Chapter 3

Mostly static pages

In this chapter, we will begin developing the professional-grade sample application that will serve as our example throughout the rest of this tutorial. Although the sample app will eventually have users, microposts, and a full login and authentication framework, we will begin with a seemingly limited topic: the creation of static pages. Despite its apparent simplicity, making static pages is a highly instructive exercise, rich in implications—a perfect start for our nascent application.

Although Rails is designed for making database-backed dynamic websites, it also excels at making the kind of static pages we might create using raw HTML files. In fact, using Rails even for static pages yields a distinct advantage: we can easily add just a *small* amount of dynamic content. In this chapter we'll learn how. Along the way, we'll get our first taste of *automated testing*, which will help us be more confident that our code is correct. Moreover, having a good test suite will allow us to *refactor* our code with confidence, changing its form without changing its function.

3.1   Sample app setup

As in Chapter 2, before getting started we need to create a new Rails project, this time called **sample_app**, as shown in Listing 3.1.[1]

Listing 3.1: Generating a new sample app.

```
$ cd ~/environment
$ rails _6.0.1_ new sample_app
$ cd sample_app/
```

(As in Section 2.1, note that users of the cloud IDE can create this project in the same environment as the applications from the previous two chapters. It is not necessary to create a new environment.)

Note: For convenience, a reference implementation of the sample app is available at GitHub,[2] with a separate branch for each of chapter in the tutorial.

As in Section 2.1, our next step is to use a text editor to update the **Gemfile** with the gems needed by our application. Listing 3.2 is identical to Listing 1.6 and Listing 2.1 apart from the gems in the **test** group, which are needed for the optional advanced testing setup (Section 3.6) and integration testing starting in Section 5.3.4. *Note*: If you would like to install *all* the gems needed for the sample application, you should use the code in Listing 13.75 at this time.

Important note: For all the Gemfiles in this book, you should use the version numbers listed at gemfiles-6th-ed.railstutorial.org instead of the ones listed below (although they should be identical if you are reading this online).

---

[1]If you're using the cloud IDE, it's often useful to use the "Go to Anything" command (under the "Go" menu), which makes it easy to navigate the filesystem by typing in partial filenames. In this context, having the hello, toy, and sample apps present in the same project can be inconvenient due to the many common filenames. For example, when searching for a file called "Gemfile", six possibilities will show up, because each project has matching files called **Gemfile** and **Gemfile.lock**. Thus, you may want to consider removing the first two apps before proceeding, which you can do by navigating to the **environment** directory and running **rm -rf hello_app/ toy_app/** (Table 1.1). (As long as you pushed the corresponding repositories up to GitHub, you can always recover them later.)

[2]https://github.com/mhartl/sample_app_6th_ed

> **Listing 3.2:** A `Gemfile` for the sample app.
>
> ```ruby
> source 'https://rubygems.org'
> git_source(:github) { |repo| "https://github.com/#{repo}.git" }
>
> gem 'rails',      '6.0.1'
> gem 'puma',       '3.12.1'
> gem 'sass-rails', '5.1.0'
> gem 'webpacker',  '4.0.7'
> gem 'turbolinks', '5.2.0'
> gem 'jbuilder',   '2.9.1'
> gem 'bootsnap',   '1.4.4', require: false
>
> group :development, :test do
>   gem 'sqlite3', '1.4.1'
>   gem 'byebug',  '11.0.1', platforms: [:mri, :mingw, :x64_mingw]
> end
>
> group :development do
>   gem 'web-console',          '4.0.1'
>   gem 'listen',               '3.1.5'
>   gem 'spring',               '2.1.0'
>   gem 'spring-watcher-listen', '2.0.1'
> end
>
> group :test do
>   gem 'capybara',                 '3.28.0'
>   gem 'selenium-webdriver',       '3.142.4'
>   gem 'webdrivers',               '4.1.2'
>   gem 'rails-controller-testing', '1.0.4'
>   gem 'minitest',                 '5.11.3'
>   gem 'minitest-reporters',       '1.3.8'
>   gem 'guard',                    '2.15.0'
>   gem 'guard-minitest',           '2.4.6'
> end
>
> group :production do
>   gem 'pg', '1.1.4'
> end
>
> # Windows does not include zoneinfo files, so bundle the tzinfo-data gem
> gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
> ```

As in the previous two chapters, we run `bundle install` to install and include the gems specified in the `Gemfile`, while skipping the installation of production gems using the option `--without production`:[3]

---

[3]It's worth noting that `--without production` is a "remembered option", which means it will be included

```
$ bundle install --without production
```

This arranges to skip the `pg` gem for PostgreSQL in development and use SQLite for development and testing. Heroku recommends against using different databases in development and production, but for the sample application it won't make any difference, and SQLite is *much* easier than PostgreSQL to install and configure locally.[4] In case you've previously installed a version of a gem (such as Rails itself) other than the one specified by the **Gemfile**, it's a good idea to *update* the gems with **bundle update** to make sure the versions match:

```
$ bundle update
```

With that, all we have left is to initialize the Git repository:

```
$ git init
$ git add -A
$ git commit -m "Initialize repository"
```

As with the first application, I suggest updating the README file to be more helpful and descriptive by replacing the default contents of **README.md** with the Markdown shown in Listing 3.3. The README includes instructions for getting started with the application.[5] (We won't actually need to run **rails db:migrate** until Chapter 6, but it does no harm to include it now.)

Note: For convenience, the full reference app README contains additional advanced information not present in Listing 3.3.

---

automatically the next time we run **bundle install**.

[4]Generally speaking, it's a good idea for the development and production environments to match as closely as possible, which includes using the same database, so I recommend eventually learning how to install and configure PostgreSQL in development—but now is not that time. When the time comes, Google "install configure postgresql <your system>" and "rails postgresql setup", and prepare for a challenge. (On the cloud IDE, <your system> is Linux.)

[5]The README also makes reference to a LICENSE file, which I've added by hand to the official reference implementation, but it isn't present by default. You can download a copy from the reference implementation repo if you want it for completeness, but it's not necessary for completing the tutorial.

