

SAMPLE CHAPTER

# Rails 3

## IN ACTION

Ryan Bigg  
Yehuda Katz





***Rails 3 in Action***

Ryan Bigg  
Yehuda Katz

**Chapter 3**

## *brief contents*

---

- 1 ■ Ruby on Rails, the framework 1
- 2 ■ Testing saves your bacon 23
- 3 ■ Developing a real Rails application 44
- 4 ■ Oh CRUD! 83
- 5 ■ Nested resources 99
- 6 ■ Authentication and basic authorization 117
- 7 ■ Basic access control 136
- 8 ■ More authorization 164
- 9 ■ File uploading 213
- 10 ■ Tracking state 243
- 11 ■ Tagging 286
- 12 ■ Sending email 312
- 13 ■ Designing an API 347
- 14 ■ Deployment 385
- 15 ■ Alternative authentication 412
- 16 ■ Basic performance enhancements 434
- 17 ■ Engines 468
- 18 ■ Rack-based applications 516

# Developing a real Rails application

---

## ***This chapter covers***

- Building the foundation for a major app
- Diving deep into app foundations
- Generating the first functionality for an app

This chapter gets 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 feature passing.

For the remainder of the book, this application is the main focus. We 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, then refining the feature until it's viewed as complete before moving on to the next one.<sup>1</sup>

---

<sup>1</sup> More information about Agile can be found on Wikipedia: [http://en.wikipedia.org/wiki/Agile\\_software\\_development](http://en.wikipedia.org/wiki/Agile_software_development).

Some of the concepts covered in this chapter were explained in chapter 1. Rather than using scaffolding, as you did previously, you write this application from the ground up using the behavior-driven development (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 Application setup

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 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 *stories*, and we use Cucumber to develop the stories.

Start with the most basic story and ask your client how they want it to behave. Then write the Cucumber scenario for it using the client's own terms. You define the step definitions when it's time to implement the *function* of the story. The client can also provide helpful information about the *form*—what the application should look like. With the function and form laid out, you have a pretty good idea of what the client wants.

You may find it helpful to put these stories into a system such as Pivotal Tracker (<http://pivotaltracker.com>) so you can keep track of them. Pivotal Tracker allows you to assign points of difficulty to a story and then, over a period of weeks, estimate which stories can be accomplished in the next iteration on the basis of how many were completed in previous weeks. This tool is exceptionally handy to use when working with clients because the client can enter stories and then follow the workflow process. In this book, we don't use Pivotal Tracker because we aren't working with a real client, but this method is highly recommended.

For this example application, your imaginary client, who has limitless time and budget (unlike those 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, then you can move on to the next prioritized chunk of work.

BDD is used all the way through the development process. It provides 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 is 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 Cucumber stories, the client may ask why all of 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: Tests will make your code more maintainable, but they won't make your code bug-proof.

Features also give you a clearer picture of what the clients *really* want. By having it all written down in features, you have a solid reference to point to if clients say they suggested something different. Story-driven development is simply BDD with emphasis on things a user can actually 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.

To start building the application you'll be developing 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 checks tickets on trains. It also has to do with this project being a ticket-tracking application, and a Rails application, at that. To generate this application, run this command:

```
rails new ticketee
```

Presto, it's done! From this bare-bones application, you'll build an application that

- 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 of 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 any other, but this book uses Git exclusively.

### 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.

### 3.1.2 Version control

It is 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 to the commit. Additionally, you can create branches for experimental features and work on those independently of the main code base 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 is 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.<sup>2</sup>

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 plugin 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 do not have Git already installed, GitHub's help site offers installation guides for Mac,<sup>3</sup> Linux,<sup>4</sup> and Windows.<sup>5</sup> The precompiled installer should work well for Macs, and the package distributed versions (APT, eMerge, and so on) work well for Linux machines. For Windows, the *msysGit* application does just fine.

For an online place to put your Git repository, we recommend GitHub,<sup>6</sup> which offers free accounts. If you set up an account now, you can upload your code to GitHub as you progress, ensuring that you don't lose it if anything were to happen to your computer. 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.<sup>7</sup> Once you generate the key, copy the public key's content (usually found at `~/.ssh/id_rsa.pub`) into the SSH Public Key field on the Signup page or, if you've already signed up, click the Account Settings link (see figure 3.1) in the menu at the top, select SSH Public Keys, and then click Add Another Public Key to enter it there (see figure 3.2).



**Figure 3.1** Click Account Settings.

<sup>2</sup> <http://progit.org/book/>.

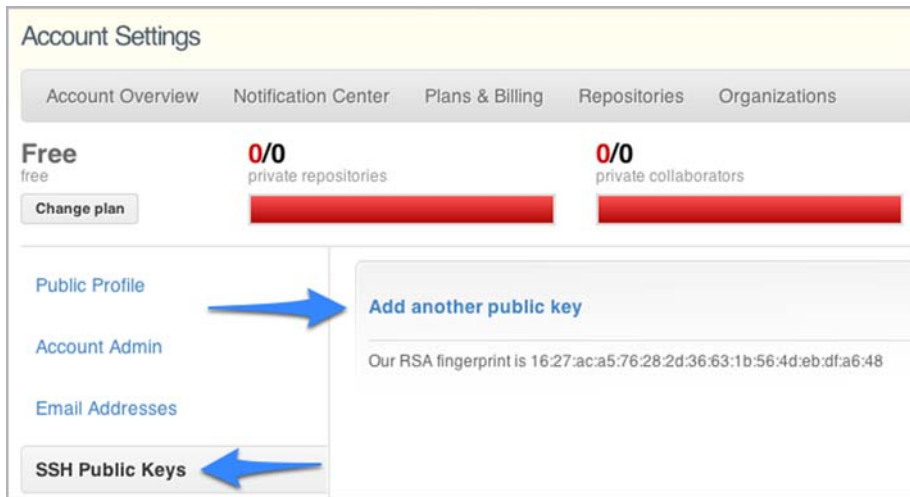
<sup>3</sup> <http://help.github.com/mac-set-up-git/>. Note this lists four separate ways, not four separate steps, to install Git.

<sup>4</sup> <http://help.github.com/linux-set-up-git/>.

<sup>5</sup> <http://help.github.com/win-set-up-git/>.

<sup>6</sup> <http://github.com>.

<sup>7</sup> A guide for this process can be found at <http://help.github.com/linux-key-setup/>.



**Figure 3.2** Add an SSH key.

Now that you're set up with GitHub, click the New Repository button on the dashboard to begin creating a new repository (see figure 3.3).

On this page, enter the Project Name as *ticketee* and click the Create Repository button to create the repository on GitHub. Now you are on your project's page. Follow the instructions, especially concerning the configuration of your identity. In listing 3.1, replace "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. The `git` commands should be typed into your terminal or command prompt.



