit` option, the necessary folders that would be created to support the built-in Ruby `Test::Unit` framework are omitted. Because we're using RSpec to perform the same role as `Test::Unit`, an additional testing framework would be redundant.
4. Edit the `Gemfile` in the `geo_pictures` directory and add the following line:

   ```
   gem 'rspec-rails', :group => [:test, :development]
   ```

5. Once the file is saved, update your installed gem dependencies from the `geo_pictures` directory by executing the following command:

   ```
   $ bundle install
   ```

6. Finally, to prepare RSpec for use with Rails, execute:

   ```
   $ rails generate rspec:install
   ```

7. The preceding command creates a spec directory, a new `.rspec` file with just the `--color` option set, and a default Rails-friendly `spec/spec_helper.rb` file.

Writing ActiveRecord specifications (Intermediate)

Continuing with the previous recipe, we will see how to write `ActiveRecord` specifications here.

How to do it...

1. Create a model to represent the `Location` class that we've already developed:

   ```
   $ rails g model location latitude:decimal longitude:decimal
   ```

   `g` is the shortened form of generate.

2. Create and migrate the environments:

   ```
   $ rake db:create:all && rake db:migrate && rake db:test:clone
   ```

3. This creates databases and then migrates the schema of the default development environment. In addition, it also clones the development database structure to the test database. Using `&&` between commands will run each command in succession, provided that the previous command does not fail. You can alternatively enter each command at a separate prompt.

4. Now, execute `rspec` to reveal that `spec/models/location_spec.rb` has a pending example.

5. Move the previous implementation of `location_spec` to this new (`spec/models/location_spec.rb`) file by replacing the contents of the automatically generated file with the contents of the file from our existing spec implementation. (Do not move the previous implementation of `Location` from `lib` to `models`; we'll rebuild this piecemeal as the need arises.)

6. Run `rspec` to see where we stand.

7. Of the four examples, two are failing because the new `Location` model class that Rails generated for us has no implementation of the `#near?` method. We can solve this problem by copying the `#near?` method and its associated private methods from the existing `lib/location.rb` file to the model class at `app/models/location.rb`:

   ```
   class Location < ActiveRecord::Base
     attr_accessible :latitude, :longitude
     R = 3_959 # Earth's radius in mi, approx
     def near?(lat, long, mile_radius)
   ```

```
    # omitted
  end
private
  def to_radians(degrees)
    # omitted
  end
  def haversine_distance(loc)
    # omitted
  end
end
```

Executing `rspec` reveals that all the specs now pass.

Next, we'll want to verify the validation behaviors of our model. We'll spec out the `latitude` attribute's validations first and then use a loop construct to have the same specs applied to the `longitude` attribute's validation behavior.

In this case, the behavior that we want to drive out is that of the invalid `Location` model instances returning `false` for `Location#valid?` invocations. A valid `Location` class is one who's `latitude` and `longitude` attributes are present and are numeric. But, we can't settle with just a Boolean result for `Location#valid?`; the actual error message itself is important and therefore must be verified.

8. In `spec/models/location_spec.rb` add the following code:

```
describe "validations" do
  before { subject.valid? }
  [ :latitude ].each do |coordinate|
    context "when #{coordinate} is nil" do
      subject { Location.new(coordinate => nil) }
      it "shouldn't allow blank #{coordinate}" do
        expect(subject.errors_on(coordinate))
          .to include("can't be blank")
      end
    end
  end
end
```

9. To get the spec passing, add the presence validator to `app/models/location_spec`:

```
class Location < ActiveRecord::Base
  validates :latitude, :presence => true
  # remainder omitted
```

10. Because `latitude` and `longitude` will behave in the same way, the previous code will set up a loop outside the `context` block. We can make use of this by changing the line in `location_spec` from:

    ```
    [ :latitude ].each do |coordinate|
    ```

    to:

    ```
    [ :latitude, :longitude ].each do |coordinate|
    ```

    and the line in the `Location` model from:

    ```
    validates :latitude, :presence => true
    ```

    to:

    ```
    validates :latitude, :longitude, :presence => true
    ```

11. Fast-forwarding and compacting important TDD steps together, the remaining behavior is driven out as follows:

    ```
    describe "validations" do
      before { subject.valid? }
       [ :latitude, :longitude ].each do | coordinate|
        context "when #{coordinate} is nil" do
          subject { Location.new(coordinate => nil) }
          it "shouldn't allow blank #{coordinate}" do
            expect(subject.errors_on(coordinate))
              .to include("can't be blank")
          end
        end
        context "when #{coordinate} isn't numeric" do
          subject { Location.new(coordinate => 'forty-two') }
          it "shouldn't allow non-numeric #{coordinate}" do
            expect(subject.errors_on(coordinate))
              .to include("is not a number")
          end
        end
        context "when #{coordinate} is an acceptable value" do
          subject { Location.new(coordinate => 42.0) }
          it "should have no errors for #{coordinate}" do
            expect(subject).to have(0).errors_on(coordinate)
          end
        end
      end
    end
    ```

12. The completed implementation in the `Location` model for validation is:

```
class Location < ActiveRecord::Base
  validates :latitude, :longitude,
    :presence => true,
    :numericality => true
  # remainder omitted
```

How it works...

