Rails-Bootstrap

Last updated 05 Jun 2014

GitHub Repository (https://github.com/RailsApps/rails-bootstrap) · Issues (https://github.com/RailsApps/rails-bootstrap/issues)

RSpec Quickstart Guide

Introduction

This guide shows how to use RSpec (http://rspec.info/) for testing Rails applications. Versions:

- RSpec 3.0
- Rails 4.1

See the article Installing Rails (http://railsapps.github.io/installing-rails.html) for instructions about setting up Rails and your development environment.

What's Here

You'll find basics here:

- how to install RSpec
- concepts and terminology
- example tests

Is It for You?

RSpec is the most popular test framework for Rails. This guide shows how to set up RSpec and explains how to write simple tests. It provides an introduction to basic concepts and recommended practices. If you've created simple Rails applications, such as the one in the book Learn Ruby on Rails (https://tutorials.railsapps.org/learn-ruby-on-rails), you'll have enough background to follow this guide. RSpec is complex and there are few comprehensive introductions available elsewhere, so experienced Rails developers as well as beginners will find this guide helpful.

Create an Application

We'll use the Rails Composer (http://railsapps.github.io/rails-composer/) tool to build a starter app. For this tutorial, we'll start with the rails-bootstrap (https://github.com/RailsApps/rails-bootstrap) example application. Though you can build the starter app with RSpec automatically installed, we'll build it without RSpec, and then add RSpec manually so you'll see everything that is required to use RSpec.

To build the starter app, Rails 4.1 must be installed in your development environment. Run the command:

```
$ rails new rails-bootstrap -m https://raw.github.com/RailsApps/rails-composer/master/composer.
rb -T
```

Be sure to add the `-T` flag to skip Test::Unit files that are not needed for RSpec.

You'll see a prompt:

```
question  Build a starter application?
      1)  Build a RailsApps example application
      2)  Contributed applications
      3)  Custom application
```

Enter “1” to select Build a RailsApps example application. You'll see a prompt:

```
question  Starter apps for Rails 4.1. More to come.
      1)  learn-rails
      2)  rails-bootstrap
      3)  rails-foundation
      4)  rails-omniauth
      5)  rails-devise
      6)  rails-devise-pundit
```

Choose rails-bootstrap. The README (https://github.com/RailsApps/rails-composer) for Rails Composer has more information about the options.

Choose these options to create a starter application for this tutorial:

- Web server for development? WEBrick (default)
- Web server for production? Same as development
- Template engine? ERB

- Test framework? None
- Set a robots.txt file to ban spiders? no
- Create a GitHub repository? no
- Use or create a project-specific rvm gemset? yes

Be sure to choose “None” for the test framework, so you can follow the instructions to install RSpec in this tutorial. If you choose the option “RSpec with Capybara,” all the installation will be done for you and it will be more difficult to follow the tutorial.

Installation

In summary, here are the steps for installing RSpec:

- add gems
- set up databases
- run `rails generate rspec:install`
- adjust configuration files

Gems

Testing with RSpec requires a suite of gems.

Here is a description of the gems we use for testing with RSpec:

- rspec-rails (https://github.com/rspec/rspec-rails) – installs RSpec gems with support for Rails
- factory_girl_rails (https://github.com/thoughtbot/factory_girl_rails) – creates test data
- capybara (https://github.com/jnicklas/capybara) – tests web pages
- database_cleaner (https://github.com/bmabey/database_cleaner) – a clean slate for databases
- launchy (https://github.com/copiousfreetime/launchy) – view errors in your web browser
- selenium-webdriver (http://docs.seleniumhq.org/projects/webdriver/) – for tests that require JavaScript

Not all developers use these gems, and some developers add more; this is the most common set used with RSpec.

Gem Alternatives and Extras

There are alternatives:

- fabrication (http://www.fabricationgem.org/) and others (https://www.ruby-toolbox.com/categories/rails_fixture_replacement) instead of factory_girl (https://github.com/thoughtbot/factory_girl)
- poltergeist (https://github.com/jonleighton/poltergeist) or capybara-webkit (https://github.com/thoughtbot/capybara-webkit) instead of selenium-webdriver (http://docs.seleniumhq.org/projects/webdriver/)

Several other gems are often used for testing with Rails:

- shoulda-matchers (https://github.com/thoughtbot/shoulda-matchers) – test ActiveRecord associations and more

- faker (https://github.com/stympy/faker) – generates dummy data
- email-spec (https://github.com/bmabey/email-spec) – test email messages
- capybara-screenshot (https://github.com/mattheworiordan/capybara-screenshot) – capture screenshots
- Konacha (https://github.com/jfirebaugh/konacha) , Jasmine (http://jasmine.github.io/), or Evergreen (https://github.com/abepetrillo/evergreen) – JavaScript testing

We'll consider each of these gems later in the guide (except JavaScript testing, that's a book in itself).

Update the Gemfile

In your Gemfile, add:

```
group :development, :test do
  gem 'rspec-rails'
  gem 'factory_girl_rails'
end
group :test do
  gem 'capybara'
  gem 'database_cleaner'
  gem 'launchy'
  gem 'selenium-webdriver'
end
```

We install the gems in two groups.

The rspec-rails and factory_girl_rails gems are used in both development and testing. In development, if you use Rails generators to create models or controllers, the rspec-rails and factory_girl_rails gems will create simple "stub" files for your tests. These gems must also be available in the test environment.

The other gems are only used when running tests, so they can be segregated in a test group so they are not loaded in development or production.

Install the Gems

Install the gems on your computer:

```
$ bundle install
```

The bundle install command will install the gems from the rubygems.org server.

RSpec Generator

Run a generator to set up RSpec:

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

The generator will install three configuration files and create a folder:

- .rspec

- spec/
- spec/spec_helper.rb
- spec/rails_helper.rb

The .rspec file contains options that will be applied when running RSpec from the command line.

The files spec/spec_helper.rb and spec/rails_helper.rb are required by every test. They set the test environment, configure RSpec for application specifics, and load support files.

Prior to RSpec 3.0, only the spec/spec_helper.rb file was required. The spec/rails_helper.rb was added to segregate Rails-specific preferences.

RSpec Configuration

RSpec reads command line configuration options from files in two different locations:

- .rspec – in the project's root directory
- ~/.rspec – in the developer's home directory

The RSpec maintainers recommend including the .rspec file in the project's root directory for project-specific settings.

If you want to set preferences for all your projects, create an .rspec file in your home directory. Options declared in the project file override those in the global file.

Edit the file .rspec to remove the line:

```
--warnings
```

You can add the `--warnings` line when you want to see every warning message. Usually it introduces too much "noise."

Add the line:

```
--format documentation
```

This provides attractive and informative output when you run tests.

Finally, add a statement to require the rails_helper.rb file:

```
--require rails_helper
```

Prior to RSpec 3.0, you were expected to include the line `require 'spec_helper'` at the top of every test file. With RSpec 3.0 and later versions, an additional `require 'rails_helper'` statement is necessary. Fortunately, starting with RSpec 3.0, these requirements can be specified in the .rspec file, eliminating clutter and duplication.

Your .rspec file should look like this:

```
--color
--format documentation
--require spec_helper
--require rails_helper
```

RSpec Support Folder

The default RSpec configuration in spec/rails_helper.rb will load any files in a spec/support folder (and subfolders). The folder is commonly used for code that is shared among tests as well as custom matchers and various configuration files.

Create the folder for future use:

```
$ mkdir spec/support
```

Capybara

The Capybara default configuration is adequate for simple projects, so you only need to configure Capybara for special situations.

You can configure Capybara by adding a file spec/support/capybara.rb. When we learn about feature tests, we'll look at options to configure Capybara.

Database Cleaner

You don't need to set up database_cleaner (https://github.com/bmabey/database_cleaner) for simple tests. If you write feature tests that run JavaScript, you'll need to set up database_cleaner. You will find details at the end of the "Feature Tests" section. You can add configuration options to a file spec/support/database_cleaner.rb or make changes to the spec/rails_helper.rb file.

By default, RSpec automatically cleans up the test database after each test using a fast *transactional* cleanup strategy. The transactional strategy can't be used for testing features that require application JavaScript. In this case, you will need to modify the RSpec configuration to use a *truncation* strategy. *Transaction* and *truncation* are terms from the database world. A transactional strategy reverses each database interaction. A truncation strategy deletes an entire database table. We'll look at this in more detail when we learn about feature tests.

FactoryGirl

You can use the FactoryGirl defaults without any configuration.

To change the defaults, create a file spec/support/factory_girl.rb with your configuration options, or add the configuration to the spec/rails_helper.rb file.

Adjust Rails Generators

When you use Rails generators to create models or controllers, the rspec-rails and factory_girl_rails gems will create stub files for your tests.

When you create a model with `rails generate model` (see the RailsGuide (http://guides.rubyonrails.org/command_line.html#rails-generate) for options), the generator will create two test files:

- spec/models/mymodel_spec.rb
- spec/factories/mymodel.rb

When you create a controller with `rails generate controller` (see the RailsGuide (http://guides.rubyonrails.org/command_line.html#rails-generate) for options), the generator will create additional test files:

- spec/controllers/mycontroller_spec.rb
- spec/helpers/my_helper_spec.rb
- spec/views/my_view/

If you use the `rails generate scaffold` command, you'll get stub files in all these directories:

- spec/controllers/
- spec/factories/
- spec/helpers/
- spec/models/
- spec/requests/
- spec/routing/
- spec/views/

Most of these files are clutter, since most developers don't attempt such complete test coverage.

Suppress the creation of the extra stub files by adding this block to the config/application.rb file:

```
.
.
.
module RailsBootstrap
  class Application < Rails::Application
    .
    .
    .
    config.generators do |g|
      g.test_framework :rspec,
        fixtures: true,
        view_specs: false,
        helper_specs: false,
        routing_specs: false,
        controller_specs: false,
        request_specs: false
      g.fixture_replacement :factory_girl, dir: "spec/factories"
    end

  end
end
```

This will automatically generate stub files for factories, but nothing else. If you write additional tests, you can always create the files manually.