**Figure 3.3** Create a new repository.

### Listing 3.1 Configuring your identity in GitHub

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

You already have a *ticketee* directory, and you're probably inside 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

```
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 as a checkpoint of 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

```
git commit -m "Generated the Rails 3 application"
```

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

```
Created initial commit cdae568: Generated the Rails 3 application
35 files changed, 9280 insertions(+), 0 deletions(-)
```

`cdae568` 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 (additional line count), 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 an output similar to the following listing.

### Listing 3.2 Viewing the commit log

```
commit cdae568599251137d1ee014c84c781917b2179e1
Author: Your Name <you@example.com>
Date: [date stamp]

    Generated the Rails 3 application
```

The hash after the word `commit` is the *long commit ID*; it's the longer version of the previously sighted 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.<sup>8</sup> With that commit in your repository, you have something to push to GitHub, which you can do by running

```
git remote add origin git@github.com:yourname/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 github username]/ticketee.git` path, which connects to the repository using SSH. The next 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.

---

<sup>8</sup> The chances of this happening are 1 in 268,435,456.

**Listing 3.3 Terminal**

```
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:rails3book/ticketee.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

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

**3.1.3 The Gemfile and generators**

The Gemfile is used for tracking which gems are used in your application. The Bundler gem is responsible for everything to do with this file; it's the Bundler's job to ensure that all the gems are installed when your application is initialized. Let's look at the following listing to see how this looks inside (commented lines are removed for simplicity).

**Listing 3.4 Gemfile**

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

gem 'rails', '3.1.0'

gem 'sqlite3'

group :assets do
  gem 'sass-rails', "~> 3.1.0"
  gem 'coffee-rails', "~> 3.1.0"
  gem 'uglifier'
end

gem 'jquery-rails'

group :test do
  # Pretty printed test output
  gem 'turn', :require => false
end
```

In this file, Rails sets a source to be <http://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 3.1.0.beta of the rails gem. Bundler inspects the dependencies of the requested gem as well as all gem dependencies of those dependencies (and so on), then does what it needs to do to make them 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 would need to take out this line and replace it with the relevant gem, such as `mysql2` for MySQL or `pg` for PostgreSQL.

The assets group inside the Gemfile contains two gems called `sass-rails` and `coffee-rails`. The `sass-rails` gem provides a bridge into the sass gem, which

provides much better templating for stylesheets, and the `coffee-rails` gem provides a similar bridge between Rails and the CoffeeScript templating languages. We'll look at these in further detail when they're required.

Finally, at the bottom of the Gemfile, the `turn` gem is specified. This gem is for making the `Test::Unit` output a lot prettier, but you're not going to be using `Test::Unit`, you can remove these lines:

```
group :test do
  # Pretty printed test output
  gem 'turn', :require => false
end
```

While you're removing `Test::Unit` things, remove the `test` directory too: you won't need that either. You'll be using the `spec` directory for your tests instead.

Chapter 2 focused on BDD and, as was more than hinted at, you'll be using it to develop this application. First, alter the Gemfile to ensure you have the correct gems for RSpec and Cucumber for your application. Add the lines from the following listing to the bottom of the file.

### Listing 3.5 Check for RSpec and Cucumber gems

```
group :test, :development do
  gem 'rspec-rails', '~> 2.5'
end

group :test do
  gem 'cucumber-rails'
  gem 'capybara'
  gem 'database_cleaner'
end
```

In the Gemfile, you specify that you wish to use the latest 2.x release of RSpec in the `test` and `development` groups. You put this gem inside the `development` group because without it, the tasks you can use to run your specs will be unavailable. 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.

With `rspec-rails`, you specified a version number with `~> 2.5`, which tells RubyGems you want RSpec 2.5 *or higher*. This means when RSpec releases 2.5.1 or 2.6 and you go to install your gems, RubyGems will install the latest version it can find rather than only 2.5.

A new gem is used in listing 3.5: *Capybara*. Capybara is a browser simulator in Ruby that is used for *integration testing*. Cucumber and Capybara are two distinct entities. Cucumber is the testing tool that interacts with Capybara to perform tasks on a simulated application. Integration testing using Cucumber and Capybara ensures that when a link is clicked in your application, it goes to the correct page, or when you fill in a form and click the Submit button, an onscreen message tells you it was successful.

Capybara also supports real browser testing by launching an instance of Firefox. You can then test your application's JavaScript, which you'll use extensively in chapter 8.

An alternative to Capybara is Webrat, which is now less preferred because of the cleaner syntax and real browser testing features of Capybara. Capybara is the better alternative, hands down.<sup>9</sup>

Another new gem in listing 3.5 is the `database_cleaner` gem, which is used by `cucumber_rails` to clear out the database at the end of each test run to ensure you're working with a pristine state each time.

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*.

Development is used for your local application, such as when you're playing with it in the browser. In development mode, page and class caching is 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. We're not there yet.

Test 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.

Production is used when you finally deploy your application. This mode is designed for speed, and any changes you make to your application's classes are not effective until the server is restarted.

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

```
Bundler.require(:default, Rails.env) if defined?(Bundler)
```

To install these gems to your system, run `bundle install --binstubs` at the root of your application. It tells the Bundler gem to read your Gemfile and install the gems specified in it.

The `--binstubs` option stores executable files in the `bin` directory for the gems that have executables. Without it, you'd have to type `bundle exec rspec spec` to run your RSpec tests. With it, you can just run `bin/rspec spec`. Much better! You don't need to run `bundle install --binstubs` every time because Bundler remembers this configuration option for later.

You don't even have to run `bundle install`! Just `bundle` will do the same thing.

**NOTE** If you're running Ubuntu, you must install the `build-essential` package because some gems build native extensions and require the `make` utility. You may also have to install the `libxslt1-dev` package because the `nokogiri` gem (a dependency of Cucumber) depends on it. You'll also need to install the `libsqlite3-dev` package to allow the `sqlite3` gem to install.

---

<sup>9</sup> As of this writing, Webrat has an open bug: <https://github.com/brynary/webrat/pull/46>. If we were using Webrat, we could run into it and be stuck in the mud (so to speak).

This command installs not only the `rspec-rails`, `cucumber`, and `capbara` gems but also their dependencies (and so on)! How great is *that*? You're just getting started.

The `bundle install --binstubs` command also creates a `Gemfile.lockfile` that contains a list of the gems and their relative versions. Once `Gemfile.lock` is created, whenever you run `bundle install`, Bundler reads *this* file rather than `Gemfile` to work out the dependencies of the application and installs from it. You commit this file to your repository so that when other people work on your project and run `bundle install`, they get exactly the same versions that you have.

Next, you want to generate the skeleton for Cucumber. A *generator* can generate either static or dynamic content, depending on the generator. For the Cucumber skeleton generator, it's set content. To run this generator, you use the `rails` command

```
rails generate cucumber:install
```

or simply

```
rails g cucumber:install
```

Rails doesn't mind if you use `generate` or `g`: it's all the same to Rails.

Let's not mind too much about all the files it has generated at the moment; they're explained in time. With the Cucumber skeleton now generated, you have a base on which to begin writing your features.