Our `Location` specs passed, although we never explicitly recreated the constructor (`#initialize`) method in our new `Location` model class. The spec passes because `ActiveRecord::Base` provides a constructor that accepts a hash as its argument. The hash is in turn used to assign values to the attributes of the class so there is no need to explicitly write our own constructor in order get our specs passing.

`ActiveRecord::Base` also yields to a block, which makes the following code valid:

```
let(:air_space) do
  Location.new do |loc|
    loc.latitude = 38.911268
    loc.longitude = -77.444243
  end
  loc.save # store it in the database
  loc      # assign loc to variable air_space
end
```

While creating tests, use the `hash` or `block` method. This results in the easiest way to read and maintain source code. Typically, this means that as the number of attributes grow, the `block` method becomes favored.

There's more...

Let's review the `Gemfile` and the changes made to the `spec_helper.rb` file where we migrated our existing code to Rails.

Gemfile environments

When we added rspec-rails to the `Gemfile`, we restricted its inclusion to a couple of groups: test and development. Typically, deploying a Rails application to production does not include either test or development configurations, and as a result the gems that support those environments don't go along for the ride. The need for rspec-rails to be available to the test environment is self explanatory, but why do we need to include it with the development environment? Including rspec-rails at development time (the default environment when the `RAILS_ENV` environment variable is not specified) hooks the Rails' generate commands, so that anytime a model, a controller, or a Rails-specific class (for example, `rails generate controller Person`) is created, an accompanying empty spec file is generated along with it.

spec_helper.rb

In the previous `spec_helper.rb` file for the `Location` class that lived in the `lib/` directory, we had to explicitly add `lib/` to `$LOAD_PATH` and even include the `location` file directly. How come we don't need to perform this step now?

Rails automatically loads all `*.rb` files that it finds under the `app/` directory as part of its startup (in addition to `config/initializers`, which is lexicographically enumerated first). Because the `Location` class lives under `app/models`, and the `spec/models/locations_spec.rb` file requires the `spec/spec_helper.rb` file, the Rails environment is loaded as part of the `rspec` execution.

Doesn't this mean that tests that rely on the Rails infrastructure are inherently slow? Yes, with all that additional overhead, even the simplest tests can take longer than you may expect to execute. The way to mitigate this problem is to write your specs and implementation independent of Rails whenever possible, and stub Rails' behavior whenever necessary.

rake spec

Instead of running `rspec` to execute your test suite, you could optionally run `rake spec`. There is a problem with doing this that may not be readily apparent, at least in Rails 3. When you execute `rake spec` from the command line, the Rails environment is loaded to support the rake task with the development profile (`RAILS_ENV=dev`). Then, in order to run as a proper test environment, `rake spec` must spawn off `rspec` in a separate child process with the test profile (`RAILS_ENV=test`). This means that the Rails environment gets loaded twice and could take considerably more time than executing `rspec` by itself.

Testing Rails routes (Intermediate)

In this recipe, we'll see how to properly test drive and verify routes in a Rails application. Routes determine which controller handles a particular request based on the data provided by the client's browser.

Getting ready

To spec controller routes, we'll first need an actual controller. Without it, Rails won't properly route requests and we won't be able to validate our specs:

```
$ rails g controller locations --no-helper
```

As a part of controller generation, if the `--no-helper` option wasn't passed, an empty helper and spec file with pending examples would be created.

How to do it...

1. First, we'll need to register routes to the controller. This is not only easy to do but can be test-driven as well. Create a new file, `spec/routing/routes_spec.rb`:

```
$ mkdir -p spec/routing
$ touch spec/routing/routes_spec.rb
```

2. Now, drive the registration of all the routes together because the Rails command to configure them, in this case, is a one-liner:

```
require 'spec_helper'
describe "Routes" do
  describe "LocationsController" do
    it "routes get new" do
      { :get => '/locations/new' }.should route_to(
        :controller => 'locations',
        :action     => 'new'
      )
    end
    it "routes post create" do
      { :post => 'locations' }.should route_to(
        :controller => 'locations',
        :action     => 'create'
      )
    end
    it "routes get index" do
      { :get => 'locations' }.should route_to(
        :controller => 'locations',
        :action     => 'index'
      )
    end
    it "routes get show" do
      { :get => 'locations/42' }.should route_to(
        :controller => 'locations',
        :action     => 'show',
        :id         => '42'
      )
    end
    it "routes delete destroy" do
```

```
        { :delete => 'locations/42' }.should route_to(
          :controller => 'locations',
          :action     => 'destroy',
          :id         => '42'
        )
      end
      it "does not route get edit" do
        { :get => 'locations/42/edit' }.should_not be_routable
      end
      it "does not route put update" do
        { :put => 'locations/42' }.should_not be_routable
      end
    end
  end
```

A lot of routes are being verified in the previous code. This is acceptable, although we end up essentially verifying the framework code, which isn't the real goal of TDD. But if you're not familiar with Rails routing, using tests to drive this configuration is a great way to arrive at a working implementation.

3. With those failing specs, we can open up `config/routes.rb` and add the one-liner that makes them all pass (within the `routes.draw` block):

   ```
   resources :locations, :except => [:edit, :update]
   ```

4. By default, resources will add all the routes that we tested for and exclude the ones that we marked as `should_not be_routable`. Run `rspec` again and you'll see everything pass, meaning that the routing configuration is good.