Listing 3.3: An improved README file for the sample app.
README.md

```
# Ruby on Rails Tutorial sample application

This is the sample application for
[*Ruby on Rails Tutorial:
Learn Web Development with Rails*](https://www.railstutorial.org/)
(6th Edition)
by [Michael Hartl](https://www.michaelhartl.com/).

## License

All source code in the [Ruby on Rails Tutorial](https://www.railstutorial.org/)
is available jointly under the MIT License and the Beerware License. See
[LICENSE.md](LICENSE.md) for details.

## Getting started

To get started with the app, clone the repo and then install the needed gems:

```
$ bundle install --without production
```

Next, migrate the database:

```
$ rails db:migrate
```

Finally, run the test suite to verify that everything is working correctly:

```
$ rails test
```

If the test suite passes, you'll be ready to run the app in a local server:

```
$ rails server
```

For more information, see the
[*Ruby on Rails Tutorial* book](https://www.railstutorial.org/book).
```

Then commit the changes as follows:

```
$ git commit -am "Improve the README"
```

You may recall from Section 1.3.4 that we used the Git command **git commit -a -m "Message"**, with flags for "all changes" (**-a**) and a message (**-m**). As shown in the second command above, Git also lets us roll the two flags into one using **git commit -am "Message"**.

You should also create a new repository at GitHub by following the same steps as in Section 1.3.3 (taking care to make it private as in Figure 3.1), and then push up to the remote repository:

```
$ git remote add origin https://github.com/<username>/sample_app.git
$ git push -u origin master
```

If you're using the cloud IDE, you'll need to prepare the application to be served locally by editing the **development.rb** file as in the previous two chapters (Listing 3.4).

Listing 3.4: Allowing connections to the local web server.
config/environments/development.rb

```
Rails.application.configure do
  .
  .
  .
  # Allow connections to local server.
  config.hosts.clear
end
```

To avoid integration headaches later on, it's also a good idea to deploy the app to Heroku even at this early stage. As in Chapter 1 and Chapter 2, I suggest following the "hello, world!" steps in Listing 3.5 and Listing 3.6. (The main reason for this is that the default Rails page typically breaks at Heroku, which makes it hard to tell if the deployment was successful or not.)
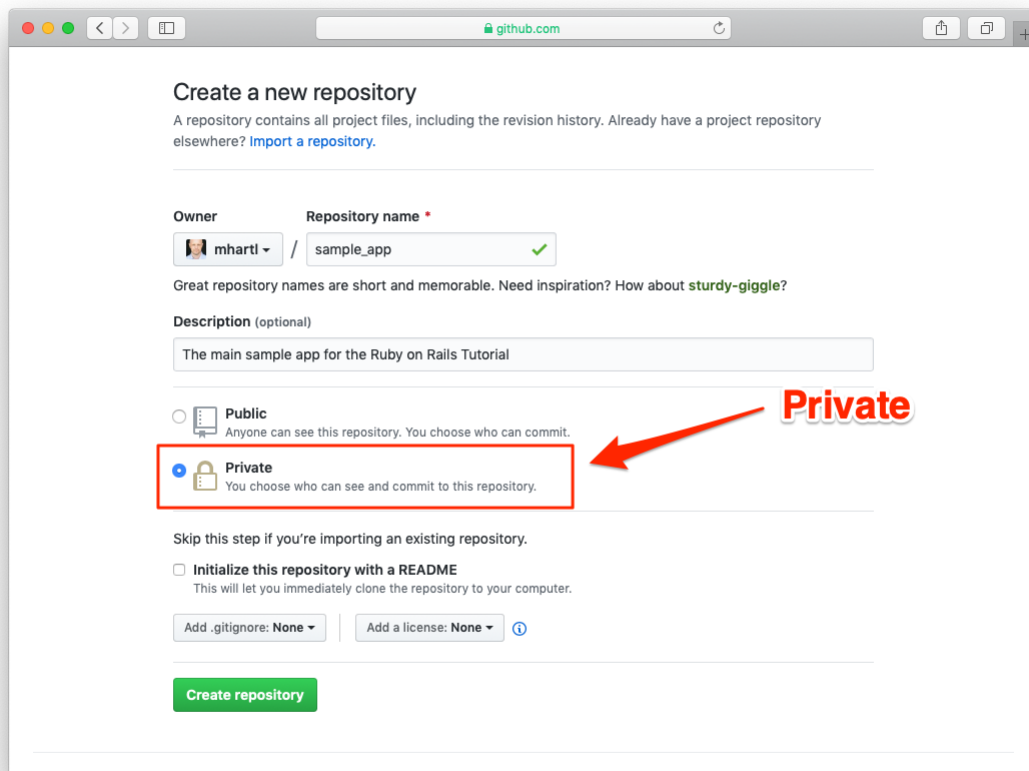
Figure 3.1: Creating the main sample app repository at GitHub.

Listing 3.5: Adding a `hello` action to the Application controller.
app/controllers/application_controller.rb

```ruby
class ApplicationController < ActionController::Base

  def hello
    render html: "hello, world!"
  end
end
```

Listing 3.6: Setting the root route.
config/routes.rb

```ruby
Rails.application.routes.draw do
  root 'application#hello'
end
```

Then commit the changes and push up to GitHub and Heroku:

```
$ git commit -am "Add hello"
$ git push
$ heroku create
$ git push heroku master
```

As in Section 1.4, you may see some warning messages, which you should ignore for now. We'll deal with them in Section 7.5. Apart from the address of the Heroku app, the result should be the same as in Figure 1.31.

As you proceed through the rest of the book, I recommend pushing and deploying the application regularly, which automatically makes remote backups and lets you catch any production errors as soon as possible. If you run into problems at Heroku, make sure to take a look at the production logs to try to diagnose the problem:

```
$ heroku logs           # to see the most recent events
$ heroku logs --tail    # to see events as they happen, Ctrl-C to quit
```

Note: If you do end up using Heroku for a real-life application, be sure to follow the production webserver configuration in Section 7.5.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Confirm that GitHub renders the Markdown for the README in Listing 3.3 as HTML (Figure 3.2).

2. By visiting the root route on the production server, verify that the deployment to Heroku succeeded.

3.2 Static pages

With all the preparation from Section 3.1 finished, we're ready to get started developing the sample application. In this section, we'll take a first step toward making dynamic pages by creating a set of Rails *actions* and *views* containing only static HTML.[6] Rails actions come bundled together inside *controllers* (the C in MVC from Section 1.2.3), which contain sets of actions related by a common purpose. We got a glimpse of controllers in Chapter 2, and will come to a deeper understanding once we explore the REST architecture more fully (starting in Chapter 6). In order to get our bearings, it's helpful to recall the Rails directory structure from Section 1.2 (Figure 1.11). In this section, we'll be working mainly in the **app/controllers** and **app/views** directories.