While you're generating things, you may as well run the RSpec generator too:

```
rails generate rspec:install
```

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 "Ran the cucumber and rspec generators"
git push
```

### 3.1.4 Database configuration

By default, Rails uses a database system called SQLite3, which stores each environment's database in separate files inside 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, with gems 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` (development configuration shown in the following listing) and alter the settings to the new database system.

#### Listing 3.6 config/database.yml

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

For example, if you want to use PostgreSQL, you change the settings to read like the following listing. It's common convention, but not mandatory, to call the environment's database [app\_name]\_[environment].

### Listing 3.7 config/database.yml

```
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. 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 switch your data over for you.

It's generally wise to use different names for the different database environments because if you use the same database in development and test, the database will be emptied of all data when the tests are run, eliminating anything you may have set up in development mode. You should never work on the live production database directly unless you are 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 utf8 for the database, using this setup in the config/database.yml file:

```
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 (UCS Transformation Format–8-bit), eliminating any potential encoding issues that may be encountered otherwise.

That's database configuration in a nutshell. Now we look at how to use a pre-prepared stylesheet to make an application look prettier than its unstyled brethren.

### 3.1.5 Applying a stylesheet

So that your application looks good as you're developing it, we have a pre-prepared stylesheet you can use to style the elements on your pages. You can download the stylesheet from <http://github.com/rails3book/ticketee/raw/master/app/assets/stylesheets/application.css> and put it in the app/stylesheets directory. By default, your Rails application includes this stylesheet; you can configure it in the app/views/layouts/application.html.erb file using this line:

```
<%= stylesheet_link_tag "application" %>
```

No further configuration is necessary: just drop and use. Simple.

With a way to make your application decent looking, let's develop your first feature to use this new style: creating projects.

## 3.2 First steps

You now have version control for your application, and you're hosting it on GitHub. You also cheated a little and got a pre-prepared stylesheet.<sup>10</sup>

It's now time to write your first Cucumber feature, which isn't nearly as daunting as it sounds. We explore things such as models and RESTful routing while we do it. It'll be simple, promise!

### 3.2.1 Creating projects

The CRUD (*create, read, update, delete*) acronym is something you 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 from your database throughout your application. The following section goes through the beginnings of generating a CRUD interface for a resource called *Project* by applying the BDD practices 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 continue to apply these practices to ensure you have a stable and maintainable application. Let's get into it!

The first story for your application is the creation (the C in CRUD). You create a resource representing projects in your application by writing a feature, creating a controller and model, and adding a resource route. Then you add a validation to ensure that no project can be created without a name.

First, you generate a feature file at `features/creating_projects.feature`, and in this file, you put the story you would have discussed with the client, as shown in the following listing.

#### Listing 3.8 features/creating\_projects.feature

```
Feature: Creating projects
  In order to have projects to assign tickets to
  As a user
  I want to create them easily

  Scenario: Creating a project
    Given I am on the homepage
    When I follow "New Project"
    And I fill in "Name" with "TextMate 2"
    And I press "Create Project"
    Then I should see "Project has been created."
```

---

<sup>10</sup> You wouldn't have a pre-prepared stylesheet in the real world, where designers would design at the same time you're developing features.

To run all features for your application, run `rake cucumber:ok`. This command causes the following error message:

```
...db/schema.rb doesn't exist yet. Run "rake db:migrate" to
create it then try again. If you do not intend to use a database,
you should instead alter ...ticketee/config/application.rb to limit
the frameworks that will be loaded
```

The `db/schema.rb` file referenced here (when generated) will contain a Ruby version of your database's schema, which can be used to import the structure of your database. The great thing is that this file is database agnostic, so if you choose to switch databases during the life of your project, you won't have to re-create this schema file. To generate this file, run this command:

```
rake db:migrate
```

For now, this command prints out the standard first line of rake output, the path to the application:

```
(in /home/us/ticketee)
```

When you run any Rake task, the first line will be this line. Its only purpose is to indicate which directory you're running inside. When you run this particular task, it generates the `db/schema.rb` file your feature required. Therefore, you can run `rake cucumber:ok` again to have it fail on the second step of your feature:

```
When I follow "New Project"
  no link with title, id or text 'New Project' found
```

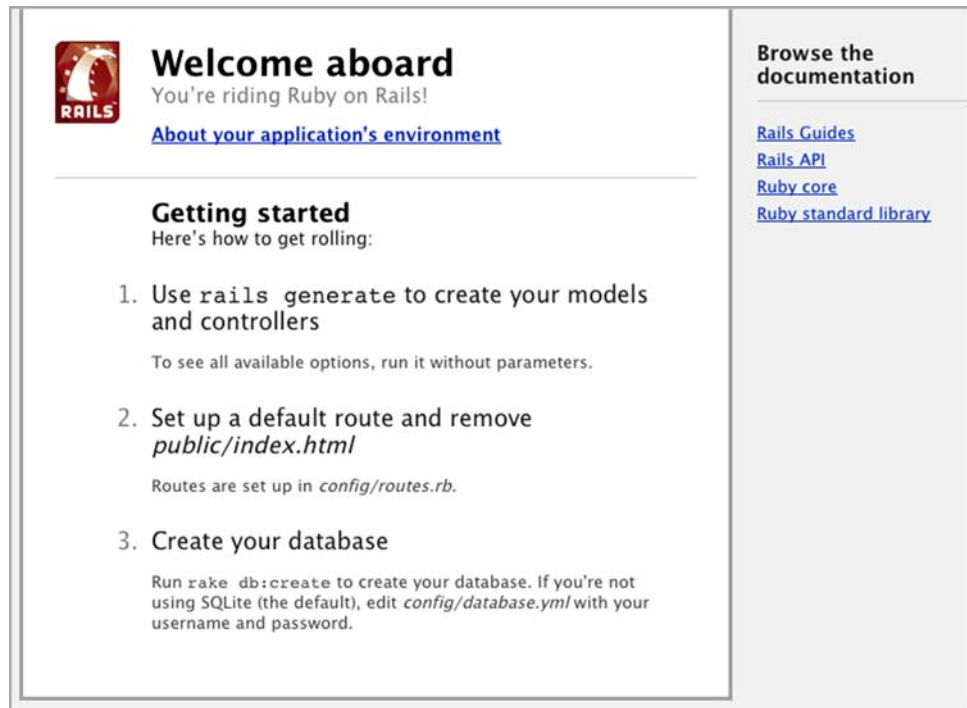
The first step here passes, but you haven't written a step definition for it, as you did in chapter 2! It nevertheless passes because of the `features/step_definitions/web_steps.rb` file, which was generated when you ran the `rails generate cucumber:install` command. This file contains a step definition that matches your first step:

```
Given /^(?:|I )am on (.+)\$/ do |page_name|
  visit path_to(page_name)
end
```

First, Cucumber interprets this step on the basis of the definition from within this file. The `visit` method inside this definition comes from Capybara and emulates going to a specific path in a virtual browser. The `path_to` method is also defined, but it's in the `features/support/paths.rb` file inside the `NavigationHelpers` module, again provided by Cucumber:

```
module NavigationHelpers
  ...
  def path_to(page_name)
    case page_name
      when /the home\s?page/
        '/'
    ...
  end
end
```

Therefore, the path that Capybara will visit is `/`. If you start the server for the application by running `rails server` (or `rails s`) and navigate to <http://localhost:3000> in



**Figure 3.4** Welcome aboard: take #2

your browser, you'll see the famous Welcome Aboard page from Rails (the same one you saw when you generated your first application in chapter 1), shown in figure 3.4.

This is the page Capybara sees, and the reason the first step passes is that Capybara can go to the homepage successfully. This Welcome Aboard page lives at `public/index.html` in your application. To proceed, delete this file using this command:

```
git rm public/index.html
```

Run `rake cucumber:ok` again. This time you'll see that the first step fails:

```
Given I am on the homepage
  No route matches [GET] "/" (ActionController::RoutingError)
```

The first step fails because you removed the `public/index.html` file that Rails was originally serving up at the root path, so the task goes to Rails. Rails claims it can't handle that route and throws an exception. You have to tell Rails what to do with a request for `/`, or the *root route* comes in. You can do this easily in `config/routes.rb`. At the moment, this file has the content seen in the following listing (comments removed).

#### Listing 3.9 `config/routes.rb`

```
Ticketee::Application.routes.draw do
end
```

The comments are good for a read if you're interested in the other routing syntax, but you needn't look at these right now. To define a root route, you put the following directly under the first line of this file:

```
root :to => "projects#index"
```

This defines the root route to point at the `ProjectsController`'s `index` action. This controller doesn't exist yet, and when you run `rake cucumber:ok` again, Cucumber complains about a missing constant, `ProjectsController`:

```
Given I am on the homepage
  uninitialized constant ProjectsController ...
```

To define this constant, you generate a *controller*. The controller is the first port of call for your routes (as you can see now!) and is responsible for querying the model for information inside 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 this controller, run this command:

```
rails g controller projects
```

This command produces output similar to the output produced when you ran `rails new` earlier, but this time it creates files just for the projects controller, the most important of these being the controller itself, which is housed in `app/controllers/projects_controller.rb`. This is where all the actions will live, just like your `app/controllers/purchases_controller.rb` back in chapter 1. Before we dive into that, a couple of 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 use in your view, and they come as blank slates when they are 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`.

`invoke rspec` shows that the RSpec generator was invoked at this point also. This means that RSpec has generated a new file at `spec/controllers/projects_controller.rb`, which you can use to test your controller; you won't use this for the time being. By generating RSpec tests rather than `Test::Unit` tests, a long-standing issue within Rails has been fixed. (In previous versions of Rails, even if you specified the RSpec gem, all the default generators would still generate `Test::Unit` tests. With Rails 3, the testing framework you use is just one of a large number of configurable things in your application.)

As mentioned previously when you ran `rails generate rspec:install`, this generator has generated an RSpec controller spec for your controller, `spec/controllers/projects_controller_spec.rb`, rather than a `Test::Unit` functional test. This file is used for testing the individual actions in your controller.

Now, 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 `rake cucumber:ok` again, it declares that the index action is missing:

```
Given I am on the homepage
  The action 'index' could not be found for ProjectsController ...
```

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

### Listing 3.10 `app/controllers/projects_controller.rb`

```
class ProjectsController < ApplicationController
  def index

  end
end
```

If you run `rake cucumber:ok` again, this time Rails complain of a missing template `projects/index`:

```
Given I am on the homepage
Missing template projects/index, application/index
  with {:handlers=>[:erb, :builder],
       :formats=>[:html],
       :locale=>[:en, :en]}.

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

The error message isn't the most helpful, but it's quite detailed. 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 this file is composed of two parts: the format followed by the handler. In your output, you've got a handler of either `:erb` or `:builder` and a format of `:html`, so it's fair to assume from this that the file it's looking for is either `index.html.erb` or `index.html.builder`. Either of these is fine, but we'll use the first one for consistency's sake.

The first part, *index*, is the name of the action; that's the easy part. The second part, *html*, indicates the format of this template. Actions in Rails can respond to different formats (using `respond_to`, which you saw in chapter 1); the default format is *html*. The third part, *erb*, indicates the templating language you're using, or the handler for this specific template. Templates in Rails can use different templating languages/handlers, but the default in Rails is ERB, hence the *erb* extension.

You could also create a file at `app/views/application/index.html.erb` to provide the view for the index action. This would work because the `ProjectsController` inherits from the `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.

To generate this view, create the `app/views/projects/index.html.erb` file for now. When you run `rake cucumber:ok` again, you get back to what looks like the original error:

```
Given I am on the homepage
  When I follow "New Project"
    no link with title, id or text 'New Project' found
```

Although this looks like the original error, it's actually your first step *truly* passing now. You've defined a homepage for your application by generating a controller, putting an action in it, and creating a view for that action. Now Cucumber (via Capybara) can navigate to it. That's the first step in the first feature passing for your first application, and it's a great first step!

The second step in your `features/creating_projects.feature` file 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". 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 one 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 are telling ERB that whatever the output of this Ruby is, put it on the page. If you only want to evaluate 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.

### 3.2.2 *RESTful routing*

The `new_project_path` method is as yet undefined. If you ran `rake cucumber:ok`, it would still complain of an undefined method or local variable, `'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, as shown in the following listing.

**Listing 3.11** `config/routes.rb`

```
resources :projects
```

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 the Representational State Transfer (REST) standard. In Rails, this means the related controller has seven actions:

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

These seven actions match to 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.

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.

**Table 3.1** RESTful routing matchup

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

To review the routes you've defined, you can run the rake routes command and get output similar to the following.

**Listing 3.12 rake routes output**

```

root          /
  {:controller=>"projects", :action=>"index"}
projects      GET    /projects(..:format)
  {:action=>"index", :controller=>"projects"
  POST    /projects(..:format)
  {:action=>"create", :controller=>"projects"}
new_project   GET    /projects/new(..:format)
  {:action=>"new", :controller=>"projects"}
edit_project  GET    /projects/:id/edit(..:format)
  {:action=>"edit", :controller=>"projects"}
project       GET    /projects/:id(..:format)
  {:action=>"show", :controller=>"projects"}
  PUT    /projects/:id(..:format)
  {:action=>"update", :controller=>"projects"}
  DELETE /projects/:id(..:format)
  {:action=>"delete", :controller=>"projects"}

```

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 would use either `project_path` or `project_url`:

```
project_path(@project)
```

This method takes one argument and generates the path according to this object. You'll see later how you can alter this path to be more user friendly, generating a URL such as `/projects/1-our-project` rather than the impersonal `/projects/1`.