How it works...

Test driving routes can often be a lot of work for a little reward. If you are or become comfortable with Rails routing configuration, it may only be necessary to test drive for any particularly thorny configurations—nested resources or URLs that include nonstandard elements such as additional variables. For example, see this spec:

```
it "routes to cars/make/model/year" do
  { :get => "cars/toyota/corolla/1994" }.should route_to(
    :controller => "cars",
    :action     => "show",
    :make       => "toyota",
    :model      => "corolla",
    :year       => "1994"
  )
end
```

Verifies this routing statement:

```
match "cars/:make/:model/:year" => "cars#show"
```

It is important to note that in the preceding code, even though `1994` could be converted to an integer when it's coming in as part of an URL, Rails correctly treats it like a string. This could lead to frustration if you had `year => 1994` instead of `year => "1994"`.

A pitfall while speccing routes is that when there is no matching controller under `app/controllers`, a route will not be routable. This can be especially troublesome if you forget to pluralize your controller name (for example, `LocationController`) and your specs are for routing to controller `locations`. RSpec will print that no route matches the expected route, which is almost not true because executing rake routes will print the expected route as a valid route; it' s just that no actual controller matches the requested controller.

There's more...

What if there's a need to validate that a particular route is available only over a given protocol, such as HTTPS? You can add a `constraints` section to a route in `config/routes.rb` and validate it through a spec as follows:

```
it "allows HTTPS for history" do
  { :get => 'https://test.host/locations/42/history' }
    .should route_to(:controller => 'locations',
      :action     => 'history',
      :id         => '42'
     )
end
it "does not route HTTP for history" do
  { :get => 'http://test.host/locations/42/history' }
    .should_not be_routable
end
```

And the configuration to match the spec (in `config/routes.rb`) is as follows:

```
match 'locations/:id/history' =>
  'locations#history',
  :constraints => { :protocol => "https://" }
```

Why this explicit `test.host` value? It turns out that, unlike other constraints such as the `format` protocol, this has to be tested as part of the URL. During test time, the virtual hostname `test.host` is substituted for an actual host (such as `localhost`) unless the configuration is changed to provide another host name.

Testing Rails controllers (Intermediate)

In this recipe we'll see how to test drive the development of a Rails controller.

How to do it...

1. Begin with `LocationsController#create` in `spec/controllers/locations_controller_spec.rb`:

```
describe LocationsController do
  describe "#create" do
    subject { post :create, { :location =>
      { :latitude => 25.0,
        :longitude => -40.0 }
      }
    }
    its(:status) { should == 302 } # redirect
  end
end
```

2. Running this fails because there's no `#create` method in `LocationsController`. Rectify this by adding it to `app/controllers/locations_controller.rb`:

```
class LocationsController < ApplicationController
  def create
  end
end
```

3. Executing `rspec` now reveals that we're failing because the view template is missing. By default, Rails will look for a template named `create`, but in actuality we're going to redirect to the `show` template after creating a `location`. This means that we have to complete two steps to get this code to pass:

```
$ touch app/views/locations/show.html.erb
```

4. Next, update the `#create` method to redirect:

```
def create
  redirect_to location_path(0)
end
```

5. The spec passes, but once again it doesn't perform the behavior ultimately expected. Save the location to the database by modifying the expectations of `spec/location_controller_spec.rb`:

```
it "saves the location" do
  subject
  Location.all.count.should == 1
end
```

Did you notice the invocation of `subject` on a line all by itself in this spec? Without it, the `subject` block would not have been invoked and the spec would then fail, even for a valid implementation.

6. Next, add an implementation in `app/controllers/locations_controller.rb`:

```
def create
  @location = Location.new(params[:location])
  @location.save
  redirect_to location_path(0)
end
```

7. There is still an artifact of the simplest solution: we're passing `0` as an argument to `location_path`. Without a spec, we have no reason to change. Let's rectify this:

```
it "should redirect to show the created location" do
  subject.should redirect_to(location_path(Location.first.id))
end
```

8. `rspec` fails, so we add the implementation in `app/controllers/locations_controller.rb` by updating the `redirect_to` line:

```
redirect_to location_path(@location.id)
```

9. The specs now pass. This isn't yet complete; validation and error handling are missing.

10. Add a `describe "#new"` block:

```
describe "#new" do
  context "when invalid longitude" do
    subject { post :create, { :location =>
      { :latitude => 25.0 } } }
    its(:status) { should == 200 } # OK
    it "should render the new view" do
      subject
      response.should render_template("new")
    end
  end
end
```

11. The empty new view template will need to exist:

```
$ touch app/views/locations/new.html.erb
```

12. No implementation of `LocationsController#new` need to be created at this time because of the way that Rails renders actions.

13. Complete the implementation that the spec requires in `LocationsController`:

```
def create
  @location = Location.new(params[:location])
  if @location.save
    redirect_to location_path(@location.id)
  else
    render :action => "new"
  end
end
```

Only a missing `longitude` was tested. Because there is good test coverage exercising the model, this should be enough to ensure that when there's at least one error the controller performs the expected action. By using stubs (discussed in a later section) you could separate the dependency of location invalidation from the actual `Location` model to make these tests less brittle.

