Rails 4 IN ACTION

Ryan Bigg
Yehuda Katz
Steve Klabnik
Rebecca Skinner

SAMPLE CHAPTER

MANNING

Rails 4 in Action

by Ryan Bigg
Yehuda Katz
Steve Klabnik
and Rebecca Skinner

Chapter 3

brief contents

3 Developing a real Rails application

This chapter covers

- Using a Git client and GitHub for Version control
- Setting up and configuring a Rails application
- Writing our first application feature
- Creating a Project model
- Creating an interface for saving new projects

This chapter will get you started on building a Ruby on Rails application from scratch using the techniques covered in the previous chapter, plus a couple of new ones. With the techniques you learned in chapter 2, you can write features describing the behavior of the specific actions in your application and then implement the code you need to get the features passing.

For the remainder of the book, this application will be the main focus. We'll guide you through it in an agile-like fashion. Agile focuses largely on iterative development: developing one feature at a time from start to finish, and then refining the feature until it's viewed as complete before moving on to the next one.

AGILE You can find more information about agile on Wikipedia: http://en.wikipedia.org/wiki/Agile_software_development.

For this example application, your imaginary client, who has limitless time and budget (unlike clients in the real world), wants you to develop a ticket-tracking application to track the company's numerous projects. You'll develop this application using the methodologies outlined in chapter 2: you'll work iteratively, delivering small working pieces of the software to the client and then gathering the client's feedback to improve the application as necessary. If no improvement is needed, you can move on to the next prioritized chunk of work.

The first couple of features you'll develop for this application will lay down the foundation for the application, enabling people to create projects and tickets. Later, in chapters 6 through 8, you'll implement authentication and authorization so that people can sign in to the application and only have access to certain projects. Other chapters cover things like adding comments to tickets, notifying users by email, and file uploading.

BDD and RSpec (the testing framework you saw in the previous chapter) are used all the way through the development process. They provide the client with a stable application, and when (not if) a bug crops up, you have a nice test base you can use to determine what's broken. Then you can fix the bug so it doesn't happen again, a process called *regression testing* (mentioned in chapter 2).

As you work with your client to build the features of the application using this BDD technique, the client may ask why all this prework is necessary. This can be a tricky question to answer. Explain that writing the tests before the code and then implementing the code to make the tests pass creates a safety net to ensure that the code is always working. (Note that tests will make your code more maintainable, but they won't make your code bug-proof.)

The tests also give you a clearer picture of what your client *really* wants. Having it all written down in code gives you a solid reference of point if clients say they suggested something different. *Story-driven development* is BDD with an emphasis on things a user can do with the system.

By using story-driven development, you know what clients want, clients know you know what they want, you have something you can run automated tests with to ensure that all the pieces are working, and, finally, if something *does* break, you have the test suite in place to catch it. It's a win-win-win situation.

Some of the concepts covered in this chapter were explained in chapter 1. But rather than using scaffolding, as you did previously, you'll write this application from the ground up using the BDD process and other generators provided by Rails. The `scaffold` generator is great for prototyping, but it's less than ideal for delivering simple, well-tested code that works precisely the way you want it to work. The code provided by the scaffold generator often may differ from the code you want. In this case, you can turn to Rails for lightweight alternatives to the scaffold code options, and you'll likely end up with cleaner, better code.

First, you need to set up your application!

3.1 First steps

Chapter 1 explained how to quickly start a Rails application. This chapter explains a couple of additional processes that improve the flow of your application development. One process uses BDD to create the features of the application; the other process uses version control. Both will make your life easier.

3.1.1 The application story

Your client may have a good idea of the application they want you to develop. How can you transform the idea that's in your client's brain into beautifully formed code? First, you sit down with your client and talk through the parts of the application. In the programming business, we call these parts *user stories*, and you'll use RSpec and Capybara to develop them.

> **WHAT IS CAPYBARA?** Capybara is a testing tool that allows you to simulate the steps of a user of your application. You can tell it to visit pages, fill in fields, click buttons (and links), and assert that pages have certain content. And there's a lot more that it can do, which you'll see throughout this book. It's used quite extensively throughout.

Start with the most basic story, and ask your client how they want it to behave. Then sketch out a basic flow of how the feature would work by building an acceptance test using RSpec and Capybara. If this feature was a login form, the test for it would look something like this:

```
RSpec.feature "Users can log in to the site" do
  scenario "as a user with a valid account" do
    visit "/login"
    fill_in "Email", with: "user@ticketee.com"
    fill_in "Password", with: "password"
    click_button "Login"
    expect(page).to have_content("You have been successfully logged in.")
  end
end
```

The form of this test is simple enough that even people who don't understand Ruby should be able to understand the flow of it. With the function and form laid out, you have a pretty good idea of what the client wants.

3.1.2 Laying the foundations

To start building the application you'll develop throughout this book, run the good old `rails` command, preferably outside the directory of the previous application. Call this app *Ticketee*, the Australian slang for a person who validates tickets on trains in an attempt to catch fare evaders. It also has to do with this project being a ticket-tracking application, and a Rails application, at that.[1] To generate this application, run this command:

```
$ rails new ticketee
```

[1] Hey, at least *we* thought it was funny!

Help!

If you want to see what else you can do with this `new` command (hint: there's a lot!), you can use the `--help` option:

```
$ rails new --help
```

The `--help` option shows you the options you can pass to the `new` command to modify the output of your application.

Presto! It's done. From this bare-bones application, you'll build an application that does the following:

- Tracks tickets (of course) and groups them into projects
- Provides a way to restrict users to certain projects
- Allows users to upload files to tickets
- Lets users tag tickets so they're easy to find
- Provides an API on which users can base development of their own applications

You can't do all this with a command as simple as `rails new [application_name]`, but you can do it step by step and test it along the way so you develop a stable and worthwhile application.

Throughout the development of the application, we advise you to use a version-control system. The next section covers that topic using Git. You're welcome to use a different version-control system, but this book uses Git exclusively.

3.2 Version control

It's wise during development to use version-control software to provide checkpoints in your code. When the code is working, you can make a commit; and if anything goes wrong later in development, you can revert back to that known-working commit. Additionally, you can create branches for experimental features and work on those independent of the main codebase, without damaging working code.

This book doesn't go into detail on how to use a version-control system, but it does recommend using Git. Git is a distributed version-control system that's easy to use and extremely powerful. If you wish to learn about Git, we recommend reading *Pro Git*, a free online book by Scott Chacon and Ben Straub (Apress, 2014, http://git-scm.com/book/en/v2).

Git is used by most developers in the Rails community and by tools such as Bundler, discussed shortly. Learning Git along with Rails is advantageous when you come across a gem or plug-in that you have to install using Git. Because most of the Rails community uses Git, you can find a lot of information about how to use it with Rails (even in this book!), should you ever get stuck.

If you don't have Git already installed, GitHub's help site offers installation guides for Mac, Linux, and Windows at https://help.github.com/articles/set-up-git.

The precompiled installer should work well for Macs, and the package-distributed versions (via apt, yum, emerge, and so on) work well for Linux machines. For Windows, the GitHub for Windows program does just fine.

3.2.1 Getting started with GitHub

For an online place to put your Git repository, we recommend GitHub (http://github.com), which offers free accounts. If you set up an account now, you can upload your code to GitHub as you progress, ensuring that you won't lose it if anything happens to your computer.

BITBUCKET Bitbucket (http://bitbucket.org) is a popular alternative to GitHub, and it also allows you to have free private repositories.

To get started with GitHub, you first need to generate a secure shell (SSH) key, which is used to authenticate you with GitHub when you do a `git push` to GitHub's servers.[2] After you sign up at GitHub, click the Settings link (see figure 3.1) in the menu at the top, select SSH Keys, and then click Add SSH Key (see figure 3.2). You can then copy your public key's content (usually found at ~/.ssh/id_rsa.pub) into the key field.

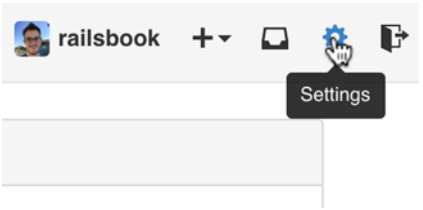

Figure 3.1 **Visiting account settings**

Now that you're set up with GitHub, click New Repository on the dashboard (see figure 3.3) to begin creating a new repository. Enter the Project Name as `Ticketee`, and click Create Repository to create the repository on GitHub.

Now you're on your project's page. It has some basic instructions on how to set up your code in your new repository, but first you need to configure Git on your own

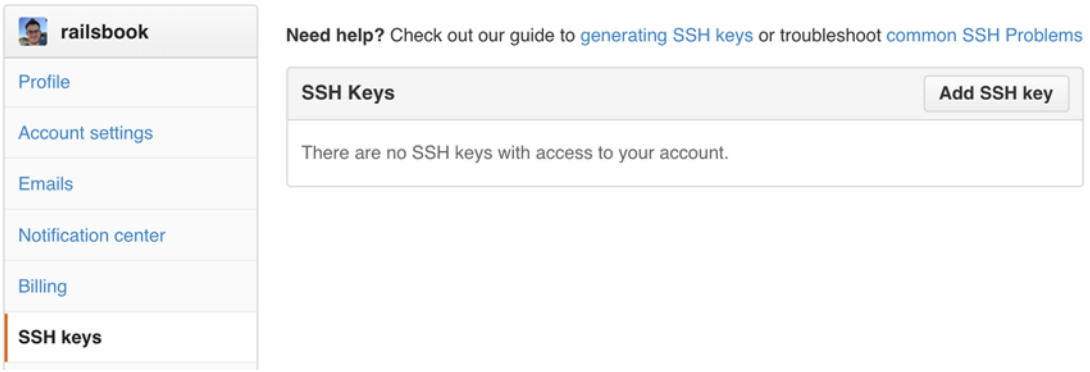

Figure 3.2 **Adding an SSH key**

[2] You can find a guide for this process at https://help.github.com/articles/generating-ssh-keys.

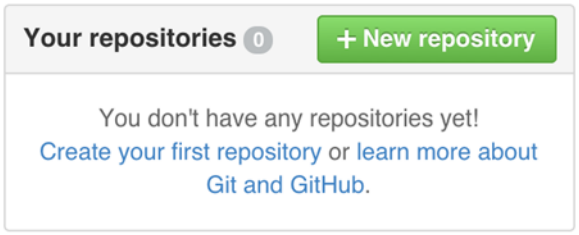

Figure 3.3 Creating a new repository

machine. Git needs to know a bit about you for identification purposes—so you can properly be credited (or blamed) for any code that you write.