The four paths mentioned earlier match up to the helpers in table 3.2. Running `rake cucumber:ok now` produces a complaint about a missing new action:

```

When I follow "New Project"
  The action 'new' could not be found for ProjectsController

```

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

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

**Table 3.2**  
RESTful routing matchup

**Listing 3.13** app/controllers/projects\_controller.rb

```
class ProjectsController < ApplicationController
  def index
  end

  def new
  end
end
```

Running `rake cucumber:ok` now results in a complaint about a missing new template, just as it did with the index action:

```
When I follow "New Project"
  Missing template projects/new with {
    :handlers=>[:erb, :builder, :sass, :scss],
    :formats=>[:html],
    :locale=>[:en, :en]
  } in view paths "/home/rails3/ticketee/app/views"
```

You can create the file at `app/views/projects/new.html.erb` to make this step pass, although this is a temporary solution. You come back to this file later to add content to it. The third step should now be the failing step, given the second one is passing, so run `rake cucumber:ok` to see if this is really the case:

```
And I fill in "Name" with "TextMate 2"
  cannot fill in, no text field, text area or password field with
  id, name, or label 'Name' found (Capybara::ElementNotFound)
```

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` so you have an object to base the fields on. Change the new to this:

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

The `Project` constant is going to be a class, located at `app/models/project.rb`, thereby making it a *model*. A model is used to retrieve information from the database. Because this model inherits 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:string
```

This syntax is similar to the controller generator's syntax except that you specified you want a model, not a controller. The other difference is that you gave it one further argument comprising a field name and a field type separated by a colon. When the generator runs, it generates not only the model file but also a *migration* containing the code to create this table and the specified field. You can specify as many fields as you like after the model's name.

Migrations are effectively version control for the database. They are 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 are first defined. For example, the code shown in the following listing comes from the migration that was just generated.

**Listing 3.14** `db/migrate/[date]_create_projects.rb`

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

      t.timestamps
    end
  end
end
```

The `change` method is a new addition to Rails 3.1. When you run the migration forward (using `rake db:migrate`), it creates the table. When you roll the migration back (with `rake db:rollback`), it deletes (or drops) the table from the database. In previous versions of Rails, this migration would have been written as follows:

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

      t.timestamps
    end
  end

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

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

In Rails 3.1, you can still use this syntax if you wish, but instead of `self.up` and `self.down`, you simply define the `up` and `down` methods:

```
class CreateProjects < ActiveRecord::Migration
  def up
    # code
  end

  def down
    # code
  end
end
```

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

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

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

This is because ActiveRecord won'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.

The first line tells Active Record 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, and the first argument is the name of that field. The `timestamps` method is special: it creates two fields, the `created_at` and `updated_at` datetime fields, which are by default set to the current time in coordinated universal time (UTC) 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 `rake db:migrate`. This command migrates the database up to the latest migration, which for now is the only migration. If you create a whole slew of migrations at once, this command migrates them in the order they were created.

With this model created and its related migration run, you can now run the feature and have the second step passing once again and the third one failing:

```
When I follow "New Project"
And I fill in "Name" with "TextMate 2"
  cannot fill in, no text field, text area or password field with
  id, name, or label 'Name' found (Capybara::ElementNotFound)
```

Now you are back to the missing field error. To add this field to the new action's view, you use a *partial*. Partials allow you to render dynamic content chunks in your views and are helpful for reducing duplicate code. You put this form in a partial because you must use it later in the `edit` action of your controller. To create a partial for your projects, create a new file called `app/views/projects/_form.html.erb`; the underscore prefix to this file's name indicates that it's a partial. You fill this file with the content from the following listing.

#### Listing 3.15 `app/views/projects/_form.html.erb`

```
<%= form_for(@project) do |f| %>
  <p>
    <%= f.label :name %>
    <%= f.text_field :name %>
  </p>
  <%= f.submit %>
<% end %>
```

So many new things! The `form_for` call allows you to specify the values for the attributes of your new `Project` object when this partial is rendered under the `new` action, and it allows you to edit the values for an existing object when you're on the `edit` action. This works because you'll set `@project` in both of these actions to point to new and existing objects respectively.

The `form_for` method is passed the `@project` object as the first argument, and with this, the helper does 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 (1) whether or not it's a new record and (2) what the class name is.

Determining what `action` attribute the form has (where the form sends to) is dependent on whether the object is a new record. A record is classified as new when it hasn't been saved to the database, and this check is performed internally to Rails using the `persisted?` method, which returns `true` if the record is stored in the database or `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. In this case, it is `/projects`. Because the record is new, the path 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 forms fields. The first element you define is a `label`. `label` tags correspond to their field elements on the page and serve two purposes in the application. First, they give users a larger area to click rather than just the field, radio button, or check box. The second purpose is so you can reference the label's text in the Cucumber story, and Cucumber will know what field to fill in.

**TIP** If you want to customize a label, you can pass a second argument:

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

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

```
<input id="project_name" name="project[name]"
      size="30" type="text">
```

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, a check is made regarding checks whether or not the record is new and sets the text to read "Create Project" if the record is new or "Update Project" if it is not.

The great thing about this partial is that later on, you can use it to implement your `edit` action. To use this partial for the `new` action, put the following code inside the `app/views/projects/new.html.erb`:

```
<h2>New project</h2>
<%= render "form" %>
```

The `render` method in this variation renders the `app/views/projects/_form.html.erb` partial at this location.

Now, running `rake cucumber:ok` once more, you can see that your feature is one step closer to finishing: the field is filled in. How did Capybara know where to find the correct field to fill in? Simple. When you defined the field inside `app/views/projects/_form.html.erb`, you used the syntax shown in the following listing.

#### Listing 3.16 `app/views/projects/_form.html.erb`

```
<%= f.label :name %>
<%= f.text_field :name %>
```

Capybara finds the label containing the "Name" text you ask for in your scenario and fills out the corresponding field. 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 this:

```
When I fill in "project_name" with "TextMate 2"
# or
When I fill in "project[name]" with "TextMate 2"
```

These aren't human friendly ways to find a field, so let's use "Name" instead.

When you run the feature again with `rake cucumber:ok`, you get this error:

```
And I press "Create Project"
  The action 'create' could not be found for ProjectsController
```

The feature now complains of a missing action called `create`. To define this action, you define the `create` method underneath the new method in the `ProjectsController`, as in the following listing.

#### Listing 3.17 `app/controllers/projects_controller.rb`

```
def create
  @project = Project.new(params[:project])
  @project.save
  flash[:notice] = "Project has been created."
  redirect_to @project
end
```

The new method takes the argument `params`, which is available inside controller methods and returns the parameters passed to the action, such as those from the form, as a `HashWithIndifferentAccess` object. These are different from normal `Hash` objects, because you can reference a `String` key by using a matching `Symbol` and vice versa. In this case, the `params` hash is

```
{
  "commit" => "Create Project",
  "action" => "create",
  "project" => {
    "name" => "TextMate 2"
```

```

  },
  "controller" => "projects"
}

```

**TIP** If you'd like to inspect the params hash at any point in time, you can put `puts params` in any action and then run the action either by accessing it through `rails server` or by running a scenario that will run an action containing this line. This outputs to the console the params hash and is equivalent to doing `puts params.inspect`.

All the hashes nested inside this hash are also `HashWithIndifferentAccess` hashes. If you want to get the name key from the `project` hash here, you can use either `{ :name => "TextMate 2" }[:name]`, as in a normal Hash object, or `{ :name => "TextMate 2" }['name']`; you may use either the `String` or the `Symbol` version—it doesn't matter.

The first key in the params hash, `commit`, comes from the Submit button, 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 being `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 field, use `params[:project][:name]`, which calls the `[]` method on `params` to get the value of the `:project` key and then, on the result, calls `[]` again, this time with the `:name` key to get the name of the project passed in.