14. The next logical spec is the `show` action. Just write the controller spec for now:

```
describe "#show" do
  context "when the location exists" do
    let(:location) { Location.create(
      :latitude => 25.0, :longitude => -40.0)
    }
    subject { get :show, :id => location.id }
    it "assigns @location" do
      subject
      assigns(:location).should eq(location)
    end
  end
end
```

15. Running `rspec` fails. The passing implementation skips past the initial baby step (`@location = Location.first`) and goes right into using the `params[:id]` value passed:

```
def show
  @location = Location.find(params[:id])
end
```

16. Now, expand the test coverage to ensure the template expected is rendered by adding the following code to the `when the location exists` context block:

```
it "renders the show template" do
  subject
  response.should render_template("show")
end
```

Now when we run `rspec`, it passes! This was unexpected and should raise a red flag because we always expect the code we write to fail at first. But, in this case, it's about Rails convention over configuration. A controller method, unless explicitly given a different render instruction in its method, will attempt to render a template named after itself.

17. This behavior can be verified by making the the `show` method attempt to render another template (optional). Now, RSpec will fail; and on removal of the incorrect `render 'xyz'` line, the specs will pass once again:

```
def show
  @location = Location.find(params[:id])
  render 'xyz'
end
```

18. Next we need to handle the `show` method, when a requested ID has no associated location inside the `describe "#show"` block:

```
context "when the location does not exist" do
  subject { get :show, :id => 404 }
  its(:status) { should == 404 }
end
```

19. Running `rspec` shows us that an `ActiveRecord::RecordNotFound` error is raised, which can be caught and an appropriate HTTP status code can be returned.

20. The implementation of `LocationsController#show` is now refactored to the following implementation:

```
def show
  begin
    @location = Location.find(params[:id])
  rescue ActiveRecord::RecordNotFound
    render :status => 404
  end
end
```

21. Next is the `#index` action beginning as always with a spec. Here, `rspec` fails because of the missing `index` method in `LocationsController`:

```
describe "#index" do
  context "when there are some locations" do
    let(:location) do
      [
        Location.create(:latitude => 25.0,
          :longitude => -40.0),
        Location.create(:latitude => -10.0,
```

```
            :longitude => 42.0)
        ]
      end
      #TODO check with let!
      before { locations }
      subject { get :index }
      it "assigns @locations" do
        subject # let!
        assigns(:locations).should eq(locations)
      end
    end
  end
```

22. Adding an empty `index` method (`def index; end`) now throws the expected `ActionView::MissingTemplate` error:

    `$ touch app/views/locations/index.html.erb`

23. After the preceding step, there's now a failing spec that can be solved by writing code implementing the `LocationsController#index` method:

```
def index
  @locations = Location.all
end
```

In the previous spec, there is an explicit before `{ location }` block. If this was absent, the `let` statement for locations would not be executed as a part of the `subject` block and our spec would not pass.

The `render_template("index")` expectation should be made for the preceding context, but the implementation has been omitted for brevity.

24. Implementing a `when there are no locations` context in the `describe "#index"` block will improve the code coverage despite the fact that it drives no new implementation:

```
context "when there are no locations" do
  subject { get :index }
  it "assigns @locations" do
    subject
    assigns(:locations).should eq([])
  end
end
```

25. The `destroy` action is described as follows:

```
describe "#destroy" do
  context "when the location exists" do
    let (:location) { Location.create(
      :latitude => 25.0, :longitude => -40)
    }
    subject { post :destroy, :id => location.id }
    it "deletes the location" do
      subject
      Location.all.count.should == 0
    end
  end
end
```

26. Now, enter a command to create the necessary view to support the `#destroy` action:

```
$ touch app/views/locations/destroy.html.erb
```

27. The following is the `LocationsController#destroy` implementation:

```
def destroy
  Location.destroy(params[:id])
end
```

This example should be speced out for the same 404 HTTP status when the location to be deleted doesn't exist, similar to `LocationsController#show`, although that exercise has been omitted here for the sake of brevity.

Stubbing (Intermediate)

Stubbing and mocking are powerful techniques that can be used to simulate a runtime environment during test time. Stubs reply with an expected result, whereas mocks verify specific behavior such as whether methods were invoked at all, with what arguments, and in what order.

How to do it...

1. In a Rails project, views are a good place to introduce stubs, although they are applicable anywhere. We'll work with the `show` action's associated view by first making a spec file for it:

```
$ mkdir -p spec/views/locations
$ touch spec/views/locations/show.html.erb_spec.rb
```

2. In the newly created `show.html.erb_spec.rb` file, we can drive the expectations of a simple view:

```
require "spec_helper"

describe "locations/show" do
  before do
    assign(:location,
      stub_model(Location, :latitude => 42.0,
        :longitude => -12.4)
      )
  end
  it "displays the latitude" do
    render
    expect(rendered).to match /Latitude:\S*42\.0/
  end
end
```

3. In `app/views/locations/show.html.erb`, write the implementation necessary to pass the spec:

```
<label>Latitude:</label><%= @location.latitude %>
```

4. Adding a similar test case for `longitude` follows the same implementation procedure. (Code omitted for brevity.)

How it works...