Remove the Test Folder

By convention, all RSpec tests go in the spec/ folder. If you use the `-T` flag when you create your application with `rails new`, you won't have the default test/ folder. If you have the test/ folder, you can remove it.

Run RSpec

Databases

Your Rails application has at least three databases: development, test, and production. When you run your application in development mode, it uses the development database. When you run RSpec, your application runs in a separate test environment and uses the test database.

Ordinarily, the test database will be empty (no records) when RSpec runs. That's because RSpec (or in some situations, the database_cleaner gem) restores the test database to an empty state after each test run.

Database Creation

Before you launch your application or run RSpec for the first time, you must create all the databases. This simply prepares the databases for use.

Use the command:

```
$ rake db:create:all
```

This creates both the development and test databases. If you're not using rvm, the Ruby version manager, you should use the command `bundle exec rake db:create:all`.

If you've generated a starter app with Rails Composer, as we did earlier with the tutorial application, the databases already are created and you'll see the message `... already exists`.

Database Migration

Prior to Rails 4.1 and RSpec 3.0, every time you modified the database schema with a migration, you'd need to run `rake db:test:prepare` before running RSpec, just to copy the schema from the development database to the test database. Beginning with RSpec 3.0, migrations are automatically applied to the test database. The spec/rails_helper.rb file contains a configuration setting that eliminates the need to run `rake db:test:prepare` (see this commit (https://github.com/rspec/rspec-rails/commit/aa5c5f76a9ae847648f48e0ea7802b8e83c11d0f)).

Open the file spec/rails_helper.rb and look for the configuration setting:

```
# Checks for pending migrations before tests are run.
# If you are not using ActiveRecord, you can remove this line.
ActiveRecord::Migration.maintain_test_schema!
```

If you see it, you're using RSpec 3.0 or newer.

If you see:

```
# Checks for pending migrations before tests are run.
# If you are not using ActiveRecord, you can remove this line.
ActiveRecord::Migration.check_pending! if defined?(ActiveRecord::Migration)
```

You are using an older RSpec gem. Be sure to update the rspec-rails gem and run `rails generate rspec:install` as described in the previous chapter. If you find a blog post or tutorial that tells you to run `rake db:test:prepare`, it is likely outdated.

Run RSpec

The files that contain tests are called "spec files." Each spec can contain multiple tests, which are called "examples" or "test cases."

When you run the `rspec` command, you'll run all the test cases in all the files in the spec/ folder.

Run all tests using the `rspec` command:

```
$ rspec
No examples found.

Finished in 0.00007 seconds
0 examples, 0 failures
```

Alternatively, you can run the rake task `rake spec`.

Run a Single Spec File

Run all tests in a single spec file:

```
$ rspec path/to/spec/file.rb
```

Run a Single Test

Spec files typically contain multiple tests. You can run a single test by specifying a line number:

```
$ rspec path/to/spec:<line number>
```

Or run a single test by specifying the description of the test:

```
$ rspec path/to/spec/file.rb -e 'Sign in with correct credentials'
```

Troubleshooting

All test activity is logged to the file log/test.log. You can examine the file to see details after running RSpec.

If you want to output any messages or variables to the test log, you can add Rails logger statements to your tests, just as you would in other Rails code:

```
Rails.logger.debug 'some message'
```

Use `rake log:clear` if your log files are overwhelming.

Concepts

Testing can be overwhelming if you don’t have a strategy to make it manageable. Let’s review basic concepts and consider what tests should be written first.

Why We Test

You may wonder why testing is emphasized as a practice in Rails development. It takes time to write tests; plus testing is one more thing to learn. Is it necessary? In fact, if you are a student or a hobbyist, testing may not be worth the effort. Build your application and try it out; if it doesn’t work, fix it. On the other hand, when money or reputation is at stake, you must make testing part of your development effort.

We use automated tests because programmers like to avoid unnecessary effort (meaning they are either efficient, or lazy, depending on your point of view). It's simply a waste of effort, if not downright impractical, to test every feature by hand, so we rely on automated testing.

Here are the three primary reasons to write tests.

For Quality Assurance

As a programmer, you want to make sure your code does what you expect, even after repeated changes and improvements. You'll often refactor, fix bugs, or add features. Testing to make sure you haven't broken anything is called *regression testing*. We write tests because we need to know our application runs as intended.

For Planning and Managing

The process of writing tests helps in planning and managing a project. When you write tests, you prepare a specification (or *specs*) that describes the application's functionality. When you've got specs, you've got something to discuss with the people who work with you, whether other developers, business associates, or the product users. If you want to monitor progress on the project, each feature described by a spec can be tracked as a task to be completed. Finally, an automated test suite can be used for *acceptance testing* to determine if everything was built as specified, at every stage of progress for the project.

For Clarity

Some developers like to say testing is a tool to write better code, and regression tests are a side effect. They say clean code is easy to test. If tests are difficult to write, it probably means that the implementation needs improvement.

In writing tests, you identify how the application should behave and how your code should be organized. You also create a workflow that helps you stay on track and organized. Writing tests that fail, followed by implementing code to make tests pass, and then either refactoring or repeating the cycle, is called *test-driven development*. It is often recommended as a way to organize your work and stay focused. Additionally, proponents of test-driven development say that writing tests helps reduce code complexity, because developers will focus on writing code to pass tests. Test-driven development makes it difficult to waste time on extras that "might be needed someday."

BDD and TDD

There are two broad approaches to testing, Behavior-Driven Development (BDD) and Test-Driven Development (TDD).

BDD provides a development approach that is suited to managing an entire project. BDD starts with *user stories* to describe how different *personas* will use the application. User stories can be the basis for defining *features*, including various *scenarios* which describe how a user interacts with the application. With RSpec, you can write *feature specs* which implement tests for each scenario.

Your feature specs provide an *integration test suite* for automated *integration testing*. Integration tests are useful for *end-to-end testing*. You can test multiple components of your application (models, controllers, views), as if a user was using the application. For a project manager, or the client of a consulting firm, integration tests can serve as acceptance tests to determine if developers have implemented every requested feature. For developers implementing an application, integration tests can be used to find out if all the application components work together. Finally, integration tests are used for regression testing, to make sure an application continues to work correctly after bugs are fixed, code is refactored, or features are added.

BDD is used most often in larger enterprises, supporting communication between developers and people who are defining the product requirements, as well as providing tools for testing deliverables. In a corporate setting, you can use the BDD approach to improve communication with the rest of the team (especially managers or non-programmers). In smaller enterprises and startups, you can use BDD to help refine the product specification.

Both BDD and TDD emphasize *test-first* development. Tests are written before features are implemented. The discipline of writing tests before writing application code helps a developer think through the application requirements and implementation possibilities. This approach has been called behavior-driven or test-driven *design*.

Unlike BDD, TDD is used extensively by solo developers as well as teams. It as a technique both for regression testing and application design. TDD is often described as “red, green, refactor.” That means tests come first, then code to make the tests pass, then reorganizing the code to improve it. Code shouldn’t be refactored without tests—otherwise how do you know you haven’t broken something? Any changes you make should result in the tests still passing. With TDD, you’ll think through the application design before you start to code and you’ll have a complete test suite when you are done.

RSpec and Capybara

RSpec is a test framework that lets you write test specifications in Ruby. It provides a set of methods to set up and execute tests, and compare the results to the developer’s expectations. Originally developed as an alternative to Ruby’s built-in Test::Unit, it is the most popular testing framework for Rails. It is popular among developers using TDD and can be used for BDD (though Cucumber (http://en.wikipedia.org/wiki/Cucumber_(software)) is also popular for BDD).

Capybara is important for BDD and all feature tests. Capybara, named after the world’s largest rodent, is a replacement for Webrat (https://github.com/brynary/webrat), the original gem used for Ruby acceptance testing (“RAT”). Capybara is a framework for tests that simulate a user interacting with a web application. You can add Capybara methods to RSpec tests to interact with a web page, including clicks, form interaction, or finding elements on the page.

Integration Tests

Integration tests typically test product features. They are your most important tests. They automate what a real user will do with your web application. They are the easiest tests to write and offer the biggest benefit. When you write integration tests, you’ll use Capybara to interact with web pages.

In this tutorial, we'll show you how to write tests that simulate a user's interaction, which we call *feature tests*, following RSpec terminology. Depending on the context, our feature tests can be called *integration tests* or *acceptance tests*. Don't be confused by the different names for the tests; the tests are the same and the terminology changes with their purpose or the context in which they are used.

Unit Tests

If you test a component in isolation, for example, when you test each method used in a model, you're doing *unit testing*. For unit tests, you'll use RSpec for your test framework, without need for Capybara.

In TDD, each component you develop will be accompanied with unit tests. For example, before you implement a model, you'll stub out tests for each method and write tests for validation of any attributes. As you implement each method, you'll add one or more test cases. When you're done, you'll have a unit test for your model.

With unit tests for each model, controller, and service object, plus integration tests to simulate use of the application, you'll have a complete test suite that allows you to make changes to the application without worrying about breaking what you've already built.

What to Test First?

Some developers like to work from the inside out, writing a model before writing controllers or views. If that's the case, you'll write tests for whatever component you implement first. However, it is common practice to start with integration tests, especially if you're doing BDD. In general, there's an advantage to writing feature tests first.

If you're not sure where to begin, start by writing a feature spec for the web application home page. As an example, consider the user story for a typical home page:

```
Feature: Home Page
  As a visitor
  I want to visit a home page
  So I can learn more about the website
```

It's easy to come up with a scenario to test it:

```
Scenario: Visit the Home Page
  Given I am a visitor
  When I visit the home page
  Then I should see "Welcome"
```

After you've written your first feature spec, you can add more features and scenarios and continue down the path of writing integration tests for each scenario. We'll put this into practice when we write our first test in the next section, "Feature Tests."