3.2.2 Configuring your Git client

Run the commands from listing 3.1 in your terminal or command prompt to tell Git about yourself, replacing `Your Name` with your real name and `you@example.com` with your email address. The email address you provide should be the same as the one you used to sign up to GitHub, so that when you push your code to GitHub, it will also be linked to your account.

Listing 3.1 Configuring your identity in Git

```
$ git config --global user.name "Your Name"
$ git config --global user.email you@example.com
```

You already have a ticketee directory, created when you generated your Rails app, and you're probably already in it. If not, you should be. To make this directory a Git repository, run this easy command:

```
$ git init
```

Your ticketee directory now contains a .git directory, which is your Git repository. It's all kept in one neat little package.

To add all the files for your application to this repository's *staging area*, run this command:

```
$ git add .
```

The staging area for the repository is the location where all the changes for the next commit are kept. A commit can be considered a checkpoint for your code. If you make a change, you must *stage* that change before you can create a commit for it.

To create a commit with a message, run the following command:

```
$ git commit -m "Generate the Rails 4 application"
```

This command generates quite a bit of output, but the most important lines are the first two:

```
[master (root-commit) d825bbc] Generate the Rails 4 application
 57 files changed, 984 insertions(+)
```

`d825bbc` is the *short commit ID*, a unique identifier for the commit, so it changes with each commit you make. (The number of files and insertions may also be different.) In Git, commits are tracked against *branches*, and the default branch for a Git repository is the *master* branch, which you just committed to.

The second line lists the number of files changed, insertions (new lines added), and deletions. If you modify a line, it's counted as both an insertion and a deletion, because, according to Git, you've removed the line and replaced it with the modified version.

To view a list of commits for the current branch, type `git log`. You should see output similar to the following listing.

Listing 3.2 Viewing the commit log

```
commit d825bbc23854cc256d5829a06516ceb19d148131
Author: Your Name <you@example.com>
Date:   [date stamp]

    Generate the Rails 4 application
```

The hash after the word `commit` is the *long commit ID*; it's the longer version of the previous short commit ID. A commit can be referenced by either the long or the short commit ID in Git, providing no two commits begin with the same short ID.[3] With that commit in your repository, you have something to push to GitHub, which you can do by running the following commands, making sure you substitute your GitHub username for `[your username]`:

```
$ git remote add origin git@github.com:[your username]/ticketee.git
$ git push origin master -u
```

The first command tells Git that you have a remote server called *origin* for this repository. To access it, you use the git@github.com:[your username]/ticketee.git path, which connects via SSH to the repository you created on GitHub. The second command pushes the named branch to that remote server, and the `-u` option tells Git to always pull from this remote server for this branch unless told differently.

The output from this command is similar to the following.

Listing 3.3 `git push` output

```
Counting objects: 73, done.
Compressing objects: 100% (58/58), done.
Writing objects: 100% (73/73), 86.50 KiB, done.
Total 73 (delta 2), reused 0 (delta 0)
To git@github.com:rubysherpas/r4ia_examples.git
* [new branch]  master -> master
Branch master set up to track remote branch master from origin.
```

[3] The chances of this happening are 1 in 268,435,456.

The second-to-last line in this output indicates that your push to GitHub succeeded, because it shows that a new branch called `master` was created on GitHub.

As we go through the book, we'll also `git push` just like you. You can compare your code to ours by checking out our repository on GitHub: https://github.com/rubysherpas/r4ia_examples.

To roll back the code to a given point in time, check out `git log`:

```
commit d1e9b6f398748d3ca8583727c1f86496465ba298
Author: Rebecca Skinner <[email redacted]>
Date:   Sat Apr 4 00:00:33 2015 +0800

    Protect state_id from users who do not have permission
      to change it

commit ceb67d45cfcddbb8439da7b126802e6a48b1b9ea
Author: Rebecca Skinner <[email redacted]>
Date:   Fri Apr 3 23:27:20 2015 +0800

    Only admins and managers can change states of a ticket

commit ef5ec0f15e7add662852d6634de50648373f6116
Author: Rebecca Skinner <[email redacted]>
Date:   Fri Apr 3 23:01:48 2015 +0800

    Auto-assign the default state to newly-created tickets
```

Each of these lines represents a commit, and the commits will line up with when we tell you to commit in the book. You can also check out the commit list on GitHub, if you find that easier: https://github.com/rubysherpas/r4ia_examples/commits.

Once you've found the commit with the right message, make note of the long commit ID associated with it. Use this value with `git checkout` to roll the code back in time:

```
$ git checkout 23729a
```

You only need to know enough of the hash for it to be unique: six characters is usually enough. When you're done poking around, go forward in time to the most recent commit with `git checkout` again:

```
$ git checkout master
```

This is a tiny, tiny taste of the power of Git. Time travel at will! You just have to learn the commands.

Next, you must set up your application to use RSpec.

3.3 Application configuration

Even though Rails passionately promotes the *convention over configuration* line, some parts of the application will need configuration. It's impossible to avoid *all* configuration. The main parts are gem dependency configuration, database settings, and styling. Let's look at these parts now.

3.3.1 The Gemfile and generators

The Gemfile is used for tracking which gems are used in your application. *Gem* is the Ruby word for a library of code, all packaged up to be included into your app—Rails is a gem, and it in turn depends on many other gems.

Bundler is a gem, and Bundler is also responsible for everything to do with the Gemfile. It's Bundler's job to ensure that all the gems listed inside the Gemfile are installed when your application is initialized. The following listing shows how it looks inside.

Listing 3.4 Default Gemfile in a new Rails app

```
source 'https://rubygems.org'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.2.1'
# Use sqlite3 as the database for Active Record
gem 'sqlite3'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 5.0'
# Use Uglifier as compressor for JavaScript assets
gem 'uglifier', '>= 1.3.0'
# Use CoffeeScript for .coffee assets and views
gem 'coffee-rails', '~> 4.1.0'
# See https://github.com/sstephenson/execjs#readme for more supported...
# gem 'therubyracer', platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'
# Turbolinks makes following links in your web application faster...
gem 'turbolinks'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbu...
gem 'jbuilder', '~> 2.0'
# bundle exec rake doc:rails generates the API under doc/api.
gem 'sdoc', '~> 0.4.0', group: :doc

# Use ActiveModel has_secure_password
# gem 'bcrypt', '~> 3.1.7'

# Use Unicorn as the app server
# gem 'unicorn'

# Use Capistrano for deployment
# gem 'capistrano-rails', group: :development

group :development, :test do
  # Call 'byebug' anywhere in the code to stop execution and get a...
  gem 'byebug'

  # Access an IRB console on exception pages or by using <%= console...
  gem 'web-console', '~> 2.0'

  # Spring speeds up development by keeping your application running in the
```

```
background. Read more: https://github.com/rails/spring
  gem 'spring'
end
```

In this file, Rails sets a source to be https://rubygems.org (the canonical repository for Ruby gems). All gems you specify for your application are gathered from the source. Next, it tells Bundler it requires version 4.2.1 of the rails gem. Bundler inspects the dependencies of the requested gem, as well as all gem dependencies of those dependencies (and so on), and then does what it needs to do to make them all available to your application.

This file also requires the sqlite3 gem, which is used for interacting with SQLite3 databases, the default when working with Rails. If you were to use another database system, you'd need to take out this line and replace it with the relevant gem, such as mysql2 for MySQL or pg for PostgreSQL.

Groups in the Gemfile are used to define gems that should be loaded in specific scenarios. When using Bundler with Rails, you can specify a gem group for each Rails *environment*, and by doing so, you specify which gems should be required by that environment. A default Rails application has three standard environments: development, test, and production.

Rails application environments

The *development* environment is used for your local application, such as when you're playing with it in the browser on your local machine. In development mode, page and class caching are turned off, so requests may take a little longer than they do in production mode. (Don't worry—this is only the case for larger applications.) Things like more detailed error messages are also turned on, for easier debugging.

The *test* environment is used when you run the automated test suite for the application. This environment is kept separate from the development environment so your tests start with a clean database to ensure predictability, and so you can include extra gems specifically to aid in testing.

The *production* environment is used when you finally deploy your application out into the world for others to use. This mode is designed for speed, and any changes you make to your application's classes aren't effective until the server is restarted.

This automatic requiring of gems in the Rails environment groups is done by the following line in config/application.rb:

```
Bundler.require(*Rails.groups)
```

The `Rails.groups` line provides two groups for Bundler to require: `default` and `development`. The latter will change depending on the environment that you're running. This code will tell Bundler to load only the gems in the `default` group (which is

all gems not in any specific group), as well as any gems in a group that has the same name as the environment.

GETTING STARTED WITH BDD

Chapter 2 focused on behavior-driven development (BDD), and, as was more than hinted at, you'll use it to develop this application. To get started, you need to alter the Gemfile to ensure that you have the correct gem for RSpec for your application.

To add the rspec-rails gem, add this line to the bottom of the `:development, :test` group in your Gemfile:

```
gem "rspec-rails", "~> 3.2.1"
```

This group in your Gemfile lists all the gems that will be loaded in the development and test environments of your application. These gems won't be available in a production environment. You add rspec-rails to this group because you're going to need a generator from it to be available in development. Additionally, when you run a generator for a controller or model, it'll use RSpec, rather than the default Test::Unit, to generate the tests for that class.

You've specified a version number with `~> 3.2.1`,[4] which tells RubyGems you want rspec-rails 3.2.1 *or higher*, but less than rspec-rails 3.3. This means when RSpec releases 3.2.2 and you go to install your gems, RubyGems will install the latest version it can find, rather than only 3.2.1.

Next, you'll need to add Capybara to the Gemfile, in a new group specifically for the test environment. You don't put Capybara in the same group as the rspec-rails gem because it doesn't offer any generators that you need, so you only need this gem loaded in the test environment.

```
group :test do
  gem "capybara", "~> 2.4"
end
```

Capybara is a browser simulator in Ruby that's used for *integration testing*, which you'll be doing shortly. This kind of testing ensures that when a link is clicked in your application, it goes to the correct page; or that when you fill in a form and click Submit, an onscreen message tells you that the form's operation was successful.

Capybara also supports real browser testing. If you tell RSpec that your test is a *JavaScript* test, it will open a new Firefox window and run the test there—you'll be able to see your tests as they occur, and your application will behave exactly the same as it does when you view it yourself. You'll use this extensively when we start writing JavaScript in chapter 9.