When `new` receives this `HashWithIndifferentAccess`, it generates a new `Project` object with the attributes based on the parameters passed in. The `Project` object will have a `name` attribute set to the value from `params[:project][:name]`.

You call `@project.save` to save your new `Project` object into the `projects` table.

The `flash` method in your `create` action is a way of passing messages to the next request, and it's also a `HashWithIndifferentAccess`. These messages are stored in the session and are cleared at the completion of the next request. Here you set the `:notice` key 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 takes either an object, as in the `create` action, or a path to redirect to as a string. If an object is given, Rails inspects that object to determine what route it should go to, in this case, `project_path(@project)` because the object has now been saved to the database. This method generates the path of something such as `/projects/:id`, where `:id` is the record `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 `Redirected to /projects/1`, which is the currently nonexistent `show` action.

Upon running `rake cucumber:ok` again, you are told that your app doesn't know about a `show` action:

```

And I press "Create Project"
  The action 'show' could not be found for ProjectsController

```

### 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, you use the `notice` flash key; otherwise you use the `alert` key. By using 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 do not wish to use either `notice` or `alert`, you must specify `flash` as a hash:

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

The `show` action is responsible for displaying a single record's information. To retrieve a record, you need an ID to fetch. You know the URL for this page is going to be something like `/projects/1`, but how do you get the `1` from that URL? Well, when you use resource routing, as you have done, this `1` 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.

Put the code from the following listing into `app/controllers/projects_controller.rb` to do this right now.

#### Listing 3.18 `app/controllers/projects_controller.rb`

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

You pass the `params[:id]` object to `Project.find` here, which 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` cannot find a record matching that ID, it raises an `ActiveRecord::RecordNotFound` exception.

When you run `rake cucumber:ok`, you get an error telling you the `show` action's template is missing:

```
And I press "Create Project"
Missing template projects/show, application/show
with { :handlers=>[:erb, :builder],
      :formats=>[:html],
      :locale=>[:en, :en]}.

Searched in:
* "/Users/ryanbigg/Sites/book/edge-ticketee/app/views"
```

You can create the file `app/views/projects/show.html.erb` with the following content for now:

```
<h2><%= @project.name %></h2>
```

When you run `rake cucumber:ok`, you see this message:

```
And I press "Create Project"
  Then I should see "Project has been created."
  expected #has_content?("Project has been created.")
  to return true, got false
```

This error message shows the `has_content?` method from Capybara, which is used to see if the page has specific content on it. It's checking for "Project has been created." and is not finding it. Therefore, you must put it somewhere, but where? The best location 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 for the flash message.

Here is quite the interesting file:

```
<html>
<head>
  <title>Ticketee</title>
  <%= stylesheet_link_tag "application" %>
  <%= javascript_include_tag "application" %>
  <%= 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 stylesheets from the `app/assets/stylesheets` directory. Using this tag results in the following output:

```
<link href="/assets/application.css"
      media="screen"
      rel="stylesheet"
      type="text/css" />
```

The `/assets` path is served by a gem called `sprockets`. In this case, you're specifying the `/assets/application.css` path, so Sprockets looks for a file called `app/assets/application.css`, which would usually act as a *manifest file* listing the Cascading Style Sheets (CSS) files that need to be included for the application. The current manifest file just provides a stylesheet for the entire application.

For your CSS files, you can use the Sass 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 don'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 its `.css.scss` extension. This final extension tells Sprockets to process the file using Sass before serving it as CSS.

`javascript_include_tag` is for including JavaScript files from the `public/javascripts` directory. When the application string is specified here, Rails loads the application JavaScript manifest file, `app/assets/javascripts/application.js`, which looks like this:

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

This file includes the `jquery.js` and `jquery_ujs.js` files located in the `jquery-rails` gem that your application depends on (see the Gemfile) and compiles them into one super-file called `application.js`, which is referenced by this line in the output of your pages:

```
<script src="/assets/application.js" type="text/javascript"></script>
```

This file is also served through the `sprockets` gem. As with your CSS 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 inside `app/assets/javascripts` with the extension `.js.coffee`, indicating to Sprockets they are to be parsed by a CoffeeScript interpreter first, then served as JavaScript. We use CoffeeScript a little later, in chapter 9.

`csrf_meta_tags` is for protecting your forms from cross-site request forgery (CSRF)<sup>11</sup> attacks. It 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 submitted. If the key is invalid, an `ActionController::InvalidAuthenticityToken` exception occurs.

Later in the file is the 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 on it:

```
<% flash.each do |key, value| %>
  <div class='flash' id='<%= key %>'>
    <%= value %>
  </div>
<% end %>
```

---

<sup>11</sup> <http://en.wikipedia.org/wiki/CSRF>.

This code renders all the flash messages that get defined, regardless of their name. It displays the `flash[:notice]` you set up in the controller. Run `rake cucumber:ok` again and see that not only the last step in your scenario is passing,

```
Then I should see "Project has been created."
```

but the entire scenario is passing!

```
1 scenario (1 passed)
5 steps (5 passed)
```

Yippee! You have just written your first BDD feature 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 any process. Remember when you were learning to drive a car? You didn't drive like Michael Schumacher first off. You learned by doing it slowly and methodically. As you progress, it becomes quicker, as all things do with practice.

### 3.2.3 **Committing changes**

You're at a point where all (just the one for now) your features are running, and points like this are great times to make a commit:

```
git add .
git commit -m "Creation of projects feature complete"
```

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 .
```

**WARNING** 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 allows you to restore the changes using `git stash apply`.

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 `git push` to put it up to GitHub's servers. You don't need to specify the remote or branch because you did that the first time you pushed.

Commit early. Commit often.

### 3.2.4 **Setting a page title**

Before you completely finish working with this story, there is one more thing to point out: the templates 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 an addendum to your Cucumber scenario for it. At the bottom of the scenario in `features/creating_projects.feature`, add the two lines shown in the following listing.

#### Listing 3.19 `features/creating_projects.feature`

```
And I should be on the project page for "TextMate 2"
And I should see "TextMate 2 - Projects - Ticketee"
```

The first line ensures that you're in the `ProjectsController`'s `show` action, or at least it would if you had defined the path to it. If you run `rake cucumber:ok` now, this first step fails:

```
Can't find mapping from "the project page for "TextMate 2"" to a path.
Now, go and add a mapping in ../ticketee/features/support/paths.rb
```

Then it tells you to add a mapping in `features/support/paths.rb`, which is what you should do. You visited this file at the beginning of your feature when you saw that it had this path to the home page defined:

```
when /the home\s?page/
  '/'
```

To define your own path, simply add another when directly underneath this one, as in the following listing.

#### Listing 3.20 `features/support/paths.rb`

```
when /the project page for "([^"]*)"/
  project_path(Project.find_by_name!($1))
```

These two whens together should now look like the following listing.

#### Listing 3.21 `features/support/paths.rb`

```
when /the home\s?page/
  '/'
when /the project page for "([^"]*)"/
  project_path(Project.find_by_name!($1))
```

When the `when` is fully matched to whatever `page_name` is, the part in the quotation marks is captured and stored as the variable `$1`. This is referred to as a *capture group*.