Recall from Section 1.3.4 that, when using Git, it's a good practice to do our work on a separate topic branch rather than the master branch. If you're using Git for version control, you should run the following command to checkout a topic branch for static pages:

---

[6]Our method for making static pages is probably the simplest, but it's not the only way. The optimal method really depends on your needs; if you expect a *large* number of static pages, using a Static Pages controller can get quite cumbersome, but in our sample app we'll only need a few. If you do need a lot of static pages, take a look at the `high_voltage` gem from thoughtbot.
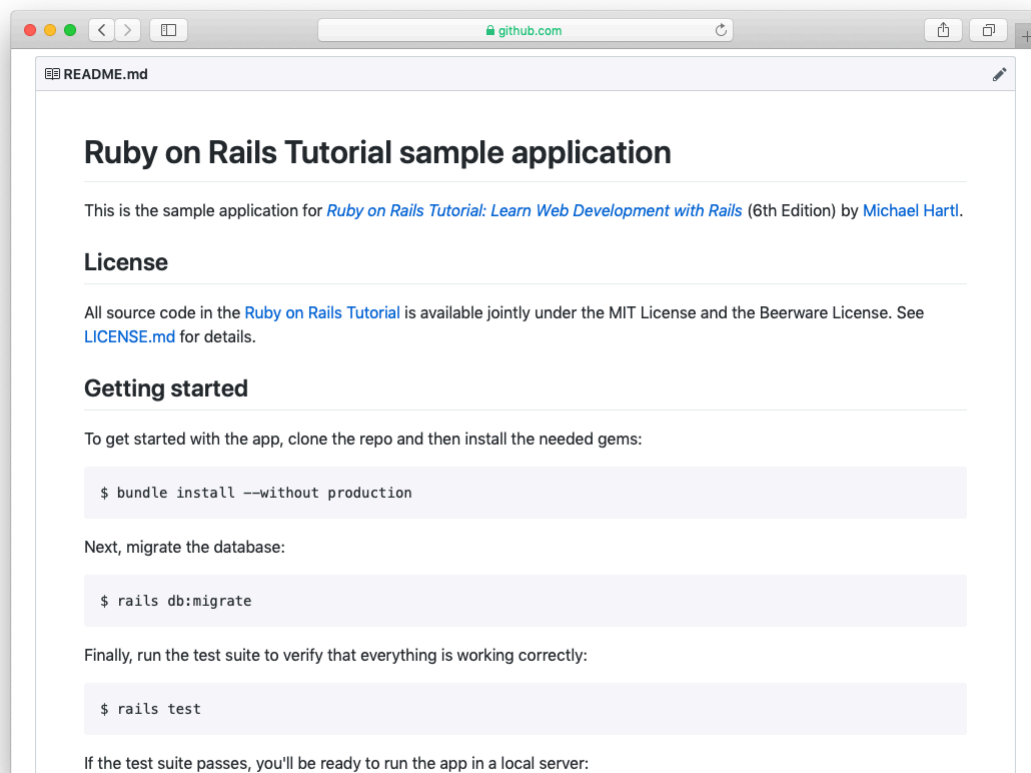
Figure 3.2: The sample app README at GitHub.

```
$ git checkout -b static-pages
```

3.2.1   Generated static pages

To get started with static pages, we'll first generate a controller using the same Rails **generate** script we used in Chapter 2 to generate scaffolding. Since we'll be making a controller to handle static pages, we'll call it the Static Pages controller, designated by the CamelCase name **StaticPages**. We'll also plan to make actions for a Home page, a Help page, and an About page, designated by the lower-case action names **home**, **help**, and **about**. The **generate** script takes an optional list of actions, so we'll include actions for the Home and Help pages directly on the command line, while intentionally leaving off the action for the About page so that we can see how to add it (Section 3.3). The resulting command to generate the Static Pages controller appears in Listing 3.7.

Listing 3.7: Generating a Static Pages controller.

```
$ rails generate controller StaticPages home help
      create  app/controllers/static_pages_controller.rb
       route  get 'static_pages/home'
get 'static_pages/help'
      invoke  erb
      create    app/views/static_pages
      create    app/views/static_pages/home.html.erb
      create    app/views/static_pages/help.html.erb
      invoke  test_unit
      create    test/controllers/static_pages_controller_test.rb
      invoke  helper
      create    app/helpers/static_pages_helper.rb
      invoke    test_unit
      invoke  assets
      invoke    scss
      create      app/assets/stylesheets/static_pages.scss
```

Incidentally, it's worth noting that **rails g** is a shortcut for **rails generate**, which is only one of several shortcuts supported by Rails (Table 3.1). For clarity, this tutorial always uses the full command, but in real life most Rails

Full command	Shortcut
`$ rails server`	`$ rails s`
`$ rails console`	`$ rails c`
`$ rails generate`	`$ rails g`
`$ rails test`	`$ rails t`
`$ bundle install`	`$ bundle`

Table 3.1: Some Rails shortcuts.

developers use one or more of the shortcuts shown in Table 3.1.[7]

Before moving on, if you're using Git it's a good idea to add the files for the Static Pages controller to the remote repository:

```
$ git add -A
$ git commit -m "Add a Static Pages controller"
$ git push -u origin static-pages
```

The final command here arranges to push the **static-pages** topic branch up to GitHub. Subsequent pushes can omit the other arguments and write simply

```
$ git push
```

The commit and push sequence above represents the kind of pattern I would ordinarily follow in real-life development, but for simplicity I'll typically omit such intermediate commits from now on. (When following this tutorial, a good rule of thumb is to make a Git commit at the end of each section.)

In Listing 3.7, note that we have passed the controller name as CamelCase (so called because it resembles the humps of a Bactrian camel), which leads to the creation of a controller file written in snake case, so that a controller called StaticPages yields a file called **static_pages_controller.rb**. This is merely a convention, and in fact using snake case at the command line also works: the command

---

[7]In fact, many Rails developers also add an *alias* (as described in *Learn Enough Text Editor to Be Dangerous*) for the **rails** command, typically shortening it to just **r**. This allows us to run, e.g., a Rails server using the compact command **r s**.

```
$ rails generate controller static_pages ...
```

also generates a controller called **static_pages_controller.rb**. Because Ruby uses CamelCase for class names (Section 4.4), my preference is to refer to controllers using their CamelCase names, but this is a matter of taste. (Since Ruby filenames typically use snake case, the Rails generator converts Camel-Case to snake case using the `underscore` method.)