In the `before` block in the previous code example, RSpec's `assign` method is used to inject the `@location` variable—normally created by a controller as part of a request and handed off to the view—in a stub. `stub_model` creates an instance of the `Location` class and assigns its attributes to the specified values. It's as if a web browser made a request to the server, it was routed to the controller, the controller retrieved the location from the database, and that location was then handed off to the view to render back to the client's browser—but without all those costly dependencies.

Using stubs keep these tests running fast, which greatly helps during the Test-Driven Development process.

Mocking (Intermediate)

Mocking is often confused with stubbing. While mocks also permit the elimination of costly dependencies, it's the behavior of the mock—what methods are invoked and with what arguments—that is what needs to be asserted.

The `Location` class has a `#near?` method we'd like to make use of in our `LocationsController`. However, we've already verified that the `Location#near?` method works in the model specs and there's no reason to repeat that same functional test in our controller. In fact, repeating a test makes your tests brittle because if the behavior changes, you'll need to update multiple tests. While test driving the controller, the only concern is whether it behaves correctly when its dependencies furnish it with different return values.

How to do it...

1. Test drive a route. The test is omitted for brevity but the relevant `config/routes.rb` file is:

```
post "locations/near/:id" => "locations#near"
```

2. Next is the spec in `spec/controllers/locations_controller_spec.rb`:

```
describe "#near" do
  let(:location) { double("Location") }
  before do
    Location.should_receive(:find).with(42)
      .and_return(location)
  end
  context "when the supplied coordinates are near" do
    it "renders the near view" do
      location.should_receive(:near?)
        .with(25.0, 62.1, 1.0).and_return(true)
      post :near, :id => "42", :latitude => 25.0,
        :longitude => 62.1
      response.should render_template("near")
    end
  end
end
```

3. The naive implementation in `app/controllers/locations_controller.rb` is:

```
def near
  location = Location.find(params[:id].to_i)
  location.near?(params[:latitude].to_f,
    params[:longitude].to_f, 1.0)
end
```

4. Create the empty `near` view template:

```
$ touch app/views/locations/near.html.erb
```

The specs pass, which means that the behavior requested that `Location#find` is invoked with the numeric value `42` and the location mock returned has its `#near?` method invoked with the expected arguments. The default action is to render the `near` view, which passes our assertion and immediately raises a red flag, as we want to see the test fail at first. We could explicitly render the incorrect view but instead we'll tackle this from the other direction—rendering a `far` view when the `Location#near?` method returns `false`.

5. In `spec/controllers/locations_controller.rb`, create a new context within the `describe "#near"` block:

```
context "when the supplied coordinates are far" do
  it "renders the far view" do
    location.should_receive(:near?)
      .with(25.0, 62.1, 1.0).and_return(false)

    post :near, :id => "42", :latitude => 25.0,
      :longitude => 62.1
    response.should render_template("far")
  end
end
```

6. And create the empty `far` view template:

```
$ touch app/views/locations/far.html.erb
```

7. On running the spec, it fails, forcing implementation of the expected behavior by replacing `LocationsController#near` with this definition:

```
def near
  location = Location.find(params[:id].to_i)
  unless location.near?(params[:latitude].to_f,
    params[:longitude].to_f, 1.0)
    render :far
  end
end
```

8. Now all specs pass. Importantly, note that we invoke the `Location#near?` method with the same arguments but return a varying reply depending on the context, meaning whether we want the mocked location to be near or far from the supplied coordinates. We don't have to go to some external resource and calculate a pair of latitude/longitude coordinates and rely on the implementation of `Location#near?`.

Working with JSON (Intermediate)

JSON is a lightweight text-based communication medium that commonly appears in place of XML in newer web applications and web services.

For this exercise, let's pretend that our web application has a sibling mobile application that will transmit geographic coordinates and the application needs an API, which it can communicate with.

How to do it...

1. Begin with updates to `spec/controllers/locations_controller_spec.rb`:

```
context "when JSON format" do
  describe "#create" do
    subject { post :create, { :format => :json,
                :location => {
                   :latitude => 25.0,
                   :longitude => -40.0 }
              }
    }
    its(:status) { should == 200 } # OK
  end
end
```

This looks a lot like our normal `create` method except as an API endpoint, we expect a `200` and not a `302`. The `:format => :json` is included in the hash of submitted data. This allows us to use the JSON format in `LocationsController`.

2. Specify that `LocationsController` can respond to JSON content by adding the following lines to `LocationsController` (`app/controllers/locations_controller`):

```
class LocationsController < ApplicationController
  respond_to :html, :json
  # remainder omitted
Next, update LocationsController#create with the following
definition:
def create
  @location = Location.new(params[:location])
  if @location.save
    respond_to do |format|
      format.json { head :ok }
      format.html { redirect_to location_path(@location.id) }
    end
  else
    respond_to do |format|
      format.json { head :bad_request }
      format.html { render :action => "new" }
    end
  end
end
```

In APIs, it's typically enough to reply with an HTTP status code so we use the Rails `#head` method (in lieu of render `:nothing, :status => :ok`). The validation failure condition, where `:bad_request` is used, leaves a lot to be desired. In an actual API, for calling a helper method that renders an error, JSON view would be acceptable; for the purposes of illustration `:bad_request` will suffice.