You then use this variable to find the project by the given name so that `project_path` has a record to act on, thereby returning the path to the project. You find this record by using a *dynamic method*. This `find_by_name!` method doesn't exist, and in Ruby when methods don't exist, another method named `method_missing` is called.

The `method_missing` method is passed the name of the method that cannot be found, and any arguments passed to it are passed as additional arguments to

method\_missing. These are then used to construct a real method call and make Ruby act as though the method exists. When you use the *bang* (`find_by_name!` with an exclamation mark as opposed to `find_by_name`) version of this method, ActiveRecord raises an `ActiveRecord::RecordNotFound` exception if the record isn't found. This can prove helpful if, for example, you misspell a project's name when trying to use this step. The exception raised if you don't capitalize the *M* in `TextMate` (thereby making it `Textmate`) is this:

```
Then I should be on the project page for "Textmate 2"
Couldn't find Project with name = Textmate 2 (ActiveRecord::RecordNotFound)
```

If you don't use the bang version of this method, the finder returns `nil`. If `project_path` is passed `nil`, you get a hard-to-debug error:

```
Then I should be on the project page for "Textmate 2"
No route matches {:action => "show", :controller => "projects"}
```

This error claims that no route matches up to the `ProjectsController`'s `show` action, but there actually is one: `/projects/:id`. The difference is that this route requires you to pass through the `id` parameter too; otherwise the error occurs.

To debug something like this, you check and double check what values were being passed where, particularly what `$1` was coming back as, and whether `Project.find_by_name` was returning any record. If you checked those two things, you'd find that `Project.find_by_name` isn't returning what you think it should be returning, and hopefully you'd realize you were passing in the wrong name.

Upon running `rake cucumber:ok`, you now see that a step passes, but the rest of the feature fails:

```
And I should be on the project page for "TextMate 2"
And I should see "TextMate 2 - Projects - Ticketee"
expected #has_content?("TextMate 2 - Projects - Ticketee")
to return true, got false
```

Why are you getting an error seemingly from *RSpec*? Because `Capybara` uses its helpers internally to determine if it can find content. This error therefore means that `Capybara` can't find the content it's looking for. What content? Have a look at the last line of the `backtrace`:

```
features/creating_projects.feature:13:
```

Line 13 of `features/creating_projects.feature` is the `And I should see "TextMate 2 - Projects - Ticketee"` step, which checks for content shown on the page. To make this step pass, you need to define the content it should see. Write the code from the following listing into `app/views/projects/show.html.erb`.

### Listing 3.22 `app/views/projects/show.html.erb`

```
<% @title = "TextMate 2 - Projects - Ticketee" %>
```

Then enter the following code in `app/views/layouts/application.html.erb` where the `title` tag currently is.

**Listing 3.23** `app/views/layouts/application.html.erb`

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

In Ruby, instance variables that aren't set return `nil` as their values. 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, your step passes when you run `rake cucumber:ok`:

```
And I should see "TextMate 2 - Projects - Ticketee"
```

With this scenario now passing, you can change your code and have a solid base to ensure that whatever you change works as you expect. To demonstrate this point, change the code in `show` to use a *helper* instead of setting a variable.

Helpers are methods you can define in the files inside `app/helpers`, and they are made available in your views. Helpers are for extracting the logic from the views, as views should just be about displaying information. Every controller that comes from the controller generator has a corresponding helper, and another helper exists for the entire application. Now open `app/helpers/application_helper.rb` and insert the code from the following listing.

**Listing 3.24** `app/helpers/application_helper.rb`

```
module ApplicationHelper
  def title(*parts)
    unless parts.empty?
      content_for :title do
        (parts << "Ticketee").join(" - ") unless parts.empty?
      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 inside 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 the opposite keyword to `if`: `unless`. If no arguments are passed to the `title` method, `parts` will be empty and therefore `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 of `"title"`. Inside this content block, you join the parts together using a hyphen (`-`), meaning this helper will output something like `"TextMate 2 - Projects - Ticketee"`.

With this, you can slightly alter the title code inside the `app/views/projects/show.html.erb` page to look like `"TextMate 2 - Projects - Ticketee"`, perhaps getting it to show the name of the project rather than `Show` in the title. Before you do that,

though, you should change the line in the feature that checks for the title on the page to the following:

```
And I should see "TextMate 2 - Projects - Ticketee"
```

When you run this feature, it should be broken:

```
expected #has_content?("TextMate 2 - Projects - Ticketee")
to return true, got false
```

Now you can fix it by replacing the line that sets `@title` in your show template with this one:

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

You don't need `Ticketee` here any more because the method puts it in for you. Let's replace the `title` tag line with this:

```
<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. 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 just output the word `Ticketee`, and that becomes the title.

When you run this feature again, it passes:

```
...
And I should see "TextMate 2 - Projects - Ticketee"

1 scenario (1 passed)
7 steps (7 passed)
```

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

```
git add .
git commit -m "Added title functionality for show page"
git push
```

Next up, we look at how to stop users from entering invalid data into your forms.

### 3.2.5 **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 issue. Validations are run just before an object is saved to the database, and if the validations fail, then the object isn't saved. When this happens, you want to tell the user what went wrong, so you write a feature that looks like the following listing.

**Listing 3.25** features/creating\_projects.feature

```
Scenario: Creating a project without a name
  Given I am on the homepage
  When I follow "New Project"
  And I press "Create Project"
  Then I should see "Project has not been created."
  And I should see "Name can't be blank"
```

The first two steps are identical to the ones you placed inside 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. It's easy to extract common code from where it's being duplicated and into a method or a module you 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, before the first scenario, you define a *background*. For Cucumber, backgrounds have a layout identical to scenarios but are executed before *every* scenario inside the feature. Delete the two steps from the top of both of these scenarios so that they now look like the following listing.

**Listing 3.26** features/creating\_projects.feature

```
Scenario: Creating a project
  And I fill in "Name" with "TextMate 2"
  And I press "Create Project"
  Then I should see "Project has been created."
  And I should be on the project page for "TextMate 2"
  And I should see "TextMate 2 - Projects - Ticketee"

Scenario: Creating a project without a name
  And I press "Create Project"
  Then I should see "Project has not been created."
  And I should see "Name can't be blank"
```

Then on the line before the first Scenario, define the Background, as in the next listing.

**Listing 3.27** features/creating\_projects.feature

```
Background:
  Given I am on the homepage
  When I follow "New Project"
```

Now when you run `rake cucumber:ok`, this will fail because it cannot see the error message on the page:

```
Then I should see "Project has not been created."
```

To get this feature to do what you want it to do, add a validation. Validations are defined on the model and typically are run before the data is put into the database, but this can be made optional, as you'll see later. To define a validation ensuring the name attribute is there, open the `app/models/project.rb` file and make it look like the following listing.