To install these gems on your system, run `bundle update` at the root of your application. This command tells Bundler to ignore your Gemfile.lock file and use your Gemfile to install all the gems specified in it. Bundler then updates Gemfile.lock with the list of gems that were installed, as well as their versions. The next time `bundle` is

[4] The ~> operator is called the *approximate version constraint*.

run, the gems will be read from the Gemfile.lock file, rather than the Gemfile. You should commit this file to your repository so that when other people work on your project and run `bundle install`, they'll get exactly the same versions that you have.

With the necessary gems for the application installed, you can run the `rspec:install` generator, a generator provided by RSpec to set your Rails application up for testing:

```
$ rails g rspec:install
```

REMINDER Remember, `rails g` is a shortcut for running `rails generate`!

You can also remove the automatically generated `test` application in the root folder of your application—you won't be using it.

With this generated code in place, you should make a commit so you have another base to roll back to if anything goes wrong:

```
$ git add .
$ git commit -m "Set up gem dependencies and run RSpec generator"
$ git push
```

3.3.2 Database configuration

By default, Rails uses a database system called SQLite3, which stores each environment's database in separate files in the db directory. SQLite3 is the default database system because it's the easiest to set up. Out of the box, Rails also supports the MySQL and PostgreSQL databases, and gems are available that can provide functionality for connecting to other database systems such as Oracle.

If you want to change which database your application connects to, you can open config/database.yml (whose development configuration is shown in the following listing) and alter the settings to the new database system.

Listing 3.5 config/database.yml, SQLite3 example

```
development:
  adapter: sqlite3
  database: db/development.sqlite3
  pool: 5
  timeout: 5000
```

For example, if you want to use PostgreSQL, change the settings to match those in the following listing. It's common convention, but not mandatory, to call the environment's database `[app_name]_[environment]`.

Listing 3.6 config/database.yml, PostgreSQL example

```
development:
  adapter: postgresql
  database: ticketee_development
  username: root
  password: t0ps3cr3t
```

You're welcome to change the database if you wish. Rails will go about its business. But it's good practice to develop and deploy on the same database system to avoid strange behavior between two different systems. Systems such as PostgreSQL perform faster than SQLite, so switching to it may increase your application's performance. Be mindful, however, that switching database systems doesn't automatically move your existing data over for you.

It's generally wise to use different names for the different database environments: if you use the same database in development and test modes, the database would be emptied of all data when the tests were run, eliminating anything you might have set up in development mode. You should never work on the live production database directly unless you're absolutely sure of what you're doing, and even then extreme care should be taken.

Finally, if you're using MySQL, it's wise to set the encoding to `utf-8` for the database, using the following setup in the config/database.yml file.

Listing 3.7 config/database.yml, MySQL example

```
development:
  adapter: mysql2
  database: ticketee_development
  username: root
  password: t0ps3cr3t
  encoding: utf8
```

This way, the database is set up automatically to work with UTF-8, eliminating any potential encoding issues that may be encountered otherwise.

That's database configuration in a nutshell. For this book and for the Ticketee application, we'll use the default of SQLite3, but it's good to know about the alternatives and how to configure them.

3.4 Beginning your first feature

You now have version control for your application, and you're hosting it on GitHub. It's time to write your first Capybara-based test, which isn't nearly as daunting as it sounds. We'll explore things such as models and RESTful routing while you do it. It'll be simple, promise!

The CRUD (create, read, update, delete) acronym is something you'll see all the time in the Rails world. It represents the creation, reading, updating, and deleting of something, but it doesn't say what that something is.

In the Rails world, CRUD is usually referred to when talking about *resources*. Resources are the representation of the information throughout your application—the things that your application is designed to manage. The following section goes through the beginnings of generating a CRUD interface for a *project* resource by applying the BDD practices you learned in chapter 2 to the application you just bootstrapped. What comes next is a sampler of how to apply these practices when developing a Rails application.

Throughout the remainder of the book, you'll continue to apply these practices to ensure that you have a stable and maintainable application.

Let's get into it!

3.4.1 Creating projects

The first story for your application is the creation (the *C* in CRUD). You'll create a resource representing projects in your application by first writing a test for the process by which a user will create projects, then creating a controller and model, and then creating a route. Then you'll add a validation to ensure that no project can be created without a name. When you're done with this feature, you'll have a form that looks like figure 3.4.

Figure 3.4 Form to create projects

First, create a new directory at spec/features—all of the specs covering your features will go there. Then, in a file called spec/features/creating_projects_spec.rb, you'll put the test that will make sure this feature works correctly when it's fully implemented. The test code is shown in the following listing.

Listing 3.8 spec/features/creating_projects_spec.rb

```
require "rails_helper"

RSpec.feature "Users can create new projects" do
  scenario "with valid attributes" do
    visit "/"

    click_link "New Project"

    fill_in "Name", with: "Sublime Text 3"
    fill_in "Description", with: "A text editor for everyone"
    click_button "Create Project"

    expect(page).to have_content "Project has been created."
  end
end
```

To run this test, run `bundle exec rspec` from inside your ticketee folder. This command will run all of your specs and display the first failure of your application's first test:

```
1) Users can create new projects with valid attributes
   Failure/Error: visit "/"
   ActionController::RoutingError:
     No route matches [GET] "/"
```

THE SCHEMA.RB FILE Before the test failure, you'll also get a warning that your schema.rb file doesn't exist yet. This schema.rb file represents Rails' knowledge about the application's database. You haven't created a database yet, so it's safe to ignore this warning for now. It will get resolved when you create your first database table shortly.

It falls on the application's *router* to figure out where the request should go. Typically, the request would be routed to an action in a controller, but at the moment there are no routes at all for the application. With no routes, the Rails router can't find the route for `"/"` and so gives you the error shown.

You have to tell Rails what to do with a request for `"/"`. You can do this easily in config/routes.rb. At the moment, this file has the following content (comments removed).

Listing 3.9 config/routes.rb

```
Rails.application.routes.draw do
end
```

The comments are good for a read if you're interested in the other routing syntax, but they're not necessary at the moment. (We've removed them from the code displayed here, but you can keep them if you like.) To define a root route, you use the `root` method like this in the block for the `draw` method:

```
Rails.application.routes.draw do
  root "projects#index"
end
```

This defines a route for requests to `"/"` (the root route) to point at the `index` action of the `ProjectsController`. This controller doesn't exist yet, and so the test should probably complain about that if you got the route right. Run `bundle exec rspec` to find out:

```
1) Users can create new projects with valid attributes
   Failure/Error: visit "/"
   ActionController::RoutingError:
     uninitialized constant ProjectsController
```

This error is happening because the route is pointing at a controller that doesn't exist. When the request is made, the router attempts to load the controller, and because it can't find it, you'll get this error.

To define this `ProjectsController` constant, you must generate a *controller*. The controller is the first port of call for your routes (as you can see now!), and it's responsible for querying the model for information in an action and then doing something with that information (such as rendering a template). (Lots of new terms are explained later. Patience, grasshopper.) To generate the controller, run this command:

```
$ rails g controller projects
```

You may be wondering why we use a pluralized name for the controller. Well, the controller is going to be dealing with a plural number of projects during its lifetime, so it makes sense to name it like this. The models are singular because their names refer to their types. Another way to put it: you're a human, not a humans. But a controller that dealt with multiple humans would be called HumansController.

The controller generator produces output similar to that produced when you ran `rails new` earlier, but this time it creates files just for the controller you've asked Rails

to generate. The most important of these is the controller itself, which is housed in app/controllers/projects_controller.rb and defines the `ProjectsController` constant that your test needs. This controller is where all the actions will live, just like app/controllers/purchases_controller.rb back in chapter 1. Here's what this command outputs:

```
create  app/controllers/projects_controller.rb
invoke  erb
create    app/views/projects
invoke  rspec
create    spec/controllers/projects_controller_spec.rb
invoke  helper
create    app/helpers/projects_helper.rb
invoke    rspec
create      spec/helpers/projects_helper_spec.rb
invoke  assets
invoke    coffee
create      app/assets/javascripts/projects.coffee
invoke    scss
create      app/assets/stylesheets/projects.scss
```

A few notes about the output:

- app/views/projects contains the views relating to your actions (more on this shortly).
- `invoke helper` shows that the `helper` generator was called here, generating a file at app/helpers/projects_helper.rb. This file defines a `ProjectsHelper` module. Helpers generally contain custom methods to be used in your view to help with the rendering of content, and they come as blank slates when they're first created.
- `invoke erb` signifies that the Embedded Ruby (ERB) generator was invoked. Actions to be generated for this controller have corresponding ERB views located in app/views/projects. For instance, the `index` action's default view is located at app/views/projects/index.html.erb.
- `invoke rspec` shows that the RSpec generator was also invoked during the generation. This means RSpec has generated a new file at spec/controllers/projects_controller_spec.rb, which you can use to test your controller—but not right now.[5]
- Finally, the assets for the controller are generated. Two files are generated here: app/assets/javascripts/projects.coffee and app/assets/stylesheets/projects.scss. The first file should contain any JavaScript related to the controller, written as CoffeeScript (http://coffeescript.org). The second file should contain any CSS related to the controller, written using SCSS (http://sass-lang.com). In the development environment, these files are automatically parsed into JavaScript and CSS, respectively.

[5] By generating RSpec tests rather than Test::Unit tests, a longstanding issue in Rails has been fixed. In previous versions of Rails, even if you specified the RSpec gem, all the default generators still generated Test::Unit tests. With Rails, the testing framework you use is just one of a large number of configurable things in your application.

You've just run the generator to generate a new `ProjectsController` class and all its goodies. This should fix the "uninitialized constant" error message. If you run `bundle exec rspec` again, it declares that the `index` action is missing:

```
1) Users can create new projects with valid attributes
   Failure/Error: visit "/"
   AbstractController::ActionNotFound:
     The action 'index' could not be found for ProjectsController
```

3.4.2 Defining a controller action

To define the `index` action in your controller, you must define a method in the `ProjectsController` class, just as you did when you generated your first application.

Listing 3.10 app/controllers/projects_controller.rb

```
class ProjectsController < ApplicationController
  def index
  end
end
```

If you run `bundle exec rspec` again, this time Rails complains of a missing projects/index template:

```
1) Users can create new projects with valid attributes
   Failure/Error: visit "/"
   ActionView::MissingTemplate:
     Missing template projects/index, application/index with
     {
      :locale => [:en],
      :formats => [:html],
      :variants => [],
      :handlers => [:erb, :builder, :raw, :ruby, :coffee, :jbuilder]
     }.

     Searched in:
     * ".../ticketee/app/views"
```

We've reformatted the error message to make it a little easier on the untrained eye. It doesn't look very helpful, but if you know how to put the pieces together, you can determine that it's trying to look for a template called projects/index or application/index, but it's not finding it. These templates are primarily kept at app/views, so it's fair to guess that it's expecting something like app/views/projects/index.