By the way, if you ever make a mistake when generating code, it's useful to know how to reverse the process. See Box 3.1 for some techniques on how to undo things in Rails.

Box 3.1. Undoing things

Even when you're very careful, things can sometimes go wrong when developing Rails applications. Happily, Rails has some facilities to help you recover.

One common scenario is wanting to undo code generation—for example, when you change your mind on the name of a controller and want to eliminate the generated files. Because Rails creates a substantial number of auxiliary files along with the controller (as seen in Listing 3.7), this isn't as easy as removing the controller file itself; undoing the generation means removing not only the principal generated file, but all the ancillary files as well. (In fact, as we saw in Section 2.2 and Section 2.3, `rails generate` can make automatic edits to the `routes.rb` file, which we also want to undo automatically.) In Rails, this can be accomplished with `rails destroy` followed by the name of the generated element. In particular, these two commands cancel each other out:

```
$ rails generate controller StaticPages home help
$ rails destroy  controller StaticPages home help
```

Similarly, in Chapter 6 we'll generate a *model* as follows:

```
$ rails generate model User name:string email:string
```

This can be undone using

```
$ rails destroy model User
```

(In this case, it turns out we can omit the other command-line arguments. When you get to Chapter 6, see if you can figure out why.)

Another technique related to models involves undoing *migrations*, which we saw briefly in Chapter 2 and will see much more of starting in Chapter 6. Migrations change the state of the database using the command

```
$ rails db:migrate
```

We can undo a single migration step using

```
$ rails db:rollback
```

To go all the way back to the beginning, we can use

```
$ rails db:migrate VERSION=0
```

As you might guess, substituting any other number for 0 migrates to that version number, where the version numbers come from listing the migrations sequentially.

With these techniques in hand, we are well-equipped to recover from the inevitable development snafus.

The Static Pages controller generation in Listing 3.7 automatically updates the routes file (`config/routes.rb`), which we first saw in Section 1.2.4 when we edited the root route for the hello app (Listing 1.11), and which we most recently saw in Listing 3.6. The routes file is responsible for implementing the router (seen in Figure 2.11) that defines the correspondence between URLs and web pages. The routes file is located in the `config` directory, where Rails collects files needed for the application configuration (Figure 3.3).

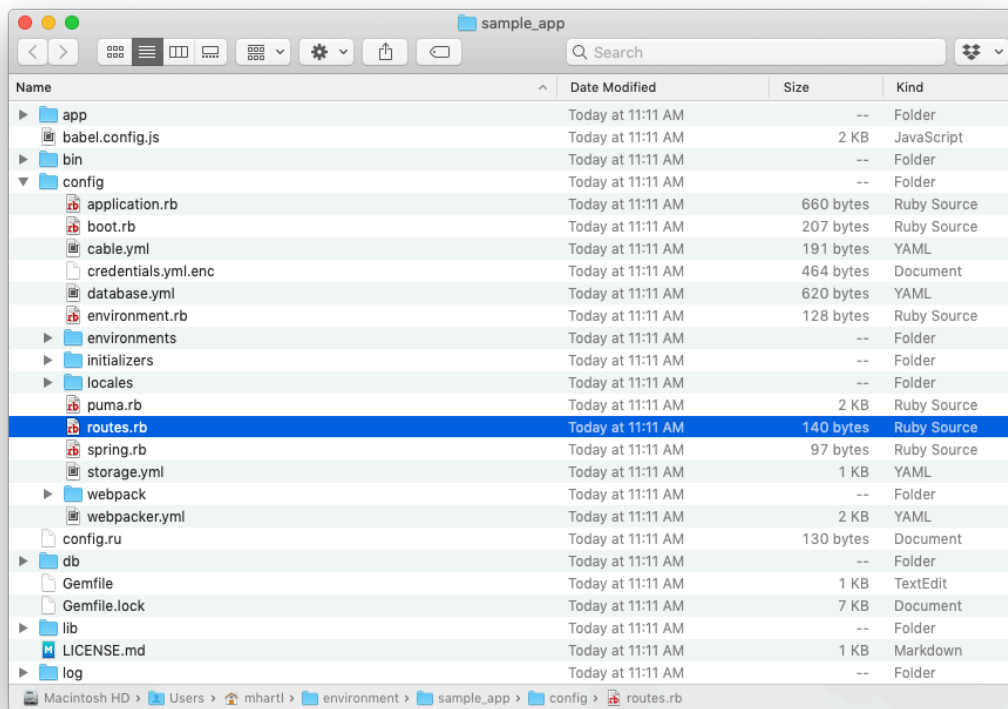Since we included the `home` and `help` actions in Listing 3.7, the routes file

Figure 3.3: Contents of the sample app's **config** directory.

already has a rule for each one, as seen in Listing 3.8.

Listing 3.8: The routes for the **home** and **help** actions in the Static Pages controller.
config/routes.rb

```ruby
Rails.application.routes.draw do
  get  'static_pages/home'
  get  'static_pages/help'
  root 'application#hello'
end
```

Here the rule

```ruby
get 'static_pages/home'
```

maps requests for the URL /static_pages/home to the **home** action in the Static Pages controller. Moreover, by using **get** we arrange for the route to respond to a GET request, which is one of the fundamental *HTTP verbs* supported by the Hypertext Transfer Protocol (Box 3.2). In our case, this means that when we generate a **home** action inside the Static Pages controller we automatically get a page at the address /static_pages/home. To see the result, start a Rails development server as described in Section 1.2.2:

```
$ rails server
```

Then navigate to /static_pages/home (Figure 3.4).

Box 3.2. GET, et cet.