3. We can take this one step further and validate the API endpoint by using a URL. First, start the Rails server:

```
$ rails server
```

4. Execute the following command:

```
$ curl -v -H "Content-type:application/json"
  -X POST -d '{"location":{"latitude":-25.0,"longitude":40.0}}'
  http://localhost:3000/locations
```

You're expecting to see an **HTTP 1.1 200 OK** reply output by cURL (the `-v` switch is necessary to see the output headers).

It's the `Content-type` header that's important as that sets the `params[:format]` value, which the controller uses to determine the format to be used. (Instead of specifying the header, make the request to `locations.json`—running rake routes reveals that there's an optional (`.format`) parameter for each route.)

5. Next, wire up the `show` action for JSON, as this will be an actual JSON reply. As usual, begin with a spec in `spec/controllers/locations_spec.rb` (within the `when JSON format` context block):

```
describe "#show" do
  let(:location) { Location.create(:latitude => 25.0,
                                   :longitude => 40.0) }
  subject { get :show, { :format => :json,
    :id => location.id } }
  its(:status) { should == 200 }
end
```

   This fails with the expected missing template error message.

6. There are several ways to proceed. We could author an `app/views/locations/show.json.erb` view template, use a third-party library such as the excellent `rabl` gem, or just a simple Rails built-in: `#respond_with`. Choose the solution that matches the requirements, and in this case there's no reason to overcomplicate things so `#respond_with` is fine:

```
def show
  begin
    @location = Location.find(params[:id])
    respond_with(@location)
  rescue ActiveRecord::RecordNotFound
    render :status => 404
  end
end
```

7. How do we know whether the reply is JSON? If we had used a view template such as `show.json.erb` or `rabl`, we'd have to use `render_views` in our controller spec. Because we're using `#respond_with`, the body of the response can be inspected:

```
it "replies with JSON" do
  json = JSON.parse(subject.body)
  json.should have_key("id")
end
```

8. `render_views` and parsing the response body within a controller is a testing antipattern—instead of verifying controller behavior, we're trying to validate view logic. We can use mocking to ensure that a JSON request results in `Location#to_json` method being invoked—a precursor to a JSON reply:

```
it "replies with JSON" do
  Location.any_instance.should_receive(:to_json).and_return({})
  subject
end
```

9. The next problem is that requests for JSON formatted nonexistent locations result in a HTML formatted reply for a missing template:

```
$ curl -v -H "Content-type: application/json" \
  -X GET http://localhost:3000/locations/42
```

10. This presents an opportunity to refactor the `begin/rescue/end` block for `ActiveRecord::RecordNotFound`. Drive out a 404 with the expectation that the content type is `application/json`:

```
context "when a location doesn't exist" do
  subject { get :show, { :format => :json, :id => "42" } }
  its(:status) { should == 404 }
  its(:content_type) { should == "application/json" }
end
```

11. This problem cannot be solved by merely adding the missing template. Instead a slightly more robust solution is required. Declare `LocationsController#not_found` and wire it up as the handler for `ActiveRecord::RecordNotFound` errors. First, at the start of `app/controllers/locations_controller.rb`, add the `rescue_from` line:

```
class LocationsController < ApplicationController
  rescue_from ActiveRecord::RecordNotFound,
    :with => :not_found
  # remainder omitted
```

```
At the end of LocationsController, add the private method #not_
found:
private
  def not_found(e)
    respond_to do |format|
      format.html {
          render :file => "public/404",
          :formats => :html,
          :status => :not_found }
      format.json { render :json => { :message => e.message },
                           :status => :not_found }
    end
  end
```

12. Finally, refactor the `LocationsController#show` method by removing several lines:

```
def show
  @location = Location.find(params[:id])
  respond_with(@location)
end
```

If you have optionally implemented the 404 action for `#destroy`, you can refactor the `begin/rescue/end` block in the same way as `show` was just refactored.

All specs now pass, including the previous specs for HTML-formatted 404 not found handlers. `LocationsController#destroy` could likewise be refactored next (although omitted for brevity).

Speccing file uploads (Advanced)

File uploads can be a little daunting to test at first. Fortunately, rspec-rails provides helper methods to make this sort of testing easier.

Getting ready

1. To enable file storage in the database, create and run a migration:

```
$ rails g migration AddImageToLocations image:binary
$ rake db:migrate
$ rake db:test:prepare
```

2. Update `app/models/location.rb` to include the `image` attribute:

```
class Location < ActiveRecord::Base
  attr_accessible :latitude, :longitude, :image
  # remainder omitted
```

3. And finally, generate a test file fixture:

```
$ mkdir -p spec/fixtures/files
$ dd if=/dev/urandom of=spec/fixtures/files/test.png \
  bs=1 count=1024
```

This fixture is used to represent our image for uploading. It's just a series of random bytes. If you're following along on Windows, you can grab any file you like, but a smaller file size is ideal since the unit test will copy this file as part of the upload process every time the spec is executed.

How to do it...

A file can be uploaded into a controller using rspec-rails' `#fixture_file_upload`. The goal is to have `LocationsController#create` to save the uploaded file in the database.