The extension of the file will be composed of two parts: the format followed by the handler. The error output lists `:html` as an available format, which is good, because we want to render an HTML page. But what's a handler?

A *handler* is a preprocessor for the template, or a templating language. There are a number of handlers built in (as the error output lists), and many more can be added by using extra gems, but the default for Rails views is `:erb`. Putting it all together, the view that your `index` action will render belongs in app/views/projects/index.html.erb.

Rails view `variants`

We haven't discussed the `variants` option in the output at all. View variants are a relatively new feature within Rails, and they allow you to provide different views based on certain criteria, such as what device a user is accessing the site through.

The Rails 4.1 release notes have the best information about variants, so if you're interested in what they can do, check them out here: http://guides.rubyonrails.org/4_1_release_notes.html#action-pack-variants.

You could also create a file at app/views/application/index.html.erb to provide the view for the `index` action from the `ProjectsController`. This would work because `ProjectsController` inherits from `ApplicationController`. If you had another controller inherit from `ProjectsController`, you could put an action's template at app/views/application, app/views/projects, or app/views/that_controller, and Rails would still pick up on it. This allows different controllers to share views in a simple fashion. Creating a view at app/views/application/index.html.erb would make this view available for all controllers that inherit from `ApplicationController`, but that's not what we want to do in this case.

To generate this view, create the app/views/projects/index.html.erb file and leave it blank for now. You can run just the single spec for creating projects with `bundle exec rspec`:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_link "New Project"
   Capybara::ElementNotFound:
     Unable to find link "New Project"
```

You've defined a homepage for your application by defining a root route, generating a controller, putting an action in it, and creating a view for that action. Now Capybara is successfully navigating to it and rendering it. That's the first step in the first test passing for your first application, and it's a great first step!

The second line in your spec is now failing, and it's up to you to fix it. You need a link on the root page of your application that reads "New Project." That link should go in the view of the controller that's serving the root route request: app/views/projects/index.html.erb. Open app/views/projects/index.html.erb and put the link in by using the `link_to` method:

```
<%= link_to "New Project", new_project_path %>
```

This single line re-introduces two old concepts and a new one: ERB output tags, the `link_to` method (both of which you saw in chapter 1), and the mysterious `new_project_path` method.

As a refresher, in ERB, when you use `<%=` (known as an ERB output tag), you're telling ERB that whatever the output of this Ruby is, put it on the page. If you only want to

evaluate (and not output) Ruby, you use an ERB evaluation tag `<%`, which doesn't output content to the page but only evaluates it. Both of these tags end in `%>`.

The `link_to` method in Rails generates an `<a>` tag with the text of the first argument and the `href` of the second argument. This method can also be used in block format if you have a lot of text you want to link to:

```
<%= link_to new_project_path do %>
  bunch
  of
  text
<% end %>
```

Where `new_project_path` comes from deserves its own section. It's the very next one.

3.4.3 RESTful routing

The `new_project_path` method is as yet undefined. If you ran the test again, it would complain of an "undefined local variable or method, 'new_project_path'." You can define this method by defining a route to what's known as a *resource* in Rails. Resources are collections of objects that all belong in a common location, such as projects, users, or tickets.

You can add the projects resource in config/routes.rb by using the `resources` method, putting it directly under the `root` method in this file.

Listing 3.11 `resources :projects` line in config/routes.rb

```
Rails.application.routes.draw do
  root "projects#index"

  resources :projects
end
```

This is called a *resource* route, and it defines the routes to the seven *RESTful* actions in your projects controller. When something is said to be RESTful, it means it conforms to Rails' interpretation of the Representational State Transfer (REST) architectural style. (See Wikipedia for more information on REST: http://en.wikipedia.org/wiki/Representational_state_transfer.)

Rails can't get you all the way there, but it can help. With Rails, this means the related controller has seven potential actions:

- index
- show
- new
- create
- edit
- update
- destroy

These seven actions match up with just four request paths:

- /projects
- /projects/new
- /projects/:id
- /projects/:id/edit

How can four be equal to seven? It can't! Not in this world, anyway. Rails will determine what action to route to on the basis of the HTTP method of the requests to these paths. Table 3.1 lists the routes, HTTP methods, and corresponding actions to make it clearer.

Table 3.1 RESTful routing match-up

HTTP method	Route	Action
GET	/projects	index
POST	/projects	create
GET	/projects/new	new
GET	/projects/:id	show
PATCH/PUT	/projects/:id	update
DELETE	/projects/:id	destroy
GET	/projects/:id/edit	edit

The routes listed in the table are provided when you use `resources :projects`. This is yet another great example of how Rails takes care of the configuration so you can take care of the coding.

To review the routes you've defined, you can run the `bundle exec rake routes` command and get output similar to that in table 3.1:

```
Prefix Verb   URI Pattern                  Controller#Action
        root GET    /                           projects#index
    projects GET    /projects(.:format)         projects#index
             POST   /projects(.:format)         projects#create
 new_project GET    /projects/new(.:format)     projects#new
edit_project GET    /projects/:id/edit(.:format) projects#edit
     project GET    /projects/:id(.:format)     projects#show
             PATCH  /projects/:id(.:format)     projects#update
             PUT    /projects/:id(.:format)     projects#update
             DELETE /projects/:id(.:format)     projects#destroy
```

The words in the leftmost column of this output are the beginnings of the method names you can use in your controllers or views to access them. If you want just the path to a route, such as /projects, then use `projects_path`. If you want the full URL, such as http://yoursite.com/projects, use `projects_url`. It's best to use these helpers rather than hardcoding the URLs; doing so makes your application consistent across the board.

For example, to generate the route to a single project, you'd use either `project_path` or `project_url`:

```
project_path(@project)
```

This method takes one argument, shown in the URI pattern with the :id notation, and generates the path according to this object. The four paths mentioned earlier match up to the helpers in table 3.2.

Table 3.2 RESTful routing match-up for GET routes

URL	Helper
GET /projects	projects_path
GET /projects/new	new_project_path
GET /projects/:id	project_path
GET /projects/:id/edit	edit_project_path

Running `bundle exec rspec` now produces a complaint about a missing `new` action:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_link "New Project"
   AbstractController::ActionNotFound:
     The action 'new' could not be found for ProjectsController
```

As shown in the following listing, you define the `new` action in your controller by defining a `new` method directly underneath the `index` method.

Listing 3.12 app/controllers/projects_controller.rb

```
class ProjectsController < ApplicationController
  def index
  end

  def new
  end
end
```

Running `bundle exec rspec` now results in a complaint about a missing `new` template, just as it did with the `index` action:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_link "New Project"
   ActionView::MissingTemplate:
     Missing template projects/new, application/new with
     {
      :locale => [:en],
      :formats => [:html],
      :variants => [],
      :handlers => [:erb, :builder, :raw, :ruby, :coffee, :jbuilder]
     }.

     Searched in:
     * ".../ticketee/app/views"
```

You can create the file at app/views/projects/new.html.erb to make this test go one step further, although this is a temporary solution. You'll come back to this file later to add content to it.

When you run the spec again, the line that should be failing is the one regarding filling in the Name field. Find out if this is the case by running `bundle exec rspec`:

```
1) Users can create new projects with valid attributes
   Failure/Error: fill_in "Name", with: "Sublime Text 3"
   Capybara::ElementNotFound:
     Unable to find field "Name"
```

Now Capybara is complaining about a missing Name field on the page it's currently on: the `new` page. You must add this field so that Capybara can fill it in. Before you do that, however, fill out the `new` action in the `ProjectsController` as follows:

```
def new
  @project = Project.new
end
```

When you fill out the view with the fields you need to create a new project, you'll need something to base the fields on—an instance of the class you want to create. This `Project` constant will be a class located at app/models/project.rb, thereby making it a *model*.

OF MODELS AND MIGRATIONS

A model is used to retrieve information from the database. Because models by default inherit from Active Record, you don't have to set up anything extra. Run the following command to generate your first model:

```
$ rails g model project name description
```

This syntax is similar to the controller generator's syntax, except that you specify that you want a model, not a controller.

When the generator runs, it generates not only the model file but also a *migration* containing the code to create the table (containing the specified fields) for the model. You can specify as many fields as you like after the model's name. They default to `string` type, so you don't need to specify them. If you wanted to be explicit, you could use a colon followed by the field type, like this:

```
$ rails g model project name:string description:string
```

A model provides a place for any business logic that your application performs—one common bit of logic is the way your application interacts with a database. A model is also the place where you define *validations* (seen later in this chapter), *associations* (discussed in chapter 5), and *scopes* (easy-to-use filters for database calls, discussed in chapter 7), among other things. To perform any interaction with data in your database, you go through a model.[6]

[6] Although it's possible to perform database operations without a model in Rails, 99% of the time you'll want to use a model.

Migrations are effectively version control for the database. They're defined as Ruby classes, which allows them to apply to multiple database schemas without having to be altered. All migrations have a `change` method in them when they're first defined. For example, the code shown in the following listing comes from the migration that was just generated.

Listing 3.13 db/migrate/[date]_create_projects.rb

```
class CreateProjects < ActiveRecord::Migration
  def change
    create_table :projects do |t|
      t.string :name
      t.string :description

      t.timestamps null: false
    end
  end
end
```

When you run the migration forward (using `bundle exec rake db:migrate`), it creates the table in the database. When you roll the migration back (with `bundle exec rake db:rollback`), it deletes (or *drops*) the table from the database.

If you need to do something different on the `up` and `down` parts, you can use those methods instead.

Listing 3.14 Explicitly using `up` and `down` methods to define a migration

```
class CreateProjects < ActiveRecord::Migration
  def up
    create_table :projects do |t|
      t.string :name
      t.string :description

      t.timestamps null: false
    end
  end

  def down
    drop_table :projects
  end
end
```

Here, the `up` method would be called if you ran the migration forward, and the `down` method would be run if you ran it backward.

This syntax is especially helpful if the migration does something that has a reverse function that isn't clear, such as removing a column:[7]

[7] Rails actually does know how to reverse the removal of a column if you provide an extra field type argument to `remove_column`; for example, `remove_column :projects, :name, :string`. We'll leave this here for demonstration purposes, though.