Walking Skeleton

It is a good idea to have a “walking skeleton” before writing your first feature test (the term comes from an article by Alistair Cockburn (http://alistair.cockburn.us/Walking+skeleton)). The walking skeleton is the code and infrastructure needed to deploy the simplest web application to production. The focus is the infrastructure, not the features, to make sure you can actually deploy a working web application in your environment. Only after you have your walking skeleton should you write your first feature test and begin implementing the first feature.

The easiest way to create your walking skeleton is to use an open source starter app, such as the ones offered by the RailsApps project. For this tutorial, our walking skeleton is the rails-bootstrap (https://github.com/RailsApps/rails-bootstrap) example application.

Feature Tests

Let’s look at how to add a feature test for the home page of our rails-bootstrap application. This is our first integration test.

Start by creating a folder named “features” within the spec directory:

```
$ mkdir spec/features
```

You must name the folder “features” so RSpec will automatically include the correct test support functions. RSpec applies magic that depends on the name of the folder.

If your application is simple, put all your feature specs in the “features” folder. For a complex application, add another level of organization. RSpec will apply the correct magic to any subfolders.

It is common to organize features by user persona, for example:

- spec/features/admin
- spec/features/user
- spec/features/visitor

In my user stories, I like to distinguish between a “visitor” (sometimes called a “guest”), who is not logged in and can only access public web pages, and a “user,” who is logged in. If you implement access control, you also might have an “admin” user. Complex applications may have additional personas. Features can also be grouped in subfolders.

Create a folder named “visitor”:

```
$ mkdir spec/features/visitor
```

When I’m writing a feature spec, I like to start with a user story and scenario description in the format recommended for BDD. RSpec doesn’t require this, so I add the user story and scenario description as comments.

Create a file for the first feature spec, spec/features/visitor/home_page_spec.rb:

```
# Feature: Home Page
#   As a visitor
#   I want to visit a home page
#   So I can learn more about the website

# Scenario: Visit the Home Page
#   Given I am a visitor
#   When I visit the home page
#   Then I should see "Welcome"
```

In other tutorials you may see a `require 'spec_helper'` statement as the first line in the test file. As of RSpec 3.0, it is no longer required if your .rspec project settings file includes `--require spec_helper`. RSpec 3.0 also requires a `require 'rails_helper'` statement. These statements are necessary because Rails doesn't automatically load RSpec. As long as they are included in your .rspec project settings file, you won't need the @require @ statements in every RSpec test file.

Typically we test a single feature with multiple scenarios in a single spec file. Add the RSpec code for a feature and a scenario:

```
# Feature: Home Page
#   As a visitor
#   I want to visit a home page
#   So I can learn more about the website
feature 'Home Page' do

  # Scenario: Visit the Home Page
  #   Given I am a visitor
  #   When I visit the home page
  #   Then I should see "Welcome"
  scenario 'Visit the Home Page' do

  end

end
```

The directives `feature` and `scenario` are part of the Capybara DSL (domain-specific language). The directive `feature` accepts a string that describes the feature. The directive `scenario` accepts a string that describes the user's behavior which the test will simulate.

Let's add code to implement the test:

```
# Feature: Home Page
#   As a visitor
#   I want to visit a home page
#   So I can learn more about the website
feature 'Home Page' do

  # Scenario: Visit the Home Page
  #   Given I am a visitor
  #   When I visit the home page
  #   Then I should see "Welcome"
  scenario 'Visit the Home Page' do
    visit root_path
    expect(page).to have_content 'Welcome'
  end

end
```

This test emulates a user visiting the home page and viewing the text “Welcome.”

The directive `visit` is a Capybara method (http://rubydoc.info/github/jnicklas/capybara/master/Capybara/Session#visit-instance_method) that takes a URL or Rails route as an argument. You could specify either `visit '/'` or `visit root_path` to direct Capybara to retrieve the home page.

The last line of the example uses the RSpec `expect` syntax to define an expected outcome. The directive `expect` takes a target object `page` which is provided by Capybara. The expectation can have two methods, either `to` or `not_to` , which take a *matcher* as an argument. The matcher `have_content` is provided by Capybara (http://rubydoc.info/github/jnicklas/capybara/master/Capybara/RSpecMatchers#have_content-instance_method) (it is an alias for `have_text`). This is a good example how RSpec and Capybara syntax encourages human readable tests. The code is very close to the English-language sentence, “Expect the page to have the content ‘Welcome’.”

Try running the test:

```
$ rspec

Home Page
  Visit the Home Page

Finished in 4.15 seconds (files took 1.9 seconds to load)
1 example, 0 failures
```

You’ve written your first RSpec test.

Capybara’s Finders, Actions, and Matchers

You can simulate almost any user interaction with an RSpec feature test using Capybara. Consider this example:

```
# Feature: Sign up
#   As a visitor
#   I want to sign up
#   So I can get access to protected sections of the site
feature 'Sign up' do

  # Scenario: Visitor Signs Up With Valid Data
  #   Given I am a visitor
  #   When I sign up with valid data
  #   Then I should see a successful sign up message
  scenario 'User Signs Up With Valid Data' do
    visit new_user_registration_path
    fill_in 'Name', :with => 'Test User'
    fill_in 'Email', :with => 'user@example.com'
    fill_in 'Password', :with => 'changeme'
    fill_in 'Password confirmation', :with => 'changeme'
    click_button 'Sign up'
    expect(page).to have_content 'Welcome! You have signed up successfully.'
  end

end
```

Capybara simulates a visitor filling in a form and clicking a submit button.

Capybara offers finders (http://rubydoc.info/github/jnicklas/capybara/master/Capybara/Node/Finders) which include:

- all – Find all elements on the page matching the given selector and options.
- find – Find an Element based on the given arguments.
- find_button – Find a button on the page.
- find_by_id – Find a element on the page, given its id.
- find_field (alias: field_labeled) – Find a form field on the page.
- find_link – Find a link on the page.
- first – Find the first element on the page matching the given selector and options.

Capybara offers actions (http://rubydoc.info/github/jnicklas/capybara/master/Capybara/Node/Actions) which include:

- attach_file – Find a file field on the page and attach a file given its path.
- check – Find a check box and mark it as checked.
- choose- Find a radio button and mark it as checked.
- click_button- Finds a button by id, text or value and clicks it.
- click_link – Finds a link by id or text and clicks it.
- click_link_or_button (alias: click_on) – Finds a button or link by id, text or value and clicks it.
- fill_in – Locate a text field or text area and fill it in with the given text.
- select – Find a select box on the page and select a particular option from it.
- uncheck- Find a check box and mark uncheck it.
- unselect – Find a select box on the page and unselect a particular option from it.

Finally, Capybara offers matchers (http://rubydoc.info/github/jnicklas/capybara/master/Capybara/Node/Matchers) which includes an equivalent for the `have_content` matcher you see in the code above. When you are writing RSpec feature tests, you will not use the Capybara matchers directly; instead you will use custom RSpec matchers provided by Capybara.

Matchers for Feature Tests

RSpec provides more than two dozen built-in matchers (http://rubydoc.info/gems/rspec-expectations/RSpec/Matchers), mostly suitable for unit testing. Capybara provides another dozen RSpec matchers (http://rubydoc.info/github/jnicklas/capybara/master/Capybara/RSpecMatchers) that are suitable for feature tests:

- have_button
- have_checked_field
- have_css
- have_field
- have_link
- have_select
- have_selector
- have_table
- have_text (alias: have_content)
- have_title(title)
- have_unchecked_field
- have_xpath

On complex pages, to find precisely the content you want, you may have to use the `have_xpath` matcher to drill deep into the HTML. But it's easier to start with plain text matchers, such as `expect(page).to have_content 'Welcome'`. If necessary, to avoid ambiguity, use the `have_css` matcher and only use the `have_xpath` matcher as a last resort.

Here is an example of the `have_css` matcher:

```
feature 'Home Page' do

  scenario 'Visit the Home Page' do
    visit root_path
    expect(page).to have_css '.navbar-brand', 'Home'
  end

end
```

I've removed the comments to save space. This code tells Capybara to look for a navigation bar with a tag containing the Bootstrap class `.navbar-brand` and the text "Home." We could be even more specific by adding a CSS id selector to the page and using `expect(page).to have_css '#myIdentifier', 'Home'`. CSS id selectors should be applied to a single, unique element only.

Debugging Feature Tests

Our example scenario is simple but real-world scenarios can be complex, with forms, multiple pages, and complex HTML. The launchy (https://github.com/copiousfreetime/launchy) gem lets us interrupt a scenario test to display the target web page. Think of it as setting a “breakpoint” to halt the flow of executing code.

Here’s an example:

```
feature 'Home Page' do

  scenario 'Visit the Home Page' do
    visit root_path
    save_and_open_page
  end

end
```

Run the test with `rspec` from the command line and a browser will open with the home page when the test reaches the `save_and_open_page` directive. Use your browser’s “view source” option to check if the text or selector you need is on the page.

Include Assets

Launchy will display a bare HTML page without CSS and JavaScript. It may be enough to debug the tests but often you’ll want to see the page as the user sees it.

If you’d like Launchy to display your page with CSS and JavaScript, you can configure Capybara by adding a file spec/support/capybara.rb containing:

```
Capybara.asset_host = 'http://localhost:3000'
```

Run `rails server` in another terminal window so that the browser can obtain the compiled assets files.

Screenshots

If your test suite is large, you may not want to wait in front of the browser for Launchy to show your test errors. Instead you can run the tests and automatically save screenshots when a test fails. Use the capybara-screenshot (https://github.com/mattheworiordan/capybara-screenshot) gem to capture a screenshot of a page instead of opening a browser. Your screenshots will be waiting for you when you get back from fetching coffee.

Test JavaScript

By default, RSpec feature tests will not run any JavaScript in your application. Capybara’s default web driver, Rack::Test, is fast but doesn’t do JavaScript. If you need to include JavaScript interactions in your tests—for example, revealing a hidden button in a modal window—you must enable one of Capybara’s alternative web drivers.