1. In `spec/controllers/locations_controller_spec.rb`, update the original `describe "#create"` block to:

```
subject { post :create, { :location =>
  { :latitude => 25.0,
    :longitude => -40.0 },
    :image => fixture_file_upload("/files/test.png",
      "image/png", :binary => true)
  } }
it "saves the file in the image column" do
  subject
  Location.first.image.size.should == File.size(
    "spec/fixtures/files/test.png")
end
# remainder omitted
```

It's important to note the file upload, although associated in the hash with a `:image` parameter key—the same as the attribute name on the `Location` model. It exists outside the parameter values that Rails will automatically bind to the `Location` model instance it creates in the controller. This is because uploaded files are special cases and must be handled differently from regular form encoded parameters.

2. The specs fail, so the next step is to make them pass by updating the definition of `LocationsController#create` in `app/controllers/locations_controller.rb`:

```
def create
  @location = Location.new(params[:location])
  image_file = params[:image]
  @location.image = image_file.read unless image_file.nil?
  # remainder omitted
```

There's more...

Storing large amounts of binary data in a database is sometimes not the correct solution, although duplicating it there may often be worth the effort (in terms of backups and replicating data to a new environment). Speccing file uploads—wherever the data ends up being stored—is still important.

It is sometimes necessary to combine a JSON API endpoint with file uploads. In these cases, there are a few different options: multipart/form-data (instead of placing JSON in the HTTP body—this is what is specced in the preceding example), multipart/mixed (where the HTTP body is the encoded payload), and finally, base 64 within JSON (or some other form of binary to UTF-8 encoding).

Integration testing with Capybara (Advanced)

Capybara is a testing tool that helps test web applications by simulating the interaction a user would have with the application through their browser. It even has support for JavaScript behavior by driving an actual web browser.

In this recipe, Capybara is used to simulate a user interacting with the new view template of `LocationController` and then add client-side JavaScript validation.

By default, Capybara uses a very limited but fast headless web browser for validating **Rack** applications though it supports additional web browsers through different drivers. These examples use the **Selenium** driver for Capybara because it is the default. It requires that you have Mozilla Firefox installed.

Getting ready

1. Add the gem dependencies to the `Gemfile`:

```
gem 'capybara', :group => :test
gem 'dynamic_form'
gem 'client_side_validations'
```

   Here, `dynamic_form` is used for showing error messages when validation fails on the server side and `client_side_validations` is a simple and fast way of extending server validations to client-side JavaScript.

Anytime you update the Gemfile dependencies, make sure you run bundle install from the command line:

```
$ bundle install
```

2. In `spec/spec_helper.rb` add the following:

```
require 'capybara/rspec'
```

3. Create a `spec/features` directory:

```
$ mkdir -p spec/features/controllers
```

4. To complete the `client_side_validations` installation, there are a few more steps that must be taken. First, execute the following command to create the initializer:

```
$ rails g client_side_validations:install
```

5. Next, open up the initializer file generated at `config/initializers/client_side_validations.rb` and uncomment the entire block following the line which reads:

```
# Uncomment the following block if you want each...
```

6. Now, add the following line to `app/assets/javascripts/application.js` at the bottom of the file:

```
//= require rails.validations
```

7. `LocationsController` (`app/controllers/locations_controller.rb`) needs a `#new` method:

```
def new
  @location = Location.new
end
```

How to do it...

1. In `spec/features/controllers/locations_controller_spec.rb`, add the following code (note the new file located under the `spec/features/controllers directory`):

```
require 'spec_helper'
describe LocationsController do
  describe "#new" do
    before { visit(new_location_path) }
    context "when using valid values" do
      it "redirects to show the location" do
        fill_in 'Latitude', :with => '-42.103826'
        fill_in 'Longitude', :with => '77.899063'
        click_button('Create')
        current_path.should =~ /locations\/\d+/
      end
    end
  end
end
```

2. Running `rspec` will fail because there are no fields or other user interface elements in `app/views/locations/new.html.erb`. Add them:

```
<%= form_for @location do |f| %>
    <%= f.error_messages %>
    <%= f.label :latitude, 'Latitude' %>
    <%= f.text_field :latitude %>
    <%= f.label :longitude, 'Longitude' %>
    <%= f.text_field :longitude %>
    <%= f.submit 'Create' %>
<% end %>
```

3. Unlike the controller specs, these integration tests are full-stack, meaning HTTP redirects are followed and rendered views are important.

As Capybara executes on a separate thread and the majority of the testing techniques in the Rails environment rely on database transactions, validating the state of the database when a spec finishes is unreliable. Instead, focus on what the end user should see as a result of these scripted actions.

The `fill_in` helper method used is case sensitive; it can accept a variety of arguments for selectors but the two most commonly used are selecting the label by its text, as shown in the previous code example, and the field by name. In the case of the previous view template, `location_latitude` would be a valid locator for the `latitude` field.

4. Next, validate the failure condition of empty values:

```
context "when using empty values" do
  it "shows four error messages" do
    click_button('Create')
    page.html.should =~
      /4 errors prohibited this location from being saved/i
  end
end
```

5. While working through these specs, it may not be known how the output will exactly look. For cases such as these, you could use `puts page.html`, or even compare it with a best guess regular expression and check the comparison output in the console.