**Listing 3.28** `app/models/project.rb`

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

This `validates` method 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, such as the `:uniqueness` key, which, when passed `true` as the value, validates the uniqueness of this field as well. In prior versions of Rails, to do this you would have to use the `validates_presence_of` method instead:

```
class Project < ActiveRecord::Base
  validates_presence_of :name
end
```

This syntax is still supported in Rails 3, but with the new syntax in Rails 3, you can specify multiple validation types for multiple fields on the same line, thereby reducing duplication in your model.

**Beware race conditions with the uniqueness validator**

The `validates_uniqueness_of` validator works by checking to see if a record matching the validation criteria exists already. If this record doesn't exist, the validation will pass.

A problem arises if two connections to the database both make this check at almost exactly the same time. Both connections will claim that a record doesn't exist and therefore will allow a record to be inserted for each connection, resulting in non-unique records.

A way to prevent this is to use a database uniqueness index so the database, not Rails, does the uniqueness validation. For information about how to do this, consult your database's manual.

Although this problem doesn't happen all the time, it *can* happen, so it's something to watch out for.

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. For those of you new to both, IRB stands for *Interactive Ruby*, and it pro-

vides an environment for you to experiment with Ruby without having to create new files. The console prompt looks like this:

```
Loading development environment (Rails 3.1.0.beta)
irb(main):001:0>
```

At this prompt,<sup>12</sup> you can enter any valid Ruby and it'll be evaluated. But 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,
      created_at: nil,
      updated_at: nil>
```

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

If you comment out or remove the validation from inside 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,
      created_at: "2010-05-06 01:00:15",
      updated_at: "2010-05-06 01:00:15">
```

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<sup>13</sup> and checks to see if that object is valid. If it is, Rails sets the `created_at` and `updated_at` attributes to the current time and then saves it 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, and therefore 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
```

---

<sup>12</sup> Although you may see something similar to `ruby-1.9.2-p180 :001 >` too, which is fine.

<sup>13</sup> The first argument for this method is the attributes. If there is no argument passed, then all attributes default to their default values.

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, showing the exception followed by a large stacktrace, which you can safely ignore for now. You are notified which validation failed. To stop it from failing, you must pass in a name attribute, and it will happily return a saved `Project` object:

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

That's how to use `create` to test it in the console, but in your `ProjectsController`, you use the method shown in the following listing instead.

#### Listing 3.29 `app/controllers/projects_controller.rb`

```
@project = Project.new(params[:project])
@project.save
```

`save` doesn't raise an exception if validations fail, as `create!` did, but instead returns `false`. If the validations pass, `save` returns `true`. You can use this to your advantage to show the user an error message when this returns `false` by using it in an `if` statement. Make the `create` action in the `ProjectsController`, as in the following listing.

#### Listing 3.30 `app/controllers/projects_controller.rb`

```
def create
  @project = Project.new(params[:project])
  if @project.save
    flash[:notice] = "Project has been created."
    redirect_to @project
  else
    flash[:alert] = "Project has not been created."
    render :action => "new"
  end
end
```

Now if the `@project` object is valid, then `save` returns `true` and executes everything between the `if` and the `else`. If it isn't valid, then everything between the `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 you run `rake cucumber:ok` here, the second step of your second scenario passes because you now have the `flash[:alert]` set:

```
Then I should see "Project has not been created."
And I should see "Name can't be blank"
  expected #has_content?("Name can't be blank")
  to return true, got false
...
12 steps (1 failed, 11 passed)
```

This scenario passes because of the changes you made to the controller and application layout, where you display all the flash messages. Of course, the third step is now failing. To display error messages in the view, you need to install the `dynamic_form` gem. To install it, add this line to your Gemfile underneath the line for the `coffee-rails` gem:

```
gem 'coffee-rails'
gem 'dynamic_form'
```

Then you must run `bundle install` to install it. Alternatively, you could install it as a plugin by using this command:

```
rails plugin install git://github.com/rails/dynamic_form.git
```

This command executes a `git clone` on the URL passed in and creates a new folder called `vendor/plugins/dynamic_form` in your application, putting the plugin’s code inside of it. Installing it as a plugin would lead to a “polluted” repository, so installing it as a gem is definitely preferred here. Also, when it’s installed as a gem, RubyGems and Bundler provide an exceptionally easy way of keeping it up to date, whereas the plugin architecture in Rails doesn’t. This is a good reason to try to use gems instead of plugins.

The helpful method you’re installing this gem for is the `error_messages` method on the `FormBuilder` object—that is, what `f` represents when you use `form_for` in your `app/views/projects/_form.html.erb` view:

```
<%= form_for(@project) do |f| %>
```

Directly under this `form_for` line, on a new line, insert the following to display the error messages for your object inside the form:

```
<%= f.error_messages %>
```

Error messages for the object represented by your form, the `@project` object, will now be displayed. When you run `rake cucumber:ok`, you get this output:

```
2 scenarios (2 passed)
12 steps (12 passed)
```

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
```

### 3.3 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 alternatives, 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, and then running the installers for these gems so your application is fully configured to use them. For instance, after running `rails g rspec:install`, your application is 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 an introduction to RESTful routing and validations. With this part of your application covered by Cucumber features, you can be notified if it is broken by running `rake cucumber:ok`, a command that runs all the features of the application and lets you know if everything is working or if anything is broken. If something is broken, the feature fails, and then it's up to you to fix it. Without this automated testing, you would have to do it all manually, and that just isn't any fun.

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

# Rails 3 IN ACTION

Ryan Bigg • Yehuda Katz



**R**ails 3 is a full stack, open source web framework powered by Ruby and this book is an introduction to it. Whether you're just starting or you have a few cycles under your belt, you'll appreciate the book's guru's-eye-view of idiomatic Rails programming.

You'll master Rails 3.1 by developing a ticket tracking application that includes RESTful routing, authentication and authorization, state maintenance, file uploads, email, and more. You'll also explore powerful features like designing your own APIs and building a Rails engine. You will see Test Driven Development and Behavior Driven Development in action throughout the book, just like you would in a top Rails shop.

## What's Inside

- Covers Rails 3.1 from the ground up
- Testing and BDD using RSpec and Cucumber
- Working with Rack

It is helpful for readers to have a background in Ruby, but no prior Rails experience is needed.

**Ryan Bigg** is a Rails developer in Sydney, recognized for his prolific and accurate answers on IRC and StackOverflow.

**Yehuda Katz** is a lead developer on SproutCore, known for his contributions to Rails 3, jQuery, Bundler, and Merb.

For access to the book's forum and a free ebook for owners of this book, go to [manning.com/Rails3inAction](http://manning.com/Rails3inAction)

“Takes you on an excellent Rails 3 adventure!”

—Anthony J. Topper  
Penn State Harrisburg

“Conversational and current. A wellspring of information.”

—Jason Rogers, Dell Inc.

“An essential roadmap for the newest features in Rails 3.”

—Greg Vaughn  
Improving Enterprises

“Essential, effective Rails techniques and habits for the modern Rubyist.”

—Thomas Athanas  
Athanas Empire, Inc.

“A holistic book for a holistic framework.”

—Josh Cronmeyer  
ThoughtWorks Studios

ISBN 13: 978-1-935182-27-6  
ISBN 10: 1-935182-27-7



9 781935 182276