In the “Installation” section of this guide, you already installed the selenium-webdriver (http://docs.seleniumhq.org/projects/webdriver/) gem. With Selenium, you will run JavaScript by running tests in a real browser. To use Selenium, you must either install Firefox (http://www.mozilla.org/en-US/firefox/new/) or change a setting (https://github.com/jnicklas/capybara#configuring-and-adding-drivers) in the file spec/support/capybara.rb to use Chrome. The Selenium web driver for Firefox is faster, so it is best to stick with the Firefox default.

Other than installing Firefox, no set up is required to use the Selenium web driver. To force your feature tests to use a real browser, add the `:js` tag to a scenario.

Here’s an example:

```
feature 'Home Page' do

  scenario 'Visit the Home Page', :js do
    visit root_path
    expect(page).to have_content 'Welcome'
  end

end
```

When you run the example test, you’ll see the Firefox browser automatically launch and the application home page will briefly appear before the browser automatically closes.

Other web drivers are available for Capybara, notably poltergeist (https://github.com/jonleighton/poltergeist) or capybara-webkit (https://github.com/thoughtbot/capybara-webkit) instead of selenium-webdriver (http://docs.seleniumhq.org/projects/webdriver/). Both are “headless” (they don’t open a real browser) and are faster than Selenium with Firefox. Configuration can be a little tricky, so use the default Rack::Test or Selenium web drivers unless your test suite is large.

Configure Database Cleaner to Test JavaScript

Ordinarily, RSpec automatically cleans up the test database after each test. It uses the *transactional* cleanup strategy, relying on SQL `BEGIN TRANSACTION` and `ROLLBACK` statements to keep the database in a pristine state. Rolling back transactions makes for fast tests, as there is no database commit and I/O overhead. When Capybara uses the default Rack::Test web driver, it uses the fast transactional cleanup strategy.

When you use the Selenium web driver to test JavaScript, the tests and the browser run in separate threads, which means the transactional strategy can’t be used to clean the database. You might see problems, like testing user registration twice and finding that the user already exists the second time you test. To keep the database pristine between tests, you need to configure the database_cleaner (https://github.com/bmabey/database_cleaner) gem to use a *truncation* strategy with Capybara. Truncation means the database tables are emptied using the SQL `TRUNCATE TABLE` command instead of using rolling back transactions.

Here’s how to configure testing to use a truncation strategy with the Selenium web driver.

In the file spec/rails_helper.rb change this line from `true` to `false`:

```
config.use_transactional_fixtures = false
```

This lets you specify a more complex database cleanup strategy.

Add a new file spec/support/database_cleaner.rb:

```
RSpec.configure do |config|

  config.before(:suite) do
    DatabaseCleaner.clean_with(:truncation)
  end

  config.before(:each) do
    DatabaseCleaner.strategy = :transaction
  end

  config.before(:each, :js => true) do
    DatabaseCleaner.strategy = :truncation
  end

  config.before(:each) do
    DatabaseCleaner.start
  end

  config.append_after(:each) do
    DatabaseCleaner.clean
  end

end
```

This configuration uses truncation to reset the database before running the entire suite, just in case anything is in the database from a failed test run. Then each test will run using the fast transaction strategy, unless the `:js => true` tag is present, in which case the truncation strategy will be applied. The last two lines make sure DatabaseCleaner is used when tests are run.

Avdi Grimm has a helpful blog post Configuring database_cleaner (http://devblog.avdi.org/2012/08/31/configuring-database_cleaner-with-rails-rspec-capybara-and-selenium/) that goes into more detail. See also a blog post by Eric Saxby (http://www.livinginthepast.org/2013/11/24/today-we-went-to-the-bad-place.html).

Unit Tests

When you test features, you run integration tests. When you test a small part of the application in isolation, you run a unit test. You'll focus solely on a class, object, or method (often called the "system under test").

In a Rails application, developers often write unit tests for models. They may also test controllers and plain old Ruby objects (POROs) that are used for services. Less commonly, developers will test Rails mailers, helpers, views, or routes.

Unit tests follow a general pattern, often called the “Four-Phase Test” pattern, which looks like this:

```
test do
  setup
  exercise
  verify
  teardown
end
```

The setup phase prepares the system under test. Often this means instantiating an object. Here is an example:

```
user = User.new(email: 'user@example.com')
```

During the exercise phase, something is executed. Often this is a method call:

```
user.save
```

During verification, the result of the exercise is verified against the developer’s expectations.

```
expect(user.email).to eq 'user@example.com'
```

During teardown, the system under test is reset to its initial state. RSpec handles this automatically so you will seldom write code for the teardown phase.

Understanding the “Four-Phase Test” pattern will help you figure out complex code you see in unit tests.

Model Tests

Let’s look at unit tests for models. These are the most common unit tests you’ll see in Rails applications.

Before writing a unit test, we’ll create a User model in our tutorial application. We’ll use a Rails generator:

```
$ rails generate model user email:string
```

This creates a folder for model specs:

- spec/models

As well as several files:

- db/migrate/…_create_users.rb
- app/models/user.rb
- spec/models/user_spec.rb
- spec/factories/users.rb

Run the migration to set up the database for the development environment:

```
$ rake db:migrate
```

RSpec is configured to clone migrations automatically (see the “Run RSpec” section of this guide). If not, you’ll have to run `rake db:test:clone` .

Replace the sample code in the file spec/models/user_spec.rb to test our User model:

```
describe User do

  it "should be valid" do
    user = User.new(email: 'user@example.com')
    expect(user).to be_valid
  end

end
```

By convention, the RSpec file structure reflects the organization of the Rails app/ directory. The spec file spec/models/user_spec.rb corresponds to the User model in the file app/models/user.rb.

Unlike our feature tests, we can’t use user stories to define our unit tests. Instead, we describe a component in terms of its expected behavior, using `describe` and `it` instead of `feature` and `scenario` .

Instead of `feature` , we use the keyword `describe` to set up a block for the unit test. Notice that the descriptive title `describe User do` does not contain quotation marks. Our feature tests used quotes, such as `scenario 'Visit the Home Page' do` . When we test a Ruby object with a unit test, we don’t use quotes because we want RSpec to instantiate the object for testing.

Within the unit test, instead of `scenario` , each `it` statement sets up a test case (also called an “example”).

Synonyms

The keywords `describe` and `it` are part of the RSpec DSL (domain-specific language). RSpec has aliases for these keywords which can be used to improve readability of tests.

“describe” and “context”

The keyword `describe` accepts a string that names the component. The keyword `context` is an alias for `describe` . The `describe` keyword can be nested and often the `context` keyword is used within a `describe` block. You may see a `describe` block outlining the general function of a model or method; within the `describe` block you may see nested `describe` or `context` blocks grouping related test cases. You could use `context` to set up different `before` conditions, each initiating different test cases.

“it” and “specify”

The keyword `it` accepts a string that describes behavior we can expect from the component. Sometimes you will see `specify` as a synonym for `it` . Both are blocks that wrap an expectation.

The DRY Version

You will often see a version of a model spec that is less verbose. This “Don’t Repeat Yourself” version can grow to include many more test cases without duplication of code.

Here's a model spec that uses RSpec shortcuts to streamline the code:

```
describe User do

  before(:each) { @user = User.new(email: 'user@example.com') }

  subject { @user }

  it { should be_valid }

end
```

The `before` statement initializes a User model for each test case. It corresponds to the setup phase in the "Four-Phase Test" pattern. A few words of advice: As you seek to DRY up your specs, remember your goal is to write readable tests. If too much code is collapsed into a `before` statement, you may have to hunt through your spec file to find the hidden functionality. Don't make your code so compact that it becomes obscure; you can duplicate the setup phase for each test case without using `before` if it makes your code easier to follow.

The `subject` statement automatically makes the `@user` object the implicit subject of every `expect` statement, so you can say `expect be_valid` instead of `expect(user).to be_valid` .

The statement `expect be_valid` is clumsy English, so RSpec allows you to use `should` instead of `expect` , for `it { should be_valid }` . In mid-2012, RSpec replaced the older `should` syntax with a more verbose `expect` syntax to address some technical issues, but the RSpec maintainers still recommend (http://stackoverflow.com/questions/12260534/using-implicit-subject-with-expect-in-rspec-2-11/) using `should` for these simple one-line test cases.

As of RSpec 3.0, you can use the alternative syntax `is_expected.to` instead of `should` . If you use the alternative syntax, your model spec will look like this:

```
describe User do

  before(:each) { @user = User.new(email: 'user@example.com') }

  subject { @user }

  it { is_expected.to be_valid }

end
```

The choice of syntax is yours.

Matchers for Models

When we wrote feature tests, we used Capybara's RSpec matchers (http://rubydoc.info/github/jnicklas/capybara/master/Capybara/RSpecMatchers). For our unit tests, we'll primarily use the matchers from the rspec-expectations or rspec-rails gems. There are too many to list here, but take a look at the lists here:

- matchers from the rspec-expectations gem (https://github.com/rspec/rspec-expectations)
- matchers from the rspec-rails gem (http://rubydoc.info/gems/rspec-rails/RSpec/Rails/Matchers)

Here's additional documentation for RSpec Expectations Matchers (http://rubydoc.info/gems/rspec-expectations/RSpec/Matchers).

You may want to add the shoulda-matchers (https://github.com/thoughtbot/shoulda-matchers) gem for even more matchers. These are particularly useful for testing models with ActiveRecord database associations (such as `have_many` associations).

When you test a model, you'll have three things to test:

- instance methods – for example, `@user.email`
- validations
- class methods – for example, `User.all`

Let's look closely at examples for each.

Testing Instance Methods

You can test instance methods in two ways. First, test if the method exists. Second, test if the method returns the value you expect.

Here's a model spec that tests the `user.email` instance method in two ways:

```
describe User do

  before(:each) { @user = User.new(email: 'user@example.com') }

  subject { @user }

  it { should respond_to(:email) }

  it "#email returns a string" do
    expect(@user.email).to match 'user@example.com'
  end

end
```

The first case uses `respond_to` to see if the method exists, using all the RSpec shortcuts. The second case uses the verbose RSpec syntax to test if the `@user.email` method returns the email address we expect.