6. In the previous examples, the browser Capybara is simulating is an extremely basic headless web browser with no client-side JavaScript support. The following example uses the `:js => true` setting to drive a browser that supports JavaScript—this is where you'll see Capybara fire up Mozilla Firefox and execute a scripted test:

```
context "when performing client-side validation" do
  context "when using non-numeric and empty values" do
    it "shows two error messages", :js => true do
      fill_in 'Latitude', :with => 'invalid'
      click_button('Create')
      find(:xpath,
'//*/label[@class="message" and @for="location_latitude"]')
        .should have_content("is not a number")
      find(:xpath,
'//*/label[@class="message" and @for="location_longitude"]')
        .should have_content("can't be blank")
    end
  end
end
```

Due to the length of time a browser integration test takes to execute, there are two validations occurring in the same spec. These could be broken up into separate tests, as done with all of the other RSpec examples so far, though that comes at a cost of extra time to execute the tests. Ultimately, you need to strike a balance between code maintainability and your integration test strategy.

7. To pass the spec enable the `client_side_validations` gem for this form in `app/views/locations/new.html.erb`:

```
<%= form_for @location, :validate => true do |f| %>
```

There's more...

When a `:js => true` spec is executed, an instance of the Firefox web browser will be launched on a separate thread and driven by Capybara's Selenium driver. For a split second it's possible to see the fields as they are filled out by the driver. To keep the page open, you'll need to add a dependency for the test group to your `Gemfile`:

```
gem "launchy", :group => :test
```

Then in any `:js => true` spec the following line:

```
save_and_open_page
```

This will cause your default browser to open a temporary file that contains the HTML snapshot of the page at that point. Depending on the version of Capybara installed, form fields may not have their values displayed but DOM elements should still be intact. Using this method with a web browser such as Chrome, you can right-click on any element by navigating to **Developer tools** | **Elements** and copy XPath. Then you're not guessing about where values appear when trying to validate your specs.

Capybara

As an integration test driver, Capybara is more commonly used after you've used TDD to drive out the design to improve test coverage and ensure user interaction is modeled correctly. That doesn't mean we can't use it as an agent of TDD, but it's important to remember that the purpose of Capybara is to verify what the end user sees.

I personally prefer lighter alternatives such as view speccing and mocks for attempting a full-stack integration test on the user interface. For small projects, though it may be a valid choice, if client-side behavior needs to be tested, adding an additional JavaScript testing framework such as Jasmine isn't justified.

Specification tagging

You may notice that execution of your specs has become very slow due to the inclusion of the spec that fires up Firefox for client-side JavaScript validation. RSpec supports tagging specs and omitting tags from your test run. To tag a spec as an integration test—the `:integration` symbol here is arbitrary —you can write the following:

```
it "shows two error messages", :js => true,
  :integration => true do
```

And then in the command line execute:

```
$ rspec -t~integration
```

The tilde (~) symbol means exclude this tag. You don't even need to specify integration since that particular test is already tagged `:js`, but you may wish to adopt the `:integration` moniker as a classification standard for all integration tests.

You can avoid having to enter `-t~integration` in the command line constantly by adding it to your `.rspec` file, on its own line:

```
--color
-t~integration
```

Then, to run integration tests you just have to include the tag:

```
$ rspec -tintegration
```

Thank you for buying
Instant RSpec Test-Driven Development How-to

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

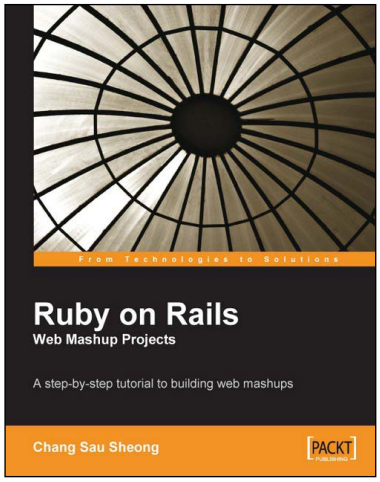
Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT]
PUBLISHING



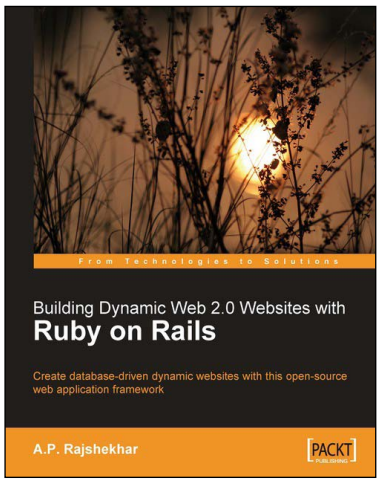
Ruby on Rails Web Mashup Projects

ISBN: 978-1-847193-93-3 Paperback: 272 pages

A step-by-step tutorial to building web mashups

1. Learn about web mashup applications and mashup plug-ins
2. Create practical real-life web mashup projects step by step
3. Access and mash up many different APIs with Ruby and Ruby on Rails

---



Building Dynamic Web 2.0 Websites with Ruby on Rails

ISBN: 978-1-847193-41-4 Paperback: 232 pages

Create database-driven dynamic websites with this open-source web application framework

1. Create a complete Web 2.0 application with Ruby on Rails
2. Learn rapid web development
3. Enhance your user interface with AJAX