The Hypertext Transfer Protocol (HTTP) defines the basic operations GET, POST, PATCH, and DELETE. These refer to operations between a *client* computer (typically running a web browser such as Chrome, Firefox, or Safari) and a
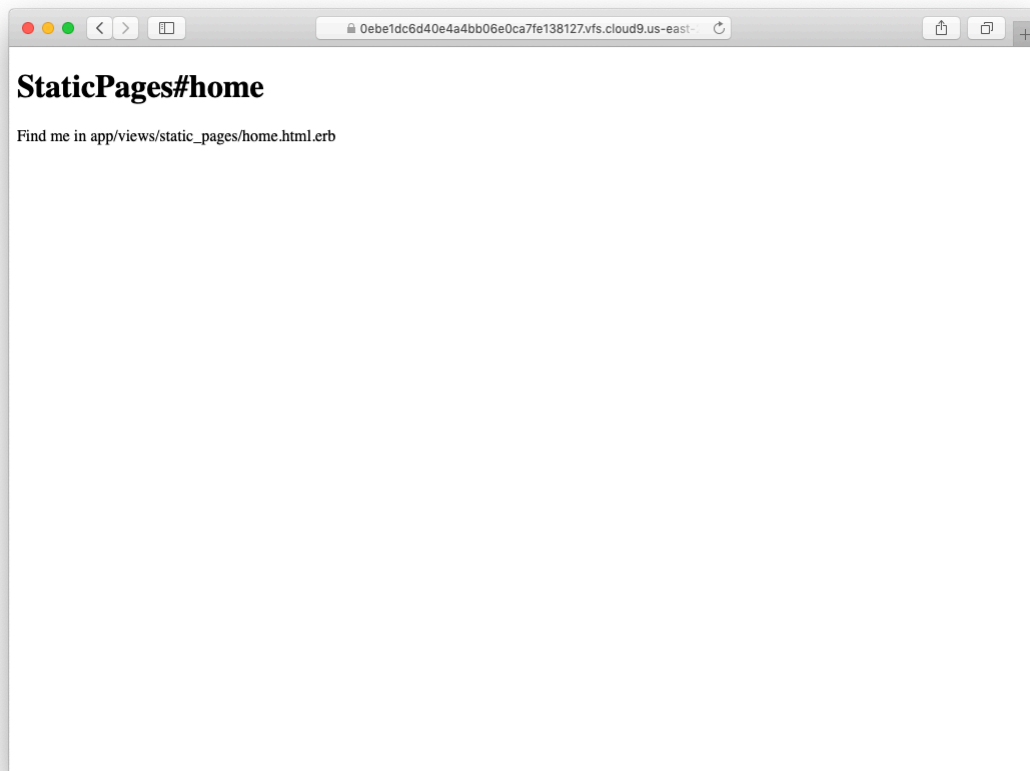

Figure 3.4: The raw home view (/static_pages/home).

server (typically running a webserver such as Apache or Nginx). (It's important to understand that, when developing Rails applications on a local computer, the client and server are the same physical machine, but in general they are different.) An emphasis on HTTP verbs is typical of web frameworks (including Rails) influenced by the *REST architecture*, which we saw briefly in Chapter 2 and will start learning more about in Chapter 7.

`GET` is the most common HTTP operation, used for *reading* data on the web; it just means "get a page", and every time you visit a site like `https://www.google.com/` or `https://www.wikipedia.org/` your browser is submitting a `GET` request. `POST` is the next most common operation; it is the request sent by your browser when you submit a form. In Rails applications, `POST` requests are typically used for *creating* things (although HTTP also allows `POST` to perform updates). For example, the `POST` request sent when you submit a registration form creates a new user on the remote site. The other two verbs, `PATCH` and `DELETE`, are designed for *updating* and *destroying* things on the remote server. These requests are less common than `GET` and `POST` since browsers are incapable of sending them natively, but some web frameworks (including Ruby on Rails) have clever ways of making it *seem* like browsers are issuing such requests. As a result, Rails supports all four of the request types `GET`, `POST`, `PATCH`, and `DELETE`.

To understand where this page comes from, let's start by taking a look at the Static Pages controller in a text editor, which should look something like Listing 3.9. You may note that, unlike the demo Users and Microposts controllers from Chapter 2, the Static Pages controller does not use the standard REST actions. This is normal for a collection of static pages: the REST architecture isn't the best solution to every problem.

Listing 3.9: The Static Pages controller made by Listing 3.7.
app/controllers/static_pages_controller.rb

```ruby
class StaticPagesController < ApplicationController
  def home
```

```
    end

  def help
  end
end
```

We see from the **class** keyword in Listing 3.9 that **static_pages_con-troller.rb** defines a *class*, in this case called **StaticPagesController**. Classes are simply a convenient way to organize *functions* (also called *methods*) like the **home** and **help** actions, which are defined using the **def** keyword. As discussed in Section 2.3.4, the angle bracket **<** indicates that **Static-PagesController** *inherits* from the Rails class **ApplicationController**; as we'll see in a moment, this means that our pages come equipped with a large amount of Rails-specific functionality. (We'll learn more about both classes and inheritance in Section 4.4.)

In the case of the Static Pages controller, both of its methods are initially empty:

```
def home
end

def help
end
```

In plain Ruby, these methods would simply do nothing. In Rails, the situation is different—**StaticPagesController** is a Ruby class, but because it inherits from **ApplicationController** the behavior of its methods is specific to Rails: when visiting the URL /static_pages/home, Rails looks in the Static Pages controller and executes the code in the **home** action, and then renders the *view* (the V in MVC from Section 1.2.3) corresponding to the action. In the present case, the **home** action is empty, so all visiting /static_pages/home does is render the view. So, what does a view look like, and how do we find it?

If you take another look at the output in Listing 3.7, you might be able to guess the correspondence between actions and views: an action like **home** has a corresponding view called **home.html.erb**. We'll learn in Section 3.4 what the **.erb** part means; from the **.html** part you probably won't be surprised that it basically looks like HTML (Listing 3.10).

Listing 3.10: The generated view for the Home page.
app/views/static_pages/home.html.erb

```erb
<h1>StaticPages#home</h1>
<p>Find me in app/views/static_pages/home.html.erb</p>
```

The view for the `help` action is analogous (Listing 3.11).

Listing 3.11: The generated view for the Help page.
app/views/static_pages/help.html.erb

```erb
<h1>StaticPages#help</h1>
<p>Find me in app/views/static_pages/help.html.erb</p>
```

Both of these views are just placeholders: they have a top-level heading (inside the `h1` tag) and a paragraph (`p` tag) with the full path to the corresponding file.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Generate a controller called `Foo` with actions `bar` and `baz`.

2. By applying the techniques described in Box 3.1, destroy the `Foo` controller and its associated actions.

3.2.2 Custom static pages

We'll add some (very slightly) dynamic content starting in Section 3.4, but as they stand the files shown in Listing 3.10 and Listing 3.11 underscore an important point: Rails views can simply contain static HTML. This means we can begin customizing the Home and Help pages even with no knowledge of Rails, as shown in Listing 3.12 and Listing 3.13.