Notice the hash mark in the title of the second test case: `#email returns a string`. Following Rails conventions, we use the hash mark to indicate that `#email` is an instance method. The hash mark has no effect on RSpec; it is simply a convention for documentation.

Testing Validations

You can think of validation tests as CYA ("cover your ass") testing. Automated tests provide thoroughness that busy programmers can't provide with manual testing. Even if you like to try a few things before deploying your application, you're probably not going to test every permutation of bad input data. That's the role for validation tests. When you add a validation to a model, write validation tests and you'll never have to manually test obscure edge cases.

We'll add a validation to our model by changing the file app/models/user.rb:

```
class User < ActiveRecord::Base
  validates :email, presence: true
end
```

Here we ensure that an email address is present before a user is saved to the database.

Here's a model spec that includes a test case for our validation:

```
describe User do

  before(:each) { @user = User.new(email: 'user@example.com') }

  subject { @user }

  it { should be_valid }

  describe "when email is blank" do
    before { @user.email = " " }
    it { should_not be_valid }
  end

end
```

The first case tests that the user created in the `before` method is valid. Every ActiveRecord model has a `valid?` method which returns true when all validation passes. The `be_valid` matcher from the rspec-rails gem (http://rubydoc.info/gems/rspec-rails/RSpec/Rails/Matchers) calls the `valid?` method and returns true or false.

The second case initializes a new user with a blank email which we expect to fail to validate. Notice that we can nest the `describe` blocks as needed.

Testing Class Methods

Here's a model spec that tests the `User.all` class method:

```
describe User do

  before(:each) { @user = User.new(email: 'user@example.com') }

  it ".all returns a list of users" do
    @user.save
    expect(User.all.count).not_to eq 0
  end

end
```

For this test case, we have to save the user to the database with `@user.save` before we query the database with `User.all.count`. Then we expect the result of `User.all.count` to be greater than zero.

In the test case title, we use the dot (period, or full stop) to indicate that `.all` is a class method. Again, this is simply a convention for documentation.

This is a trivial example for illustration only. There's no need to write a test for ActiveRecord class methods such as `User.all` because you can safely assume that ActiveRecord always works as intended. However, if you add your own class methods to your model, you will need to add tests of class methods.

Other Tests

The art of testing lies in making good choices about what to test. The architecture of your Rails application will determine what you test. It's common to write feature tests because they will test the entire application from the viewpoint of the user. However, feature tests are slow because they require interaction with a web browser and database.

It is also common to see tests for models, especially when programmers follow the "skinny controller, fat model" practice and move functionality from controllers into models. If developers implement a service-oriented architecture, and move functionality from models into plain old Ruby objects (POROs) in a services/ folder, you will see tests for each service object. Tests for models and service objects can execute quickly, especially if there is no interaction with a database or external services.

If you do nothing else, write feature tests plus unit tests for models and POROs. Other tests are less common. Here we'll look at other tests and consider when they might be used.

Controller Tests

There is controversy about the merits of testing controllers. If your controller methods are "skinny" and stripped of excess application logic, they do no more than transfer an HTTP request to another component of your application before rendering a view. Testing a controller method will only assure it responds and serves a page as it should, which may already be tested with a feature test.

On the other hand, feature tests are slow, controller tests are faster, and you may want to test all the possible routes provided by a controller. Piotr Solnica made a compelling argument for Yes, You Should Write Controller Tests (http://solnic.eu/2012/02/02/yes-you-should-write-controller-tests.html) (but see a lengthy debate that followed in his blog comments). We'll examine a controller test here, but in all likelihood, you won't test a controller that only offers the seven standard RESTful actions. You are more likely to test additional non-RESTful actions, where you will use the same principles.

Our rails-bootstrap starter app has a VisitorsController with a single action, `#index`, that renders the application home page.

Create a folder for controller tests:

```
$ mkdir spec/controllers
```

We can create a controller test in the file spec/controllers/visitors_controller_spec.rb:

```
describe VisitorsController do

  describe 'GET #index' do

    it "renders the :index template" do
      get :index
      expect(response).to render_template :index
    end

  end

end
```

Notice that the descriptive title `describe VisitorsController do` does not contain quotation marks. When we test a controller with a unit test, we don't use quotes because we want RSpec to instantiate the controller for testing. If you use quotes, you'll get a runtime error "controller is nil."

This simple controller test is very similar to our other unit tests. When we test the controller, we test each method. The title for the test case is `GET #index`. By convention, we identify the HTTP request method `GET` and prefix the instance method name with a hash mark. If you have the seven RESTful actions in a controller, you'll have tests for:

- GET #index
- GET #show
- GET #new
- GET #edit
- POST #create
- PATCH #update
- DELETE #destroy

Our test case indicates that we expect the `GET #index` action to render the `:index` template (the file app/views/visitors/new.html.erb).

There is no setup phase to this test (no instance variables are needed to render the page). The exercise phase uses the RSpec `get` keyword to retrieve the `:index` page. In the verification phase, we expect the response to render the `:index` template.

This is the simplest possible test of a controller. Controller testing quickly becomes complicated.

In the setup phase, here's what you need to do:

- GET #index – create an array of instance variables
- GET #show – create an instance variable
- GET #new – nothing extra required
- GET #edit – create an instance variable
- POST #create – create a params hash
- PATCH #update – create an an instance variable and a params hash
- DELETE #destroy – create an instance variable

In the verify phase, you'll need to:

- GET #index – check that we have the collection of objects we expect
- GET #show – check that an instance variable has the value we expect
- GET #new – check that the controller was instantiated with the instance variable we expect
- GET #edit – check that the controller has processed the instance variable we expect
- POST #create – check that the object was added and invalid attributes were rejected
- PATCH #update – check that the object changed and invalid attributes were rejected
- DELETE #destroy – check that the object was deleted

If your controller uses *nested routes* you'll need to create multiple objects in the setup phase.

If you controller renders JSON or XML output instead of HTML, you'll need an extra twist to verify the content is what is expected.

If access to your controller is restricted by authentication or authorization requirements, you'll have to set session ids and create tests for users in different roles.

As you can see, controller tests quickly become complicated.

It's not possible to cover all these controller examples in this Quickstart Guide. Aaron Sumner's book Everyday Rails Testing with RSpec (http://everydayrails.com/) devotes two lengthy chapters to controller testing. The book is excellent; please refer to it for the details on controller testing. Before you do so, consider carefully if controller tests will add any value, as they are difficult to write and maintain.

Request Specs

We've looked closely at feature tests using Capybara to simulate user interaction with a web application. RSpec offers a second kind of integration test, called *request specs*. Instead of interacting with a web page, RSpec request specs interact directly with the Rails API. Where Capybara offers `visit` , request specs use `get` . Where Capybara verifies the content on a `page` , request specs examine a `response` .

For example, with Capybara you might express:

```
visit root_path
expect(page).to have_content 'Welcome'
```

A request spec would look like this:

```
get '/'
expect(response.body).to have_content 'Welcome'
```

Request specs are a holdover from early versions of RSpec that didn't fully integrate Capybara. They are faster than feature tests using Capybara but are much more limited. If your application offers an API, you might test it with request specs. The creator of Capybara says, Do not test APIs with Capybara (http://www.elabs.se/blog/34-capybara-and-testing-apis). You can read the article Rails API Testing Best Practices (http://matthewlehner.net/rails-api-testing-guidelines/) about using request specs.

View Specs

RSpec supports testing of Rails views. For a view spec of the application home page, create a file spec/views/visitors/index.html.erb_spec.rb:

```
describe 'visitors/index.html.erb' do
  it 'displays home' do
    render
    expect(rendered).to match /Welcome/
  end
end
```

A view spec simply renders a Rails view template in isolation. View specs are rarely used. When they are used, it is typically to make sure a page contains (or does not contain) an element under certain conditions, such as an error message, or a restricted section of a page. These elements are usually tested with feature tests.

Helper Specs

If you write custom Rails view helpers (http://guides.rubyonrails.org/action_view_overview.html#overview-of-helpers-provided-by-action-view), you can write RSpec helper specs. See the RSpec documentation for a Helper spec (https://www.relishapp.com/rspec/rspec-rails/v/3-0/docs/helper-specs/helper-spec). The helper specs go in a folder spec/helpers/ and RSpec will set up the necessary conditions to test a view helper.

Most developers avoid complex logic in helpers. If you only have simple HTML markup in helpers, it doesn't need to be tested.

Routing Specs

Routing specs go in the spec/routing folder. The RSpec documentation (https://www.relishapp.com/rspec/rspec-rails/v/3-0/docs/routing-specs) sums up everything you need to know about routing specs: "Simple apps with nothing but standard RESTful routes won't get much value from routing specs, but they can provide significant value when used to specify customized routes, like vanity links, slugs, etc.”

Testing External Services

Many web applications interact with the API of an external service. When you write tests, often you don’t want to actually connect with an external service, even for feature tests. External services can be slow or may fail due to connectivity issues. Or the service may not have a test mode.

Common practice is to write stubs to fake the response of an external service. Several gems are available to simplify the process. Each works to replace real HTTP requests with a fake request and response. Take a look at:

- VCR (https://github.com/vcr/vcr)
- WebMock (https://github.com/bblimke/webmock)
- FakeWeb (https://github.com/chrisk/fakeweb)
- ShamRack (https://github.com/mdub/sham_rack)
- Puffing Billy (https://github.com/oesmith/puffing-billy)

VCR is particularly interesting because it records actual interactions with an external service and replays them during your tests. If it is difficult to manually reproduce the response from an external service, you can use VCR to capture the interaction for playback.

A word of caution about stubbing external services: If the API of an external service changes, your tests may continue to pass even though your application will fail in production. You aren’t protected from external API changes. If you can, make sure you have integration tests that verify the external service behaves as expected.

Testing Email

You might wonder how we test email. After all, email ends up on someone else’s computer, so how do we test that?