```
class CreateProjects < ActiveRecord::Migration
  def up
    remove_column :projects, :name
  end

  def down
    add_column :projects, :name, :string
  end
end
```

In this case, Active Record wouldn't know what type of field to re-add this column as, so you must tell it what to do in the case of this migration being rolled back.

In the projects migration, the first line of the `change` method tells Active Record that you want to create a table called `projects`. You call this method in the block format, which returns an object that defines the table. To add fields to this table, you call methods on the block's object (called `t` in this example and in all model migrations), the name of which usually reflects the type of column it is; the first argument is the name of that field. The `timestamps` method is special: it creates two fields, `created_at` and `updated_at`, which are by default set to the current time in coordinated universal time (UTC)[8] by Rails when a record is created and updated, respectively.

A migration doesn't automatically run when you create it—you must run it yourself using this command:

```
$ bundle exec rake db:migrate
```

This command migrates the database up to the latest migration, which for now is your only migration. If you create a whole slew of migrations at once, then invoking `bundle exec rake db:migrate` will migrate them in the order in which they were created. This is the purpose of the timestamp in the migration filename—to keep the migrations in chronological order.

The end of `rake db:test:prepare`

In older versions of Rails (before 4.1), whenever you ran a migration in the development environment with `rake db:migrate`, you also had to manually keep your test database in sync with `rake db:test:prepare`. This led to much confusion. Your tests could raise errors about missing database fields, but they were there—you'd run migrations to create them but not run `rake db:test:prepare` to sync those changes to the test database.

Now the two databases are kept in sync automatically, with this line in your spec/rails_helper.rb file:

```
ActiveRecord::Migration.maintain_test_schema!
```

No more need to call `rake db:test:prepare`!

[8] Yes, coordinated universal time has an initialism of *UTC*. This is what happens when you name things by committee (http://en.wikipedia.org/wiki/Coordinated_Universal_Time#Etymology).

With this model created and its related migration run, your test won't get any further, but you can start building out the form to create a new project.

FORM BUILDING

To add the fields for creating a new project to the `new` action's view, you can put them in a form, but not just any form: a `form_for`.

Listing 3.15 app/views/projects/new.html.erb

```
<h1>New Project</h1>
<%= form_for(@project) do |f| %>
  <p>
    <%= f.label :name %><br>
    <%= f.text_field :name %>
  </p>

  <p>
    <%= f.label :description %><br>
    <%= f.text_field :description %>
  </p>

  <%= f.submit %>
<% end %>
```

So many new things!

Starting at the top, the `form_for` method is Rails' way of building forms for Active Record objects. You pass it the `@project` object you defined in your controller as the first argument, and with this, the helper does much more than simply place a form tag on the page. `form_for` inspects the `@project` object and creates a form builder specifically for that object. The two main things it inspects are whether it's a new record and what the class name is.

What `action` attribute the form has (the URL the form submits its data to) depends on whether or not the object is a new record. A record is classified as new when it hasn't been saved to the database. This check is performed internally to Rails using the `persisted?` method, which returns `true` if the record is stored in the database and `false` if it's not.

The class of the object also plays a pivotal role in where the form is sent—Rails inspects this class and, from it, determines what the route should be. Because `@project` is new and is an object of class `Project`, Rails determines that the submit URL is `/projects` and the method for the form is `POST`. Therefore, a request is sent to the `create` action in `ProjectsController`.

After that part of `form_for` is complete, you use the block syntax to receive an `f` variable, which is a `FormBuilder` object. You can use this object to define your form's fields. The first element you define is a `label`. `label` tags directly relate to the input fields on the page and serve two purposes. First, they give users a larger area to click, rather than just the field, radio button, or check box. Second, you can reference the label's text in the test, and Capybara will know what field to fill in.

Alternative label naming

By default, the label's text value will be the "humanized" value of the field name; for example, `:name` becomes "Name." If you want to customize the text, you can pass the `label` method a second argument:

```
<%= f.label :name, "Your name" %>
```

After the label, you add the `text_field`, which renders an `<input>` tag corresponding to the label and the field. The output tag looks like this:

```
<input type="text" name="project[name]" id="project_name" />
```

Then you use the `submit` method to provide users with a Submit button for your form. Because you call this method on the `f` object, Rails checks whether the record is new and sets the text to read "Create Project" if the record is new or "Update Project" if it isn't. You'll see this in use a little later when you build the `edit` action. For now, we'll focus on the `new` action!

Run `bundle exec rspec spec/features/creating_projects_spec.rb` once more, and you can see that your spec is one step closer to finishing—the field fill-in steps have passed:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_button "Create Project"
   AbstractController::ActionNotFound:
     The action 'create' could not be found for ProjectsController
```

Capybara finds the label containing the `Name` text you ask for in your scenario, and fills out the corresponding field with the value you specify. Capybara has a number of ways to locate a field, such as by the name of the corresponding label, the `id` attribute of the field, or the `name` attribute. The last two look like these:

```
fill_in "project_name", with: "Sublime Text 3"
# or
fill_in "project[name]", with: "Sublime Text 3"
```

SHOULD YOU USE THE ID OR THE LABEL? Some argue that using the field's ID or name is a better approach, because these attributes don't change as often as labels may. But your tests should aim to be as human-readable as possible—when you write them, you don't want to be thinking of field IDs; you're describing the behavior at a higher level than that. To keep things simple, you should continue using the label name.

Capybara does the same thing for the Description field and then clicks the button you told it to click. The spec is now complaining about a missing action called `create`. Let's fix that.

CREATING THE CREATE ACTION

To define this action, you define the `create` method underneath the `new` method in the `ProjectsController`.

Listing 3.16 The `create` action of `ProjectsController`

```
def create
  @project = Project.new(project_params)

  if @project.save
    flash[:notice] = "Project has been created."
    redirect_to @project
  else
    # nothing, yet
  end
end
```

The `Project.new` method takes one argument, which is a list of attributes that will be assigned to this new `Project` object. For now, we'll just call that list `project_params`.

After you build your new `@project` instance, you call `@project.save` to save it to the `projects` table in your database. Before that happens, though, Rails will run all the data validations on the model, ensuring that it's valid. At the moment, you have no validations on the model, so it will save just fine.

The `flash` method in your `create` action is a way of passing messages to the next request, and it takes the form of a hash. These messages are stored in the session and are cleared at the completion of the next request. Here you set the `:notice` key of the flash hash to be "Project has been created" to inform the user what has happened. This message is displayed later, as is required by the final step in your feature.

The `redirect_to` method can take several different arguments—an object, or the name of a route. If an object is given, Rails inspects it to determine what route it should go to; in this case, it goes to `project_path(@project)` because the object has now been saved to the database. This method generates a path in the form of /projects/:id, where :id is the record's `id` attribute assigned by your database system. The `redirect_to` method tells the browser to begin making a new request to that path and sends back an empty response body; the HTTP status code will be a "302 Redirect," and the URL to redirect to will match the URL of the currently nonexistent `show` action.

If you run `bundle exec rspec` now, you'll get an error about an undefined local variable or method `"project_params"`:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_button "Create Project"
   NameError:
     undefined local variable or method `project_params' for
     #<ProjectsController:0x007fe704e31848>
```

Where does the data you want to make a new project from, come from? It comes from the `params` provided to the controller, available to all Rails controller actions.

Combining `redirect_to` and `flash`

You can combine `flash` and `redirect_to` by passing the `flash` as an option to the `redirect_to`. If you want to pass a success message, use the `notice` flash key; otherwise use the `alert` key.

To use either of these two keys, you can use this syntax:

```
redirect_to @project, notice: "Project has been created."
# or
redirect_to @project, alert: "Project has not been created."
```

If you don't wish to use either `notice` or `alert`, you must specify `flash` as a hash:

```
redirect_to @project, flash: { success: "Project has been created."}
```

The `params` method returns the parameters passed to the action, such as those from the form or query parameters from a URL, as a `HashWithIndifferentAccess` object. This is different from a normal `Hash` object, because you can reference a `String` key by using a matching `Symbol`, and vice versa.

In this case, the `params` hash looks like this:

```
{
  "utf8" => "?",
  "authenticity_token" => "WRHnKqU...",
  "project" => {
    "name" => "Sublime Text 3",
    "description" => "A text editor for everyone"
  },
  "commit" => "Create Project",
  "controller" => "projects",
  "action" => "create"
}
```

You can easily see what parameters your controller is receiving by looking at the server logs in your terminal console. If you run your `rails server`, visit http://localhost:3000/projects/new, and submit the data that your test is trying to submit, you'll see the following in the terminal:

```
Started POST "/projects" for 127.0.0.1 at [timestamp]
Processing by ProjectsController#create as HTML
  Parameters: {"utf8"=>"?", "authenticity_token"=>"WRHnKqU...",
  "project"=>{"name"=>"Sublime Text 3", "description"=>"A text editor
  for everyone"}, "commit"=>"Create Project"}
```

The parameters are all listed right there.

All the hashes nested inside the `params` hash are also `HashWithIndifferentAccess` hashes. If you want to get the name key from the `project` hash here, you can use either `{ :name => "Sublime Text 3" }[:name]`, as in a normal `Hash` object, or `{ :name => "Sublime Text 3" }['name']`; you may use either the `String` or the `Symbol` version—it doesn't matter.

The `utf8` and `authenticity_token` params

There are two special parameters in the `params` hash: `utf8` and `authenticity_token`.

The `utf8` parameter is a hack for older browsers (read: old versions of Internet Explorer) to force them into UTF-8 compatibility. You can safely ignore this one.

The `authenticity_token` parameter is used by Rails to validate that the request is authentic. Rails generates this in the `<meta>` tag on the page (using `<%= csrf_meta_tags %>` in app/views/layouts/application.html.erb) and also stores it in the user's session. Upon the submission of the form, it compares the value in the form with the one in the session, and if they match the request is deemed authentic. Using `authenticity_token` mitigates cross-site request forgery (CSRF) attacks and so is a recommended best practice.

The first key in the `params` hash, `commit`, comes from the submit button of the form, which has the value `"Create Project"`. This is accessible as `params[:commit]`. The second key, `action`, is one of two parameters always available; the other is `controller`. These represent exactly what their names imply: the controller and action of the request, accessible as `params[:controller]` and `params[:action]`, respectively. The final key, `project`, is, as mentioned before, a `HashWithIndifferentAccess`. It contains the fields from your form and is accessible via `params[:project]`. To access the `name` key in the `params[:project]` object, use `params[:project][:name]`, which calls the `[]` method on `params` to get the value of the `:project` key and then, on the resulting hash, calls `[]` again, this time with the `:name` key to get the name of the project passed in.

`params[:project]` has all the data you need to pass to `Project.new`, but you can't just pass it directly in. If you try to substitute `project_params` with `params[:project]` in your controller, and then run `bundle exec rspec` again, you'll get the following error:

```
Failure/Error: click_button "Create Project"
ActiveModel::ForbiddenAttributesError:
  ActiveModel::ForbiddenAttributesError
```

STRONG PARAMETERS

Oooh, forbidden attributes. Sounds scary. But this is important: it's one form of security help that Rails gives you via a feature called *strong parameters*, new as of Rails 4.

You don't want to accept just any submitted parameters; you want to accept the ones that you want and expect, and no more. That way, someone can't mess around with your application by doing things like tampering with the form and adding new fields before submitting it.

Strong parameters vs. `attr_accessible`

Before Rails 4.0, Rails supported a feature called `attr_accessible` for protecting your models from unexpected attributes. You may see this used in older Rails projects—it involves listing out every field in the model that should be mass-assignable via user-submitted data.

The `attr_accessible` approach caused problems because different controllers might want to make available different sets of parameters for the same model, depending on the context. For instance, an admin area of the site may want to permit different fields than the user-facing area of the site—admins might have more fields, such as the ability to set the owner of a project.

The advantage of strong parameters is that the permitted parameters are now listed on a controller level, rather than at the model level. This allows for a higher degree of flexibility.

Change the `ProjectsController` code to add a new definition for the `project_params` method:

```
def create
  @project = Project.new(project_params)

  if @project.save
    flash[:notice] = "Project has been created."
    redirect_to @project
  else
    # nothing, yet
  end
end

private

def project_params
  params.require(:project).permit(:name, :description)
end
```

You now call the `require` method on your `params`, and you require that the `:project` key exists. You also allow it to have `:name` and `:description` entries—any other fields submitted will be discarded. Finally, you wrap up that logic into a method so you can use it in other actions, and you make it private so you don't expose it as some kind of weird action! You'll use this method in one other action in this controller later on—the `update` action.

With that done, run `bundle exec rspec` again, and you'll get a new error:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_button "Create Project"
   AbstractController::ActionNotFound:
     The action 'show' could not be found for ProjectsController
```

The test has made it through the `create` action, followed the redirect you issued, and now it's stuck on the next request—the page you redirected to, the `show` action.

The `show` action is responsible for displaying a single record's information. Retrieving a record to display is done by default using the record's ID. You know the URL for this page will be something like /projects/1, but how do you get the *1* from that URL? Well, when you use resource routing, as you've done already, the *1* part of this URL is available as `params[:id]`, just as `params[:controller]` and `params[:action]` are also automatically made available by Rails. You can then use this `params[:id]` parameter in your `show` action to find a specific `Project` object. In this case, the `show` action should be showing the newly created project.

Put the code from the following listing into app/controllers/projects_controller.rb to set up the `show` action. Make sure it comes above the `private` declaration, or you won't be able to use it as an action!

Listing 3.17 The `show` action of `ProjectsController`

```
def show
  @project = Project.find(params[:id])
end
```

You pass the `params[:id]` object to `Project.find`. This gives you a single `Project` object that relates to a record in the database, which has its `id` field set to whatever `params[:id]` is. If Active Record can't find a record matching that ID, it raises an `ActiveRecord::RecordNotFound` exception.

When you rerun `bundle exec rspec spec/features/creating_projects_spec.rb`, you'll get an error telling you that the `show` action's template is missing:

```
1) Users can create new projects with valid attributes
   Failure/Error: click_button "Create Project"
   ActionView::MissingTemplate:
     Missing template projects/show, application/show with
     {
      :locale => [:en],
      :formats => [:html],
      :variants => [],
      :handlers => [:erb, :builder, :raw, :ruby, :coffee, :jbuilder]
     }.

     Searched in:
     * ".../ticketee/app/views"
```

You can create the file app/views/projects/show.html.erb, with the following content for now, to display the project's name and description:

```
<h1><%= @project.name %></h1>
<p><%= @project.description %></p>
```

It's a pretty plain page for a project, but it'll serve our purpose.

When you run the test again with `bundle exec rspec spec/features/creating_projects_spec.rb`, you'll see this message:

```
1) Users can create new projects with valid attributes
     Failure/Error: expect(page).to have_content "Project has been
     created."
       expected to find text "Project has been created." in "Sublime
       Text 3 A text editor for everyone"
     # ./spec/features/creating_projects_spec.rb:13:in ...
```

This error message shows that the "Project has been created" text isn't being displayed on the page. You must put it somewhere, but where?

THE APPLICATION LAYOUT

The best location for this text is in the application layout, located at app/views/layouts/application.html.erb. This file provides the layout for all templates in your application, so it's a great spot to output a flash message—no matter what controller you set it in, it will be rendered on the page.

The application layout is quite the interesting file.

Listing 3.18 app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
<head>
  <title>Ticketee</title>
  <%= stylesheet_link_tag    'application', media: 'all',
    'data-turbolinks-track' => true %>
  <%= javascript_include_tag 'application',
    'data-turbolinks-track' => true %>
  <%= csrf_meta_tags %>
</head>
<body>

<%= yield %>

</body>
</html>
```

The first line sets up the doctype to be HTML for the layout, and three new methods are used: `stylesheet_link_tag`, `javascript_include_tag`, and `csrf_meta_tags`.

`stylesheet_link_tag` is for including CSS stylesheets from the app/assets/stylesheets directory. Using this tag results in the following output, where `[digest]` represents an MD5 hash of the contents of the file:

```
<link rel="stylesheet" href="/assets/projects-[digest].css?body=1"
media="all" data-turbolinks-track="true" />
<link rel="stylesheet" href="/assets/application-[digest].css?body=1"
media="all" data-turbolinks-track="true" />
```

The /assets path is served by a gem called Sprockets. In this case, the tag specifies the /assets/application-[digest].css path, and any route prefixed with /assets is served by

Sprockets. Sprockets provides a feature commonly referred to as the *asset pipeline*. When files are requested through the asset pipeline, they're preprocessed and then served out to the browser.

> **ASSET PIPELINE GUIDE** If you want to know about the ins and outs of the asset pipeline, read the official Ruby on Rails guide: http://guides.rubyonrails.org/asset_pipeline.html. It covers far more functionality than we'll touch on in this entire book!

There's also a second tag for the projects-[digest].css file. In development mode, Rails generates tags for all of your stylesheets and JavaScript separately, for ease of debugging. If you were to run the application in production mode, you'd get something very different:

```
<link data-turbolinks-track="true" href="/assets/application-[digest].css"
media="all" rel="stylesheet" />
```

This single stylesheet is all of your stylesheets, concatenated together and minified. That way, your users will load up all of your styles on their first visit, and the styles will be cached for the rest of their stay, increasing overall performance.

When the /assets/application-[digest].css asset is requested, Sprockets looks for a file named application.css in the asset paths for your application. The three asset paths it searches by default are app/assets, lib/assets, and vendor/assets, in that order. Some gems add extra paths to this list, so you're able to use assets from within those gems as well.

If the file has any additional extensions on it, such as a file called application.css.scss, Sprockets will look up a preprocessor for the .scss extension and run the file through that before serving it as CSS. You can chain together any number of extensions, and Sprockets will parse the file for each one, working right to left.

The application.css file that's searched for in this example lives at app/assets/application.css.scss. This has an additional .scss extension on it, so it'll be preprocessed by the Sass preprocessor before being served as CSS by the `stylesheet_link_tag` call in the application layout.

Using Sass or SCSS

For your CSS files, you can use the Sass or SCSS language to produce more powerful stylesheets. Your application depends on the sass-rails gem, which itself depends on sass, the gem for these stylesheets. We won't go into detail here because the Sass site covers most of that ground: http://sass-lang.com/.

Rails automatically generates stylesheets for each controller that uses Sass, as indicated by the .scss extensions.

The `javascript_include_tag` is for including JavaScript files from the JavaScript directories of the asset pipeline. When the `application` string is specified here, Rails loads the app/assets/javascripts/application.js file, which looks like this:

```
//= require jquery
//= require jquery_ujs
//= require turbolinks
//= require_tree .
```

This file includes some Sprockets-specific code that includes the jquery.js and jquery_ujs.js files, located in the jquery-rails gem. (The jquery-rails gem is listed in your Gemfile as a dependency of your application.) It also includes the JavaScript file for Turbolinks, which is a feature we'll discuss later. It compiles these three files, plus all the files in the app/assets/javascripts directory, with the `//= require_tree .` into one superfile called application.js.

In development mode, these JavaScript files all get included into your page separately, just like stylesheets, for ease of debugging. In the output of your page, you'll have the following:

```
<script src="/assets/jquery-[digest].js?body=1"
  data-turbolinks-track="true"></script>
<script src="/assets/jquery_ujs-[digest].js?body=1"
  data-turbolinks-track="true"></script>
<script src="/assets/turbolinks-[digest].js?body=1"
  data-turbolinks-track="true"></script>
<script src="/assets/projects-[digest].js?body=1"
  data-turbolinks-track="true"></script>
<script src="/assets/application-[digest].js?body=1"
  data-turbolinks-track="true"></script>
```

These files are also served through the Sprockets gem. As with your stylesheets, you can use an alternative syntax called CoffeeScript (http://coffeescript.org), which provides a simpler JavaScript syntax that compiles into proper JavaScript. Just as with the Sass stylesheets, Rails generates CoffeeScript files in app/assets/javascripts with the extension .coffee, indicating to Sprockets that they're to be parsed by a CoffeeScript interpreter before serving. You'll use CoffeeScript a little later, in chapter 9.

(If this is all going a bit over your head for now, don't worry. We'll come back to it later when we need to modify assets and add a design to the application.)

`csrf_meta_tags` is for protecting your forms from CSRF (http://en.wikipedia.org/wiki/CSRF) attacks. These types of attacks were mentioned a short while ago when we looked at the parameters for the `create` action. The `csrf_meta_tags` helper creates two `meta` tags, one called `csrf-param` and the other `csrf-token`. This unique token works by setting a specific key on forms that is then sent back to the server. The server checks this key, and if the key is valid, the form is deemed valid. If the key is invalid, an `ActionController::InvalidAuthenticityToken` exception occurs and the user's session is reset as a precaution.

Later in app/views/layouts/application.html.erb is this single line:

```
<%= yield %>
```

This line indicates to the layout where the current action's template is to be rendered. Create a new line just before `<%= yield %>`, and place the following code there:

```
<% flash.each do |key, message| %>
  <div><%= message %></div>
<% end %>
```

This code renders all the `flash` messages that are defined, regardless of their name and the controller they come from. These lines will display the `flash[:notice]` that you set up in the `create` action of the `ProjectsController`.

Run `bundle exec rspec` again, and you'll see that the test is now fully passing:

```
3 examples, 0 failures, 2 pending
```

Why do you have two pending tests? If you examine the output more closely, you'll see this:

```
.**

Pending: (Failures listed here are expected and do not affect your
suite's status)

  1) ProjectsHelper add some examples to (or delete)
     .../ticketee/spec/helpers/projects_helper_spec.rb
     # Not yet implemented
     # ./spec/helpers/projects_helper_spec.rb:14

  2) Project add some examples to (or delete)
     .../ticketee/spec/models/project_spec.rb
     # Not yet implemented
     # ./spec/models/project_spec.rb:4

Finished in 0.07268 seconds (files took 1.26 seconds to load)
3 examples, 0 failures, 2 pending
```

The key part is "or delete." Let's delete those two files, because you're not using them yet:

```
$ rm spec/models/project_spec.rb
$ rm spec/helpers/projects_helper_spec.rb
```

Afterward, run `bundle exec rspec` one more time:

```
.