Listing 3.12: Custom HTML for the Home page.
app/views/static_pages/home.html.erb

```
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="https://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

Listing 3.13: Custom HTML for the Help page.
app/views/static_pages/help.html.erb

```
<h1>Help</h1>
<p>
  Get help on the Ruby on Rails Tutorial at the
  <a href="https://www.railstutorial.org/help">Rails Tutorial Help page</a>.
  To get help on this sample app, see the
  <a href="https://www.railstutorial.org/book"><em>Ruby on Rails Tutorial</em>
  book</a>.
</p>
```

The results of Listing 3.12 and Listing 3.13 are shown in Figure 3.5 and Figure 3.6.

3.3   Getting started with testing

Having created and filled in the Home and Help pages for our sample app (Section 3.2.2), now we're going to add an About page as well. When making a change of this nature, it's a good practice to write an *automated test* to verify that the feature is implemented correctly. Developed over the course of building an application, the resulting *test suite* serves as a safety net and as executable documentation of the application source code. When done right, writing tests also allows us to develop *faster* despite requiring extra code, because we'll end up wasting less time trying to track down bugs. This is true only once we get good at writing tests, though, which is one reason it's important to start practicing as early as possible.
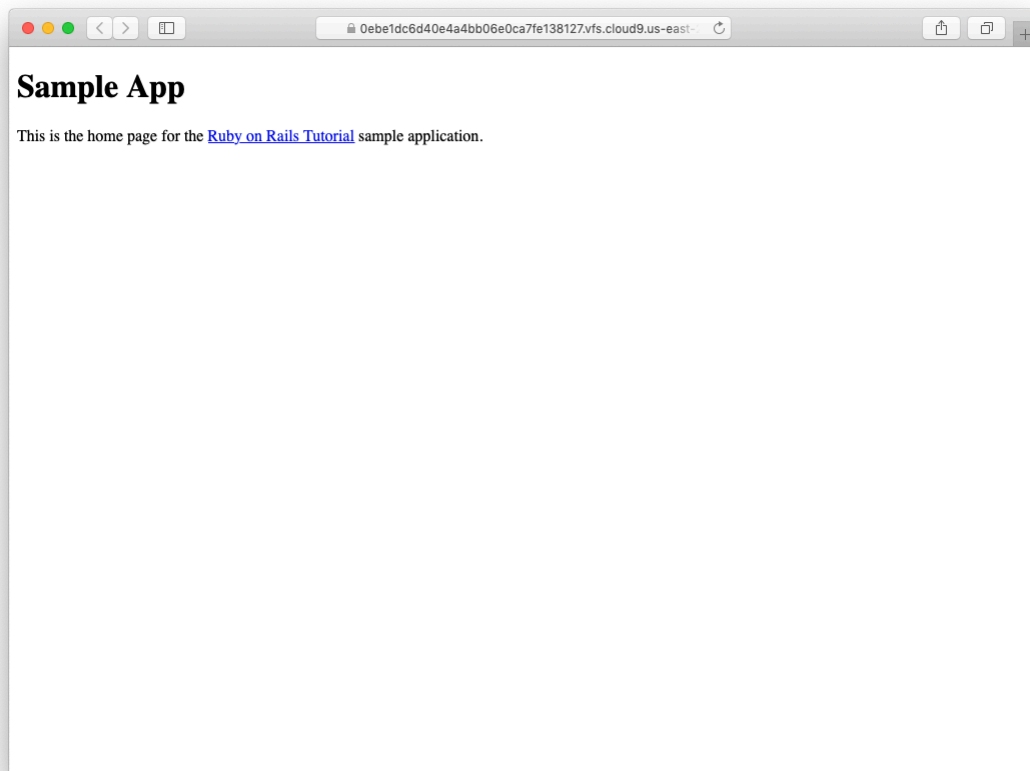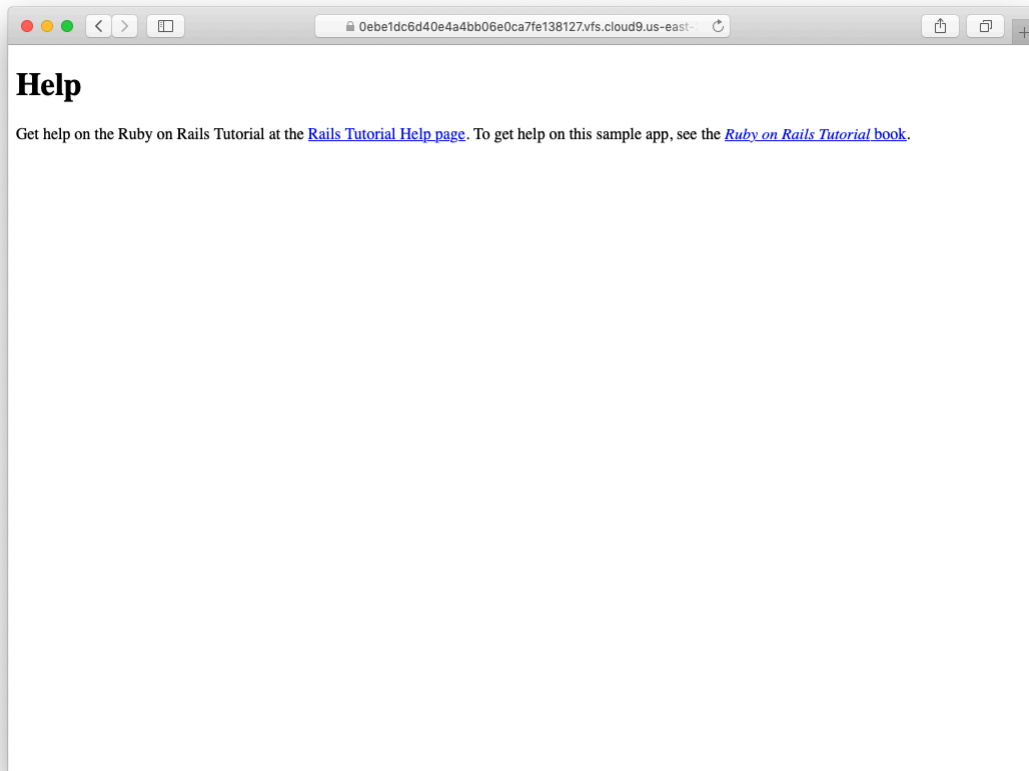
Figure 3.5: A custom Home page.

Figure 3.6: A custom Help page.

Although virtually all Rails developers agree that testing is a good idea, there is a diversity of opinion on the details. There is an especially lively debate over the use of test-driven development (TDD),[8] a testing technique in which the programmer writes failing tests first, and then writes the application code to get the tests to pass. The *Ruby on Rails Tutorial* takes a lightweight, intuitive approach to testing, employing TDD when convenient without being dogmatic about it (Box 3.3).

Box 3.3. When to test

When deciding when and how to test, it's helpful to understand *why* to test. In my view, writing automated tests has three main benefits:

1. Tests protect against *regressions*, where a functioning feature stops working for some reason.

2. Tests allow code to be *refactored* (i.e., changing its form without changing its function) with greater confidence.

3. Tests act as a *client* for the application code, thereby helping determine its design and its interface with other parts of the system.

Although none of the above benefits *require* that tests be written first, there are many circumstances where test-driven development (TDD) is a valuable tool to have in your kit. Deciding when and how to test depends in part on how comfortable you are writing tests; many developers find that, as they get better at writing tests, they are more inclined to write them first. It also depends on how difficult the test is relative to the application code, how precisely the desired features are known, and how likely the feature is to break in the future.

In this context, it's helpful to have a set of guidelines on when we should test first (or test at all). Here are some suggestions based on my own experience:

---

[8]See, e.g., "TDD is dead. Long live testing." by Rails creator David Heinemeier Hansson.

- When a test is especially short or simple compared to the application code it tests, lean toward writing the test first.

- When the desired behavior isn't yet crystal clear, lean toward writing the application code first, then write a test to codify the result.

- Because security is a top priority, err on the side of writing tests of the security model first.

- Whenever a bug is found, write a test to reproduce it and protect against regressions, then write the application code to fix it.

- Lean against writing tests for code (such as detailed HTML structure) likely to change in the future.

- Write tests before refactoring code, focusing on testing error-prone code that's especially likely to break.

In practice, the guidelines above mean that we'll usually write controller and model tests first and integration tests (which test functionality across models, views, and controllers) second. And when we're writing application code that isn't particularly brittle or error-prone, or is likely to change (as is often the case with views), we'll often skip testing altogether.

Our main testing tools will be *controller tests* (starting in this section), *model tests* (starting in Chapter 6), and *integration tests* (starting in Chapter 7). Integration tests are especially powerful, as they allow us to simulate the actions of a user interacting with our application using a web browser. Integration tests will eventually be our primary testing technique, but controller tests give us an easier place to start.

3.3.1   Our first test

Now it's time to add an About page to our application. As we'll see, the test is short and simple, so we'll follow the guidelines from Box 3.3 and write the test first. We'll then use the failing test to drive the writing of the application code.

Getting started with testing can be challenging, requiring extensive knowledge of both Rails and Ruby. At this early stage, writing tests might thus seem hopelessly intimidating. Luckily, Rails has already done the hardest part for us, because **rails generate controller** (Listing 3.7) automatically generated a test file to get us started:

```
$ ls test/controllers/
static_pages_controller_test.rb
```

Let's take a look at it (Listing 3.14).

Listing 3.14: The default tests for the StaticPages controller. GREEN
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

  test "should get home" do
    get static_pages_home_url
    assert_response :success
  end

  test "should get help" do
    get static_pages_help_url
    assert_response :success
  end
end
```

It's not important at this point to understand the syntax in Listing 3.14 in detail, but we can see that there are two tests, one for each controller action we included on the command line in Listing 3.7. Each test simply gets a URL and verifies (via an *assertion*) that the result is a success. Here the use of **get** indicates that our tests expect the Home and Help pages to be ordinary web pages,

accessed using a `GET` request (Box 3.2). The response `:success` is an abstract representation of the underlying HTTP status code (in this case, 200 OK). In other words, a test like

```
test "should get home" do
  get static_pages_home_url
  assert_response :success
end
```

says "Let's test the Home page by issuing a `GET` request to the Static Pages **home** URL and then making sure we receive a 'success' status code in response."

To begin our testing cycle, we need to run our test suite to verify that the tests currently pass. We can do this with the **rails** command as follows:

Listing 3.15: GREEN

```
$ rails db:migrate    # Necessary on some systems
$ rails test
2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

As required, initially our test suite is passing (GREEN). (Some systems won't actually display the color green unless you add the minitest reporters from the optional Section 3.6.1, but the terminology is common even when literal colors aren't involved.) Note that here and throughout this tutorial, I'll generally omit some lines from the test output in order to highlight only the most imporant parts.

By the way, on some systems you may see generated files of the form

```
/db/test.sqlite3-0
```

show up in the **db** directory. To prevent these generated files from being added to the repository, I suggest adding a rule to the **.gitignore** file (Section 1.3.1) to ignore them, as shown in Listing 3.16.

Listing 3.16: Ignoring generated database files.
.gitignore

```
.
.
.
# Ignore db test files.
db/test.*
```

3.3.2   Red

As noted in Box 3.3, test-driven development involves writing a failing test first, writing the application code needed to get it to pass, and then refactoring the code if necessary. Because many testing tools represent failing tests with the color red and passing tests with the color green, this sequence is sometimes known as the "Red, Green, Refactor" cycle. In this section, we'll complete the first step in this cycle, getting to RED by writing a failing test. Then we'll get to GREEN in Section 3.3.3, and refactor in Section 3.4.3.[9]

Our first step is to write a failing test for the About page. By following the models from Listing 3.14, can you guess what it should be? The answer appears in Listing 3.17.

Listing 3.17: A test for the About page. RED
test/controllers/static_pages_controller_test.rb

```ruby
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

  test "should get home" do
    get static_pages_home_url
    assert_response :success
  end

  test "should get help" do
    get static_pages_help_url
    assert_response :success
```

---

[9]On some systems, **rails test** shows red when the tests fail but doesn't show green when the tests pass. To arrange for a true Red–Green cycle, see Section 3.6.1.