We can test mailers in isolation with unit tests. And we can write integration tests to verify that email is sent as part of a feature. The secret is that we must set up our test environment so emails are not actually delivered. Instead, they are saved to an array in memory that can be examined in the verification phase of a test.

Configuring the Test Environment

Open the file config/environments/test.rb and confirm the following statement is present:

```
Rails.application.configure do
  .
  .
  .
  # Tell Action Mailer not to deliver emails to the real world.
  # The :test delivery method accumulates sent emails in the
  # ActionMailer::Base.deliveries array.
  config.action_mailer.delivery_method = :test
  .
  .
  .
end
```

By default, `config.action_mailer.delivery_method` is set to `:test`.

Create a Mailer

We’ll use the Rails generator to add a simple mailer to our starter application:

```
$ rails generate mailer UserMailer
```

The rails generate command will create two files and one folder:

- app/mailers/user_mailer.rb
- app/views/user_mailer
- spec/mailers/user_mailer_spec.rb

Add a send_email method to the mailer by editing the file app/mailers/user_mailer.rb:

```
class UserMailer < ActionMailer::Base
  default from: 'from@example.com'

  def send_email(user)
    mail(to: user.email, from: 'sender@example.com', :subject => 'You Have Mail')
  end
end
```

Create a mailer view app/views/user_mailer/send_email.text.erb:

```
This is a test message.
```

Refer to the book Learn Ruby on Rails (http://learn-rails.com/learn-ruby-on-rails.html) or the RailsGuide Action Mailer Basics (http://guides.rubyonrails.org/action_mailer_basics.html) if you want a better example of a mailer. This mailer just fits our needs for an example.

Unit Test for Mailers

Testing a Rails mailer is very similar to other unit tests.

Edit the file spec/mailers/user_mailer_spec.rb to create a mailer spec:

```
describe UserMailer do

  describe '#send_email' do

    before(:each) do
      @user = FactoryGirl.create(:user)
      UserMailer.send_email(@user).deliver
    end

    it 'sends an email' do
      expect(ActionMailer::Base.deliveries.count).to eq 1
    end

    it 'sends an email to the correct recipient' do
      expect(ActionMailer::Base.deliveries.first.to).to match [@user.email]
    end

    it 'sends an email from the correct sender' do
      expect(ActionMailer::Base.deliveries.first.from).to match ['sender@example.com']
    end

    it 'sends an email with the correct subject' do
      expect(ActionMailer::Base.deliveries.first.subject).to match 'You Have Mail'
    end

  end

end
```

During the setup phase, we use the `before(:each)` block to instantiate a User object and deliver a mail message.

Then we verify:

- the email is delivered
- the recipient is correct
- the sender is correct
- the subject line is correct

We verify by checking the first element in the array `ActionMailer::Base.deliveries` and calling `to` , `from` , and `subject` methods. We only need the simple matchers provided by RSpec.

Integration Test

Your unit test only confirms that the mailer delivers the message you expect. It may be more important to confirm that a message gets sent in response to the user's behavior, which requires an integration test.

You can borrow the expectations from the mailer unit test for use in your feature tests.

I won't modify the starter application to include a complete example, but imagine a web application with a page that contains a "Contact Us" form. When the visitor submits the form, an email message goes to the site owner. Here's an integration test for the feature:

```
feature 'Contact Page' do

  scenario 'Send a Message' do
    @contact = FactoryGirl.build(:contact)
    visit new_contact_path
    fill_in 'contact_name', :with => 'Test User'
    fill_in 'contact_email', :with => 'user@example.com'
    fill_in 'contact_content', :with => 'test'
    click_button 'submit'
    expect(ActionMailer::Base.deliveries.count).to eq 1
  end

end
```

You can see I'm using the same expectation I used in the unit test: `expect(ActionMailer::Base.deliveries.count).to eq 1`.

Email Spec Matchers

For convenience and a greater choice of matchers, you can add the email-spec (https://github.com/bmabey/email-spec) gem.

Add email-spec to your Gemfile like this. Make sure you've got an underscore (not a hyphen) in the gem name:

```
.
.
.
group :test do
  .
  .
  .
  gem 'email_spec'
end
```

Run `$ bundle install` to download the gem.

You'll need to configure RSpec to add email-spec. You can add email-spec to a file spec/support/emailspec.rb or make changes to the spec/rails_helper.rb file:

```
require "email_spec"
RSpec.configure do |config|
  config.include(EmailSpec::Helpers)
  config.include(EmailSpec::Matchers)
end
```

Take a look at the email-spec README (https://github.com/bmabey/email-spec) and documentation (http://rubydoc.info/gems/email_spec/1.5.0/frames) for a list of all the matchers and helpers that are available.

We can use email-spec matchers in the unit test we created earlier.

Edit the file spec/mailers/user_mailer_spec.rb to use email-spec matchers:

```
describe UserMailer do

  describe '#send_email' do

    before(:each) do
      @user = FactoryGirl.create(:user)
      UserMailer.send_email(@user).deliver
    end

    it 'sends an email' do
      expect(ActionMailer::Base.deliveries.count).to eq 1
    end

    it 'sends an email to the correct recipient' do
      expect(open_last_email).to be_delivered_to @user.email
    end

    it 'sends an email from the correct sender' do
      expect(open_last_email).to be_delivered_from 'sender@example.com'
    end

    it 'sends an email with the correct subject' do
      expect(open_last_email).to have_subject 'You Have Mail'
    end

  end

end
```

The method `open_last_email` is a helper that substitutes for `ActionMailer::Base.deliveries.first`. It returns the most recent email.

We're using the matchers `deliver_to`, `deliver_from`, and `have_subject`.

You can also use matchers `cc_to`, `bcc_to`, `have_body_text`, and `have_header` and a few others.

I'm not sure the email-spec gem adds much value, since RSpec's built-in matchers are adequate in most cases.

Troubleshooting

The default settings in the file config/environments/test.rb should be adequate for testing your mailers. If not, in the setup phase of your tests you can configure ActionMailer directly.

Here's an example:

```
describe UserMailer do

  describe '#send_email' do

    before(:each) do
      ActionMailer::Base.delivery_method = :test
      ActionMailer::Base.perform_deliveries = true
      ActionMailer::Base.deliveries = []
      @user = FactoryGirl.create(:user)
      UserMailer.send_email(@user).deliver
    end

    after(:each) do
      ActionMailer::Base.deliveries.clear
    end

  .
  .
  .
  end

end
```

The `after(:each)` block makes sure messages are deleted after each test.

Sample Data

In this section we’ll look closely at the setup phase of testing, specifically creating sample data. Setup can be challenging because we need to create the data and conditions necessary for our tests outside of the normal application flow.

Sample Data in Integration Tests

Sample data is not needed for a feature test if the test recreates the entire application flow. For example, if Capybara fills in a sign up form before logging in and testing a feature, the application will create the User object that is required for the test. You can speed up your tests by eliminating the sign up and log in steps and directly creating an authenticated User object in your tests. This is a common use case for sample data in integration testing.

Sample Data in Unit Tests

Unit tests need sample data when there are dependencies on objects outside the system under test. In principle, a unit test should test an object in isolation, without any external dependencies. To achieve that, developers use *stubs* or *mocks* (also called test doubles (http://xunitpatterns.com/Test%20Double.html)) to stand in for external dependencies. Stubs fake the response of an external dependency. Mocks are fake objects that return predetermined responses. (For a detailed explanation see stubs (http://xunitpatterns.com/Test%20Stub.html) and mocks (http://xunitpatterns.com/Mock%20Object.html).) With stubs and mocks, tests run faster and developers will refine an application design to reduce coupling and complexity.

It takes time to write stubs and mocks and lots of experience to use them correctly, so as a beginner, you probably won't write stubs and mocks. Instead, use PORO's, fixtures, or factories to create sample data for your unit tests. You won't have "pure" unit tests that are free of all external dependencies but you'll still have useful unit tests.

POROs

So far, we've created sample data using plain old Ruby objects (POROs). For example, we tested our User model's `User.all` class method:

```
describe User do

  before(:each) { @user = User.new(email: 'user@example.com') }

  it ".all returns a list of users" do
    @user.save
    expect(User.all.count).not_to eq 0
  end

end
```

We simply initialized a new User in the `before` statement. It works well for this simple model. But imagine our User has a dozen data attributes:

```
@user = User.new(email: 'user@example.com', honorific: 'Mr.', first_name: 'Testy', last_name: '
Tester', nickname: 'testdude', active: true, role: 0)
```

Imagine we need to initialize the User object in a dozen different spec files. And, just to complicate things, thirty days into our project we decide to add a new attribute to the User model. We'd have to change the initialization of the User model in a dozen files. Obviously, this calls for modularity and DRYing up our code.

Programmers have solved this problem using fixtures or factories.

Fixtures

RSpec allows you to create a folder spec/fixtures/ containing YAML files that look like this:

```
user_one:
  email: user@example.com
```

If the file is named spec/fixtures/users.yml, and you have a User model, an object will be available for use in a test:

```
describe User do
  fixtures :users

  it ".all returns a list of users" do
    user = users(:user_one)
    user.save
    expect(User.all.count).not_to eq 0
  end

end
```

In current practice, developers seldom use fixtures with RSpec. In rare situations, fixtures may be used sparingly to speed up slow test suites.

Fixtures consolidate the attributes needed to create a model in one place. But fixtures still must be updated when attributes are added to a model. They are an external dependency that must be kept in sync with a model's changes. Fixtures are also not very flexible. If you need instances of a model with different attributes, you must duplicate the entries in a fixtures file. For example, you might need a user with a nil email address, in which case you need fixtures for both `user_one:` and `user_with_nil_email:`. Lastly, fixtures bypass ActiveRecord validations in a model, which means you can't use fixtures to test validations.

Instead of fixtures, developers use factories.

Why Use Factories?

A factory is an object that creates other objects. Factories used in RSpec testing are examples of Object Mother (http://martinfowler.com/bliki/ObjectMother.html) or Test Data Builder (http://nat.truemesh.com/archives/000714.html) patterns.