Finished in 0.07521 seconds (files took 1.25 seconds to load)
1 example, 0 failures
```

Yippee! You've just written your first BDD test for this application! That's all there is to it.

If this process feels slow, that's how it's supposed to feel when you're new to anything. Remember when you were learning to drive a car? You didn't drive like Michael Schumacher as soon as you got behind the wheel. You learned by doing it slowly and methodically. As you progressed, you were able to do it more quickly, as you can all things with practice.

3.4.4 Committing changes

Now you're at a point where all your specs are running (just the one, for now). Points like this are great times to make a commit:

```
$ git add .
$ git commit -m "'Create a new project' feature complete."
```

> **Committing with older Git versions**
>
> If you're using a version of Git older than 2.0, running `git add .` when you want to stage the deletion of files will raise an error. You'll see something like the following:
>
> ```
> warning: You ran 'git add' with neither '-A (--all)' or '--ignore-removal',
> whose behaviour will change in Git 2.0 with respect to paths you removed.
> Paths like '[filename]' that are
> removed from your working tree are ignored with this version of Git.
> ```
>
> If this happens, you can add the `-A` option to `git add`, which will stage *all* file changes, including file deletions:
>
> ```
> $ git add -A .
> ```

You should commit often, because commits provide checkpoints you can revert back to if anything goes wrong. If you're going down a path where things aren't working, and you want to get back to the last commit, you can revert all your changes by using

```
$ git checkout .
```

> **USE `GIT CHECKOUT .` CAREFULLY!** This command doesn't prompt you to ask whether you're sure you want to take this action. You should be incredibly sure that you want to destroy your changes. If you're not sure and want to keep your changes while reverting back to the previous revision, it's best to use the `git stash` command. This command stashes your unstaged changes to allow you to work on a clean directory and lets you restore the changes using `git stash pop`.

With the changes committed to your local repository, you can push them off to the GitHub servers. If for some reason the code on your local machine goes missing, you have GitHub as a backup.

Run this command to push the code up to GitHub's servers:

```
$ git push
```

Commit early. Commit often.

3.4.5 Setting a page title

Before you completely finish working with this story, there's one more thing to point out: the templates (such as show.html.erb) are rendered *before* the layout. You can use this to your benefit by setting an instance variable such as `@title` in the `show` action's

template; then you can reference it in your application's layout to show a title for your page at the top of the tab or window.

To test that the page title is correctly implemented, add a little bit extra to your scenario for it. At the bottom of the test in spec/features/creating_projects_spec.rb, add the four lines shown in the following listing.

Listing 3.19 spec/features/creating_projects_spec.rb

```
project = Project.find_by(name: "Sublime Text 3")
expect(page.current_url).to eq project_url(project)

title = "Sublime Text 3 - Projects - Ticketee"
expect(page).to have_title title
```

The first line here uses the `find_by` method to find a project by its name. This finds the project that has just been created by the code directly above it. The second line ensures that you're on what should be the `show` action in the `ProjectsController`. The third and fourth lines find the `title` element on the page by using Capybara's `find` method and check using `have_title` that this element contains the page title of "Sublime Text 3 - Projects - Ticketee."

If you run `bundle exec rspec spec/features/creating_projects_spec.rb` now, you'll see this error:

```
1) Users can create new projects with valid attributes
   Failure/Error: expect(page).to have_title title
     expected "Ticketee" to include "Sublime Text 3 - Projects -
     Ticketee"
```

This error is happening because the `title` element doesn't contain all the right parts, but this is fixable! Write this code into the top of app/views/projects/show.html.erb:

```
<% @title = "Sublime Text 3 - Projects - Ticketee" %>
```

This sets up a `@title` instance variable in the template. Because the template is rendered before the layout, you're able to then use this variable in the layout.

But if a page doesn't have a `@title` variable set, there should be a default title of "Ticketee." To set this up, enter the following code in app/views/layouts/application.html.erb where the `title` tag currently is:

```
<title><%= @title || "Ticketee" %></title>
```

In Ruby, instance variables that aren't set return `nil` as their value. If you try to access an instance variable that returns a `nil` value, you can use `||` to return a different value, as in this example.

With this in place, the test should pass when you run `bundle exec rspec`:

```
1 example, 0 failures
```

Now that this test passes, you can change your code and have a solid base to ensure that whatever you change works as you expect. To demonstrate this point, let's change the code in `show` to use a *helper* instead of setting a variable.

Helpers are methods you can define in the files in app/helpers, and they're made available in your views. Helpers are for extracting the logic from the views; views should just be about displaying information. Every controller that comes from the controller generator has a corresponding helper, and another helper module exists for the entire application: the `ApplicationHelper` module, which lives at app/helpers/application_helper.rb.

Open app/helpers/application_helper.rb and insert the code from the following listing.

Listing 3.20 app/helpers/application_helper.rb

```
module ApplicationHelper
  def title(*parts)
    unless parts.empty?
      content_for :title do
        (parts << "Ticketee").join(" - ")
      end
    end
  end
end
```

When you specify an argument in a method beginning with the splat operator (`*`), any arguments passed from this point will be available in the method as an array. Here that array can be referenced as `parts`. Inside the method, you check to see if `parts` is `empty?` by using a keyword that's the opposite of `if`: `unless`. If no arguments are passed to the `title` method, `parts` will be empty and `empty?` will return `true`.

If parts are specified for the `title` method, then you use the `content_for` method to define a named block of content, giving it the name `title`. Inside this content block, you join the parts together using a hyphen (-), meaning that this helper will output something like `"Sublime Text 3 - Projects - Ticketee"`.

This helper method will build up a text string that you can use as the title of any page, including the default value of `"Ticketee"`, and all you need to do is call it from the view with the right arguments—an array of the parts that will make up the title of the page. Neat.

Now you can replace the title line in app/views/projects/show.html.erb with this:

```
<% title(@project.name, "Projects") %>
```

Let's replace the `title` tag line in app/views/layouts/application.html.erb with this code:

```
<title>
  <% if content_for?(:title) %>
    <%= yield(:title) %>
  <% else %>
    Ticketee
  <% end %>
</title>
```

This code uses a new method called `content_for?`, which checks that the specified content block is defined. It's defined only if `content_for(:title)` is called somewhere, such as in the template. If it is, you use `yield` and pass it the name of the content block, which causes the content for that block to be rendered. If it isn't, then you output the word *Ticketee*, and that becomes the title.

When you run this test again with `bundle exec rspec`, it will still pass:

```
1 example, 0 failures
```

That's a lot neater, isn't it? Create a commit for that functionality and push your changes:

```
$ git add .
$ git commit -m "Add title functionality for project show page"
$ git push
```

Next up, we'll look at how you can stop users from entering invalid data into your forms.

3.4.6 Validations

The next problem to solve is preventing users from leaving a required field blank. A project with no name isn't useful to anybody. Thankfully, Active Record provides *validations* for this purpose. Validations are run just before an object is saved to the database, and if the validations fail, the object isn't saved. Ideally, in this situation, you want to tell the user what went wrong so they can fix it and attempt to create the project again.

With this in mind, you can add another test to ensure that this happens. Add it to spec/features/creating_projects_spec.rb using the code from the following listing.

Listing 3.21 spec/features/creating_projects_spec.rb

```
scenario "when providing invalid attributes" do
  visit "/"

  click_link "New Project"
  click_button "Create Project"

  expect(page).to have_content "Project has not been created."
  expect(page).to have_content "Name can't be blank"
end
```

The first two lines are identical to the ones you placed in the other scenario. You should eliminate this duplication by making your code DRY (Don't Repeat Yourself!). This is another term you'll hear a lot in the Ruby world.[9] It's easy to extract common code from where it's being duplicated and move it into a method or module that you

[9] Some people like to use "DRY" like an adjective, and also refer to code that isn't DRY as WET (which doesn't actually stand for anything). We think those people are a bit weird.

can use instead of the duplication. One line of code is 100 times better than 100 lines of duplicated code.

To DRY up your code, define a `before` block before the first scenario. For RSpec, `before` blocks are run before *every* test in the file. Change spec/features/creating_projects_spec.rb to look like this.

Listing 3.22 spec/features/creating_projects_spec.rb

```
require "rails_helper"

RSpec.feature "Users can create new projects" do
  before do
    visit "/"

    click_link "New Project"
  end

  scenario "with valid attributes" do
    fill_in "Name", with: "Sublime Text 3"
    fill_in "Description", with: "A text editor for everyone"
    click_button "Create Project"

    expect(page).to have_content "Project has been created."

    project = Project.find_by(name: "Sublime Text 3")
    expect(page.current_url).to eq project_url(project)

    title = "Sublime Text 3 - Projects - Ticketee"
    expect(page).to have_title title
  end

  scenario "when providing invalid attributes" do
    click_button "Create Project"

    expect(page).to have_content "Project has not been created."
    expect(page).to have_content "Name can't be blank"
  end
end
```

There! That looks a lot better!

Now when you run `bundle exec rspec`, it will fail because it can't see the error message that it's expecting to see on the page:

```
1) Users can create new projects when providing invalid attributes
   Failure/Error: expect(page).to have_content "Project has not been
   created."
     expected to find text "Project has not been created." in "Project
     has been created."
```

ADDING VALIDATIONS

To get this test to do what you want it to do, you'll need to add a validation. Validations are defined on the model and are run before the data is saved to the database.

To define a validation to ensure that the `name` attribute is provided when a project is created, open the app/models/project.rb file and make it look like the following listing.

Listing 3.23 app/models/project.rb