Factories create objects independently of a model. That means a factory won't break if you add an attribute to a model. There's no need to keep a factory in sync with a model.

You can use a factory to create an object with specific attributes you need in your test, for example, a user with a nil email attribute. The attributes that are significant to the test can be defined in the test and passed to the factory as parameters. One factory can create multiple objects, each with different attributes. In fact, you can use a factory to create dozens or hundreds of objects with fake data or attributes in a sequence (like invoice numbers).

When you use FactoryGirl (https://github.com/thoughtbot/factory_girl), you have the option of saving your object to the database and testing a model's validations (which is slow) or building your object in memory only (which is faster).

FactoryGirl is the most popular gem for creating sample data in RSpec, but some developers prefer fabrication (http://www.fabricationgem.org/) or others (https://www.ruby-toolbox.com/categories/rails_fixture_replacement).

FactoryGirl

You’ve already installed the factory_girl (https://github.com/thoughtbot/factory_girl) gem. For simple use, no configuration is required.

Create a folder for factories:

```
$ mkdir spec/factories
```

FactoryGirl isn’t picky about filenames, as long as all factory files are in the spec/factories/ folder or subfolders. You could have a single file containing all factory definitions. Or follow the convention and create a file for each factory definition. By convention, the filename is the plural of the object created by the factory. A file that contains a User factory definition should be named users.rb.

Creating a Factory

The User model is needed in many of our tests, so create a User factory in a file spec/factories/users.rb:

```
FactoryGirl.define do
  factory :user do
    email 'user@example.com'
  end
end
```

Now we’ve got a factory that will create a User object. We could include sample data for every attribute from the real User model but that’s not necessary; we only need to create sample data for any attributes that are touched by our tests.

Testing With a Factory

Let’s modify a unit test for the User model to use the factory. Replace the sample code in the file spec/models/user_spec.rb:

```
describe User do

  before(:each) { @user = FactoryGirl.create(:user) }

  it ".all returns a list of users" do
    expect(User.all.count).not_to eq 0
  end

end
```

You’ll recall this tests the `User.all` class method. In the earlier example, we had:

```
describe User do

  before(:each) { @user = User.new(email: 'user@example.com') }

  it ".all returns a list of users" do
    @user.save
    expect(User.all.count).not_to eq 0
  end

end
```

Now we have:

```
before(:each) { @user = FactoryGirl.create(:user) }
```

And we eliminate `@user.save` .

In this example, we need to find the object in the database for the class method `User.all` . We use the `FactoryGirl.create` method to get a User object that is saved to the database. In the earlier example, we used a `@user.save` method to save the object to the database before our verification step. We no longer need `@user.save` because `FactoryGirl.create` instantiates and saves the object in the database.

Build Versus Create

FactoryGirl offers three options for instantiating an object:

- `FactoryGirl.create` – saves your object to the database (plus any associated objects)
- `FactoryGirl.build` – builds your object in memory only (but saves any associated objects to the database)
- `FactoryGirl.build_stubbed` – builds your object (and associated objects) in memory only

As you might guess, `FactoryGirl.create` is not as fast as `FactoryGirl.build` or `FactoryGirl.build_stubbed` .

Try changing `FactoryGirl.create(:user)` to `FactoryGirl.build(:user)` in the example above. Run RSpec and you'll see the test fail because `FactoryGirl.build` does not save the object to the database. Even if we'd like speedy tests, there are times we need `FactoryGirl.create` .

Let's look at an example that uses both `FactoryGirl.build` and `FactoryGirl.create` .

Replace the sample code in the file spec/models/user_spec.rb:

```
describe User do

  it ".all returns a list of users" do
    user = FactoryGirl.create(:user)
    expect(User.all.count).not_to eq 0
  end

  it "should be valid" do
    user = FactoryGirl.build(:user)
    expect(user).to be_valid
  end

end
```

Instead of using `before` to instantiate the User object, we instantiate the User object in the set up phase of each test case. When we test the `User.all` class method, we use `FactoryGirl.create` , which saves the object to the database.

We use `FactoryGirl.build` method before testing the object to determine if is is valid. This illustrates an interesting feature of FactoryGirl. Even though `FactoryGirl.build` does not save the object to the database, the object built in memory has all the validation methods of the User model. This is an advantage of FactoryGirl over fixtures; you cannot test validation with fixtures.

Overriding Attributes

FactoryGirl offers a significant advantage over fixtures by supporting custom attributes for each test case.

You'll recall we added a validation to our model in the file app/models/user.rb:

```
class User < ActiveRecord::Base
  validates :email, presence: true
end
```

Let's see if we can create a user with a blank email.

```
describe User do

  it "should be valid" do
    user = FactoryGirl.build(:user)
    expect(user).to be_valid
  end

  it "should be invalid with a blank email address" do
    user = FactoryGirl.build(:user, email: nil)
    expect(user).to_not be_valid
  end

end
```

In the `FactoryGirl.build` method, we specify an `email: nil` parameter to override the default attribute in the factory.

When we run RSpec, both test cases will pass. In the second case, we expect the object `to_not be_valid`, which verifies that the validation fails with a blank email address.

Sequenced Data

FactoryGirl lets you generate objects with attributes that are slightly different each time.

For example, in the file spec/factories/users.rb:

```
FactoryGirl.define do
  factory :user do
    sequence(:email) { |n| "user#{n}@example.com"}
  end
end
```

Each time the User object is created, the email address will be different:

- user1@example.com
- user2@example.com
- user3@example.com
- …

This could be useful in generating invoice numbers, or when multiple users must be created with unique email addresses.

Fake Data

Several gems are available (https://www.ruby-toolbox.com/categories/random_data_generation) that will generate fake data for a Rails application and each can be used with FactoryGirl. The faker (https://github.com/stympy/faker) gem is the most popular. Add it to your Gemfile like this and run `$ bundle install`:

```
.
.
.
group :test do
  .
  .
  .
  gem 'faker'
end
```

You'll need to consult the Faker README (https://github.com/stympy/faker) for a list of all the fake data you can generate.

Here's an example of using Faker in a factory, in the file spec/factories/users.rb:

```
require 'faker'

FactoryGirl.define do
  factory :user do
    email { Faker::Internet.email }
  end
end
```

Notice that `require 'faker'` is needed to enable use of the `Faker` classes.

Each time the object is created, a different fake email address will generated.

Associations

FactoryGirl is adept at handling ActiveRecord associations (http://guides.rubyonrails.org/association_basics.html).

For example, if a User model has an associated Phone model, and each has a factory, you can specify the association in the factory definition:

```
FactoryGirl.define do
  factory :user do
    email 'user@example.com'
    phone
  end
end
```

As you might expect, the FactoryGirl associations feature has additional complexity, so you'll want to refer to the documentation (https://github.com/thoughtbot/factory_girl/blob/master/GETTING_STARTED.md#associations) for details.

FactoryGirl Shortcuts

FactoryGirl is not immune from Rails developers' urges to abbreviate code wherever possible. If you'd like to eliminate extra typing when writing `FactoryGirl.build` and `FactoryGirl.create` you can configure FactoryGirl for a shortcut.

Create a file spec/support/factory_girl.rb and add this configuration option:

```
RSpec.configure do |config|
  config.include FactoryGirl::Syntax::Methods
end
```

Alternatively you can add the configuration to the file spec/rails_helper.rb.

Then you can shorten the FactoryGirl build or create syntax:

```
describe User do

  it ".all returns a list of users" do
    user = create(:user)
    expect(User.all.count).not_to eq 0
  end

  it "should be valid" do
    user = build(:user)
    expect(user).to be_valid
  end

end
```

The directives `FactoryGirl.build` and `FactoryGirl.create` become simply `build` and `create` . You'll often see this in spec files. You may prefer to see an explicit reference to the `FactoryGirl` class. Don't feel you have to use the shortcut.

FactoryGirl Resources

FactoryGirl is a mature and full-featured project. The README for the factory_girl (https://github.com/thoughtbot/factory_girl) is deceptively short. The useful documentation for FactoryGirl is on the FactoryGirl wiki (https://github.com/thoughtbot/factory_girl/wiki), particularly the lengthy Getting Started (https://github.com/thoughtbot/factory_girl/blob/master/GETTING_STARTED.md) guide.

Selective Testing

Most of the time, when you are writing tests, you won't want to run the entire test suite. You'll be focused on a particular feature and you'll want to skip tests for features that haven't changed.

RSpec accommodates your workflow by providing, appropriately, `skip` and `focus` tags that can be applied to any test. Use `skip` and `focus` tags when you are working on a particular feature, then remove the tags so you can periodically run your regression tests on the entire project.

Skip

RSpec offers a number of ways to indicate that a test should be skipped and not executed (see the documentation (https://relishapp.com/rspec/rspec-core/v/3-0/docs/pending-and-skipped-examples/skip-examples) for all the options).

Here is a simple example:

```ruby
feature 'Home page' do

  scenario 'visit the home page' do
    skip 'skip a slow test'
  end

end
```

You’ll see:

```
$ rspec
Home page
  visit the home page (PENDING: skip a slow test)

Pending:
  Home page visit the home page
    # skip a slow test
    # ./spec/features/visitors/home_page_spec.rb:3
```

RSpec gives you several more ways to indicate that a test should be skipped. Here are some examples from a unit test of a model:

```ruby
describe User do
  skip 'not implemented yet' do
  end

  it 'does something', :skip => true do
  end

  it 'does something', :skip => 'skipped for some reason' do
  end

  it 'does something else' do
    skip
  end

  it 'does something else' do
    skip 'skipped for another reason'
  end
end
```

The skipped tests are always listed in the output (in yellow when colors are enabled) so you won’t forget that some of your tests are disabled.

Skipping With an ‘X’

Using the single character `x` , RSpec gives you a shortcut to bypass tests.