```
class Project < ActiveRecord::Base
  validates :name, presence: true
end
```

The `validates` method's usage is the same as how you used it in chapter 1. It tells the model that you want to validate the `name` field, and that you want to validate its presence. There are other kinds of validations as well; for example, the `:uniqueness` key, when passed `true` as the value, validates the uniqueness of this field as well, ensuring that only one record in the table has that specific value.

> **UNIQUENESS VALIDATION** There are potential gotchas with the Active Record uniqueness validation that may allow duplicate data to be saved to the database. We're intentionally ignoring them for now, but we'll cover them, and how to resolve the issues they raise, in section 6.1.

With the `presence` validation in place, you can experiment with the validation by using the Rails console, which allows you to have all the classes and the environment from your application loaded in a sandbox environment. You can launch the console with this command,

```
$ rails console
```

or with its shorter alternative:

```
$ rails c
```

If you're familiar with Ruby, you may realize that this is effectively IRB with some Rails sugar on top. If you're new to both, IRB stands for Interactive Ruby, and it provides an environment for you to experiment with Ruby without having to create new files. The console prompt looks like this:[10]

```
Loading development environment (Rails 4.2.1)
irb(main):001:0>
```

At this prompt, you can enter any valid Ruby, and it'll be evaluated.

For now, the purpose of opening this console is to test the newly appointed validation. To do this, try to create a new project record by calling the `create` method. The `create` method is similar to the `new` method, but it attempts to create an object and then a database record for it rather than just the object. You use it identically to the `new` method:

```
irb(main):001:0> Project.create
=> #<Project id: nil, name: nil, description: nil, created_at: nil,
     updated_at: nil>
```

[10] Alternatively, you may see something similar to `ruby-2.2.1:001 >`, which is fine.

Here you get a new `Project` object with the `name` and `description` attributes set to `nil`, as you should expect, because you didn't specify it. The `id` attribute is `nil` too, which indicates that this object isn't persisted (saved) in the database.

If you comment out or remove the validation from the `Project` class and type `reload!` in your console, the changes you just made to the model are reloaded. When the validation is removed, you have a slightly different outcome when you call `Project.create`:

```
irb(main):001:0> Project.create
=> #<Project id: 1, name: nil, description: nil,
     created_at: [timestamp], updated_at: [timestamp]>
```

Here, the `name` field is still expectedly `nil`, but the other three attributes have values. Why? When you call `create` on the `Project` model, Rails builds a new `Project` object with any attributes you pass it and checks to see if that object is valid.[11] If it is, Rails sets the `created_at` and `updated_at` attributes to the current time and then saves the object to the database. After it's saved, the `id` is returned from the database and set on your object. This object is valid, according to Rails, because you removed the validation, so Rails goes through the entire process of saving.

The `create` method has a bigger, meaner brother called `create!` (pronounced *create BANG!*). Re-add or uncomment the validation from the model, and type `reload!` in the console, and you'll see what this mean variant does with this line:

```
irb(main):001:0> Project.create!
ActiveRecord::RecordInvalid: Validation failed: Name can't be blank
```

The `create!` method, instead of nonchalantly handing back a `Project` object regardless of any validations, raises an `ActiveRecord::RecordInvalid` exception if any of the validations fail; it shows the exception followed by a large stack trace, which you can safely ignore for now. You're notified which validation failed.

To stop it from failing, you must pass in a `name` attribute, and `create!` will happily return a saved `Project` object:

```
irb(main):002:0> Project.create!(name: "Sublime Text 3")
=> #<Project id: 2, name: "Sublime Text 3", description: nil,
     created_at: [timestamp], updated_at: [timestamp]>
```

That's how to use `create!` to test your validations in the console.

You've created some bad data in your database during this experimentation, so you should clean that up before you continue:

```
irb(main):003:0> Project.delete_all
=> 2
```

Back in your `ProjectsController`, you use the method shown in the following listing instead.

---

[11] The first argument for this method is the attributes. If no argument is passed, then all attributes default to their default values.

Listing 3.24 Part of the `create` action of `ProjectsController`

```
def create
  @project = Project.new(project_params)

  if @project.save
    ...
```

The `save` method doesn't raise an exception if validations fail, as `create!` does; instead it returns `false`. If the validations pass, `save` returns `true`.

You can use this to your advantage to show the user an error message when `save` returns `false` by using it in an `if` statement. Make the `create` action in the `ProjectsController` look like the following listing.

Listing 3.25 The new `create` action from `ProjectsController`

```
def create
  @project = Project.new(project_params)

  if @project.save
    flash[:notice] = "Project has been created."
    redirect_to @project
  else
    flash.now[:alert] = "Project has not been created."
    render "new"
  end
end
```

`flash` vs. `flash.now`

The controller action in listing 3.25 uses two different methods to access the array of flash messages for your page—`flash` and `flash.now`. What's the difference?

`flash` is the standard way of setting flash messages, and it will store the message to display on the very *next* page load. You do this immediately before issuing redirects—in this case you redirect immediately to the `show` page in the `ProjectsController`, and that page is the next page load, meaning that the flash message displays on the `show` view.

`flash.now` is an alternative way of setting flash messages, and it will store the message to display on the *current* page load. In this case, you don't redirect anywhere, you simply render a view out from the same action, so you need to use `flash.now` to make sure the user sees the error message when you render the `new` view.

There's also a third method—`flash.keep`—but this is used very rarely. If you want to keep an existing flash message around for another request, you can call `flash.keep` in your controller, and the flash message will hang around for a little while longer.

If you were to use `flash` instead of `flash.now` in this case, the user would see the message twice—once on the current page and once on the next page.

If the `@project` object has a `name` attribute—meaning it's valid—`save` returns `true` and executes everything between `if` and `else`. If it isn't valid, then everything between `else` and the following `end` is executed. In the `else`, you specify a different key for the flash message because you'll want to style alert messages differently from notices later in the application's lifecycle. When good things happen, the messages for them will be colored with a green background; when bad things happen, red.

When you run `bundle exec rspec spec/features/creating_projects_spec.rb` now, the line in the spec that checks for the "Project has not been created" message doesn't fail, so it goes to the next line, which checks for the "Name can't be blank" message. You haven't done anything to make this message appear on the page yet, which is why the test is failing again:

```
1) Users can create new projects when providing invalid attributes
   Failure/Error: expect(page).to have_content "Name can't be blank"
     expected to find text "Name can't be blank" in "Project has not
     been created. New Project Name Description"
```

The validation errors for the project aren't being displayed on this page, which is causing the test to fail. To display validation errors in the view, you need to code something up yourself.

When an object fails validation, Rails will populate the `errors` of the object with any validation errors. You can test this back in your Rails console:

```
irb(main):001:0> project = Project.create
=> #<Project id: nil, name: nil, description: nil, created_at: nil,
     updated_at: nil>
rb(main):002:0> project.errors
=> #<ActiveModel::Errors:0x007fd5938197f8 @base=#<Project id: nil,
     name: nil, description: nil, created_at: nil, updated_at: nil>,
     @messages={:name=>["can't be blank"]}>
```

`ActiveModel::Errors` provides some nice helper methods for working with validation errors that you can use in your views to display the errors to the user. In the app/views/projects/new.html.erb file, directly under the `form_for` line, on a new line, insert the following into app/views/projects/new.html.erb to display the error messages in the form:

```
<% if @project.errors.any? %>
 <div id="error_explanation">
   <h2><%= pluralize(@project.errors.count, "error") %>
   prohibited this project from being saved:</h2>

   <ul>
     <% @project.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
     <% end %>
   </ul>
 </div>
<% end %>
```

Error messages for the object represented by your form, the `@project` object, will now be displayed by `each`.

When you run `bundle exec rspec`, you'll now get this output:

```
2 examples, 0 failures
```

Commit and push, and then you're done with this story!

```
$ git add .
$ git commit -m "Add validation to ensure names are specified when
  creating projects"
$ git push
```

3.5 Summary

We first covered how to version-control an application, which is a critical part of the application development cycle. Without proper version control, you're liable to lose valuable work or be unable to roll back to a known working stage. We used Git and GitHub as examples, but you may use an alternative, such as SVN or Mercurial, if you prefer. This book covers only Git, because covering everything would result in a multi-volume series, which is difficult to transport.

Next we covered the basic setup of a Rails application, which started with the `rails new` command that initializes an application. Then we segued into setting up the Gemfile to require certain gems for certain environments, such as RSpec in the test environment. You learned about the beautiful Bundler gem in the process, and then you ran the installers for these gems so your application was fully configured to use them. For instance, after running `rails g rspec:install`, your application was set up to use RSpec and so will generate RSpec specs rather than the default Test::Unit tests for your models and controllers.

Finally, you wrote the first story for your application, which involved generating a controller and a model as well as getting an introduction to RESTful routing and validations. With this feature of your application covered by RSpec, you can be notified if it's broken by running `bundle exec rspec`. This command runs all the tests of the application and lets you know if everything is working or if anything is broken. If something is broken, the spec will fail, and then it's up to you to fix it. Without this automated testing, you'd have to do it all manually, and that isn't any fun.

Now that you've got a first feature under your belt, let's get into writing the next one!

WEB DEVELOPMENT/RUBY

Rails 4 IN ACTION

Bigg • Katz • Klabnik • Skinner

Rails is a full-stack, open source web framework powered by Ruby. Now in version 4, Rails is mature and powerful, and to use it effectively you need more than a few Google searches. You'll find no substitute for the guru's-eye-view of design, testing, deployment, and other real-world concerns that this book provides.

Rails 4 in Action is a hands-on guide to the subject. In this fully revised new edition, you'll master Rails 4 by developing a ticket-tracking application that includes RESTful routing, authentication and authorization, file uploads, email, and more. Learn to design your own APIs and successfully deploy a production-quality application. You'll see test-driven development and behavior-driven development in action throughout the book, just like in a top Rails shop.

What's Inside

- Creating your own APIs
- Using RSpec and Capybara
- Emphasis on test-first development
- Fully updated for Rails 4

For readers of this book, a background in Ruby is helpful but not required. No Rails experience is assumed.

Ryan Bigg, **Yehuda Katz**, **Steve Klabnik**, and **Rebecca Skinner** are contributors to Rails and active members of the Rails community.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/rails-4-in-action

Free eBook
SEE INSERT

"There's no better source for Rails 4. This book blows away the competition."
—Damien White, Visoft, Inc.

"A gentle yet thorough guide to Rails 4."
—William Wheeler
ProData Computer Services

"Very clear, with excellent examples. A must-read for everyone in the Rails world."
—Michele Bursi, Nokia

"Well-written, intuitive, and easy to understand."
—Lee Allen
SecuritySession.com

ISBN 13: 978-1-617291-09-8
ISBN 10: 1-617291-09-9
54999
9 781617 291098

MANNING $49.99 / Can $57.99 [INCLUDING eBOOK]