Consider this test:

```
describe User do

  it 'expects the truth' do
    expect(true).to be(true)
  end

end
```

Add an `x` to `describe`:

```
xdescribe User do

  it 'expects the truth' do
    expect(true).to be(true)
  end

end
```

You'll get this output:

```
$ rspec
User
  expects the truth (PENDING: Temporarily skipped with xdescribe)

Pending:
  User expects the truth
    # Temporarily skipped with xdescribe
    # ./spec/models/user_spec.rb:3
```

This trick works with the directives `describe`, `it`, `context`, and `scenario`. Oddly, it doesn't work with `feature`.

Skipping With Tags

RSpec allows you to add tags to any test or group of tests. The RSpec documentation describes tags as user-defined metadata (https://www.relishapp.com/rspec/rspec-core/docs/metadata/user-defined-metadata). Tags are used to alter how tests run. You saw tags used in feature tests when we added the `:js` tag to a scenario to trigger use of Capybara for testing JavaScript:

```
feature 'Home Page' do

  scenario 'Visit the Home Page', :js do
    visit root_path
    expect(page).to have_content 'Welcome'
  end

end
```

We can use tags to selectively skip tests. For example, tag all your slow tests like this:

```
feature 'Home Page' do

  scenario 'Visit the Home Page', :slow do
    visit root_path
    expect(page).to have_content 'Welcome'
  end

end
```

Notice that we precede the tag with a `:` (colon) character to mark the tag as a Ruby symbol.

Then run your tests, excluding the slow tests:

```
$ rspec --tag ~slow
```

By adding the argument `--tag ~slow` to the `rspec` command, you will exclude all tests marked with the `slow` tag. Notice that you prepend the `~` (tilde) character to indicate tests with the tag `slow` tag should be excluded. Leave off the tilde character, `rspec --tag slow`, and only the slow tests will run.

Focus

You'll use the `skip` tag to bypass a test that is slow or not yet implemented. But it is not practical to mark every test with `skip` when you want to bypass your entire test suite. Instead, use the `focus` tag.

To use the `focus` tag, you must add a configuration setting to the spec/rails_helper.rb file:

```
RSpec.configure do |config|
  .
  .
  .
  config.filter_run :focus
end
```

The `config.filter_run` setting allows you to specify a tag for tests that you want to run. Tests without the tag will be filtered out (excluded).

Mark the tests that you want to run with the `focus` tag, using the `:` (colon) character to mark the tag as a Ruby symbol. Here is an example:

```
feature 'Home Page' do

  scenario 'Visit the Home Page', :focus do
    visit root_path
    expect(page).to have_content 'Welcome'
  end

end
```

Run your tests with the `rspec` command:

```
$ rspec
Run options: include {:focus=>true}

Home page
  Visit the Home Page

Finished in 0.22556 seconds (files took 1.82 seconds to load)
1 example, 0 failures
```

Only the test marked with the `focus` tag will run. The `config.filter_run` setting in the spec/rails_helper.rb file will exclude all other tests.

You can also mark a group of tests with a tag:

```
feature 'Home Page', :focus do

  scenario 'Visit the Home Page' do
    visit root_path
    expect(page).to have_content 'Welcome'
  end

end
```

The behavior will be the same as the previous example.

Focus With an 'F'

If the `config.filter_run` setting in the spec/rails_helper.rb is set to run only tests tagged `focus` , you can use the single character `f` as a shortcut to mark tests to run.

Here we've added `f` to `describe` :

```
fdescribe User do

  it 'expects the truth' do
    expect(true).to be(true)
  end

end
```

You'll get this output:

```
$ rspec
Run options: include {:focus=>true}

User
  expects the truth

Finished in 0.04461 seconds (files took 1.8 seconds to load)
```

Only the tests with the directive `fdescribe` will run.

This trick works with the directives `describe` , `it` , or `context` . It doesn't work with `feature` or `scenario` .

Personally, I feel directives `fdescribe` , `fit` , and `fcontext` make tests much less readable. You should only use this trick as a quick-and-dirty hack when you're in the middle of writing tests. Don't leave the unreadable `fdescribe` , `fit` , and `fcontext` in place when you commit your code to a repository where others will see it.

Use Any Tag

You can mark your tests with any tag. For example, use the tag `home` in this example:

```
feature 'Home Page', :home do

  scenario 'Visit the Home Page' do
    visit root_path
    expect(page).to have_content 'Welcome'
  end

end
```

Remove the `config.filter_run` setting in the spec/rails_helper.rb.

Now you can selectively run only the tests tagged `home` :

```
$ rspec --tag home
Run options: include {:home=>true}

Home Page
  visit the home page

Finished in 0.13307 seconds (files took 1.76 seconds to load)
1 example, 0 failures
```

Only the tests with the tag `home` will run.

Use the tilde character for the inverse with `rspec --tag ~home` and all the tests tagged `home` will be excluded, running everything else.

You can apply multiple tags to any test (or group of tests). For example:

```
feature 'Home Page', :home, :slow do

  scenario 'Visit the Home Page' do
    visit root_path
    expect(page).to have_content 'Welcome'
  end

end
```

Use multiple tags if you want to organize your tests into categories or collections for selective testing.

Resources for RSpec

Testing is one of the areas of Rails development that is unevenly documented. You can find many blog posts from senior developers that debate finer points of advanced testing, plus a few introductory articles that show how to set up RSpec. But there's a gap when it comes to solid advice for getting started with testing.

Compounding the documentation problem is the changing syntax of RSpec. Tutorials written before 2013 offer examples using the now-deprecated `should` instead of `expect` syntax. And it doesn't help that there are many ways to write the same test in RSpec.

Books

The best resource I've seen for learning to use RSpec is a book by Aaron Sumner (http://www.aaronsumner.com/) titled Everyday Rails Testing with RSpec (https://leanpub.com/everydayrailsrspec). It's not expensive and Aaron is updating it regularly.

I also like the book Rails Test Prescriptions (http://pragprog.com/book/nrtest/rails-test-prescriptions) by Noel Rappin (http://www.noelrappin.com/). But the book hasn't been updated since 2011. It's the same problem with The RSpec Book (http://pragprog.com/book/achbd/the-rspec-book) which hasn't been updated since 2010.

Michael Hartl’s Rails Tutorial (http://ruby.railstutorial.org/) shows how to build a Rails application using test-driven development with RSpec. The examples are good and the syntax is up to date.

The books above offer examples of RSpec but don’t provide much background in the concepts and terminology of testing. The book XUnit Test Patterns (http://xunitpatterns.com/index.html) by Gerard Meszaros and his xUnit Patterns (http://xunitpatterns.com/index.html) website show the software design patterns and explain the terminology of testing. The book Working Effectively with Legacy Code (http://www.amazon.com/Working-Effectively-Legacy-Michael-Feathers/dp/0131177052) by Michael Feathers was one of the first books to show how unit tests are an essential development practice. Finally, Test Driven Development: By Example (http://www.amazon.com/Test-Driven-Development-By-Example/dp/0321146530) by Kent Beck is the classic introduction to TDD. All three books give examples in Java or C-family languages.

Best Practices and Advice

The best single source of RSpec advice is the site Better Specs (http://betterspecs.org/). It shows the right and wrong ways to use RSpec. You can read it in a few minutes to learn best practices.

You’ll find useful advice in these articles:

- How We Test Rails Applications (http://robots.thoughtbot.com/how-we-test-rails-applications) by Thoughtbot (2014)
- Test Driven Rails (http://karolgalanciak.com/blog/2014/01/04/test-driven-rails-part-1/) by Karol Galanciak (2014)
- RSpec Best Practices (http://www.jacopretorius.net/2013/11/rspec-best-practices.html) by Jaco Pretorius (2013)
- RSpec Practices (http://tutorials.jumpstartlab.com/topics/internal_testing/rspec_practices.html) by Jumpstart Lab
- Testing Tuesday series (http://blog.codeship.io/category/testing-tuesday) from Codeship
- Testing Tips (https://semaphoreapp.com/blog/tags/testing.html) from Semaphore

Screencasts and Online Tutorials

Sadly, some of these tutorials haven’t been revised for the current RSpec syntax:

- How I Test (http://railscasts.com/episodes/275-how-i-test) from Railscasts
- RSpec the Right Way (https://peepcode.com/products/rspec-i) from PeepCode
- Test-Driven Rails (https://learn.thoughtbot.com/workshops/18-test-driven-rails) from Thoughtbot
- Testing with RSpec (https://www.codeschool.com/courses/testing-with-rspec) from Code School
- Various Screencasts (https://www.destroyallsoftware.com/screencasts/catalog) from Destroy All Software

Questions

You can use the tag “rspec” when you ask questions on Stack Overflow (http://stackoverflow.com/questions/tagged/rspec).

You can also ask questions on the RSpec Google Group (https://groups.google.com/forum/#!forum/rspec).

Words of Encouragement

Testing often intimidates the newcomer. It is difficult to find good examples. Sometimes there is little consistency among examples, especially because RSpec syntax has evolved over time. Older examples are not a good guide to current practices. But once you familiarize yourself with the current RSpec syntax and practices described in this guide, you can start writing tests.

Testing is one of the few things in Rails that you can jump into without worrying about getting exactly right. Your tests won't break anything. Tests won't affect the performance of your application in production. You just need to write tests that succeed or fail appropriately and you can keep trying until you get it right. If you're not sure if you've written tests the way you should, realize that RSpec gives you a lot of different ways to write your tests and what you've got will be good enough. If your code is clumsy, don't worry, you'll get better with practice. What's most important is that you've begun writing tests and that indicates you are committed to Rails best practices.

Your tests are only "bad" if they don't cover your code adequately or if they give you a false sense of assurance. You will only discover this over time, as you find bugs you didn't anticipate (which is inevitable). It's better to just begin testing, even if you're not sure you're doing it right, than to not test at all.

I hope this guide will start you down the path to better testing.

Comments

Credits

Daniel Kehoe wrote the guide.

Did You Like This Guide?

Was the guide useful to you? Follow @rails_apps (http://twitter.com/rails_apps) on Twitter and tweet some praise. I'd love to know you were helped out by the article.

You can also find me on Facebook (https://www.facebook.com/daniel.kehoe.sf) or Google+ (https://plus.google.com/+DanielKehoe/).