```ruby
  end

  test "should get about" do
    get static_pages_about_url
    assert_response :success
  end
end
```

We see from the highlighted lines in Listing 3.17 that the test for the About page is the same as the Home and Help tests with the word "about" in place of "home" or "help".

As required, the test initially fails:

Listing 3.18: RED

```
$ rails test
3 tests, 2 assertions, 0 failures, 1 errors, 0 skips
```

3.3.3   Green

Now that we have a failing test (RED), we'll use the failing test's error messages to guide us to a passing test (GREEN), thereby implementing a working About page.

We can get started by examining the error message output by the failing test:

Listing 3.19: RED

```
$ rails test
NameError: undefined local variable or method `static_pages_about_url'
```

The error message here says that the Rails code for the About page URL is undefined, which is a hint that we need to add a line to the routes file. We can accomplish this by following the pattern in Listing 3.8, as shown in Listing 3.20.

Listing 3.20: Adding the **about** route. RED
config/routes.rb

```
Rails.application.routes.draw do
  get  'static_pages/home'
  get  'static_pages/help'
  get  'static_pages/about'
  root 'application#hello'
end
```

The highlighted line in Listing 3.20 tells Rails to route a GET request for the
URL /static_pages/about to the **about** action in the Static Pages controller. This
automatically creates a helper called

```
static_pages_about_url
```

Running our test suite again, we see that it is still RED, but now the error
message has changed:

Listing 3.21: RED

```
$ rails test
AbstractController::ActionNotFound:
The action 'about' could not be found for StaticPagesController
```

The error message now indicates a missing **about** action in the Static Pages
controller, which we can add by following the model provided by **home** and
help in Listing 3.9, as shown in Listing 3.22.

Listing 3.22: The Static Pages controller with added **about** action. RED
app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController

  def home
  end
```

```
  def help
  end

  def about
  end
end
```

As before, our test suite is still RED, but the error message has changed again:

```
$ rails test
ActionController::UnknownFormat: StaticPagesController#about is missing
a template for this request format and variant.
```

This indicates a missing template, which in the context of Rails is essentially the same thing as a view. As described in Section 3.2.1, an action called **home** is associated with a view called **home.html.erb** located in the **app/views/-static_pages** directory, which means that we need to create a new file called **about.html.erb** in the same directory.

The way to create a file varies by system setup, but most text editors will let you control-click inside the directory where you want to create the file to bring up a menu with a "New File" menu item. Alternately, you can use the File menu to create a new file and then pick the proper directory when saving it. Finally, you can use my favorite trick by applying the Unix touch command as follows:

```
$ touch app/views/static_pages/about.html.erb
```

As mentioned in *Learn Enough Command Line to Be Dangerous*, **touch** is designed to update the modification timestamp of a file or directory without otherwise affecting it, but as a side-effect it creates a new (blank) file if one doesn't already exist. (If using the cloud IDE, you may have to refresh the file tree as described in Section 1.2.1. This is a good example of technical sophistication (Box 1.2).)

Once you've created the **about.html.erb** file in the right directory, you should fill it with the contents shown in Listing 3.23.

Listing 3.23: Code for the About page. GREEN
app/views/static_pages/about.html.erb

```
<h1>About</h1>
<p>
  The <a href="https://www.railstutorial.org/"><em>Ruby on Rails
  Tutorial</em></a>, part of the
  <a href="https://www.learnenough.com/">Learn Enough</a> family of
  tutorials, is a
  <a href="https://www.railstutorial.org/book">book</a> and
  <a href="https://screencasts.railstutorial.org/">screencast series</a>
  to teach web development with
  <a href="https://rubyonrails.org/">Ruby on Rails</a>.
  This is the sample app for the tutorial.
</p>
```

At this point, running **`rails test`** should get us back to GREEN:

Listing 3.24: GREEN

```
$ rails test
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

Of course, it's never a bad idea to take a look at the page in a browser to make sure our tests aren't leading us astray (Figure 3.7).

3.3.4   Refactor

Now that we've gotten to GREEN, we are free to refactor our code with confidence. When developing an application, often code will start to "smell", meaning that it gets ugly, bloated, or filled with repetition. The computer doesn't care what the code looks like, of course, but humans do, so it is important to keep the code base clean by refactoring frequently. Although our sample app is a little too small to refactor right now, code smell seeps in at every crack, and we'll get started refactoring in Section 3.4.3.
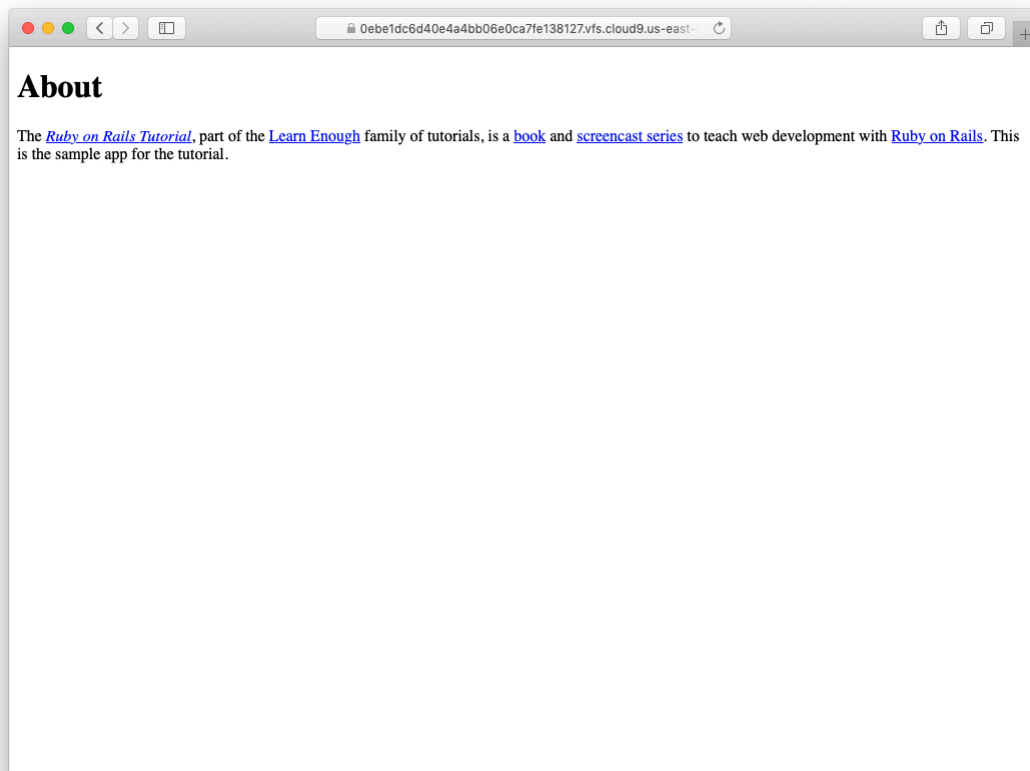
Figure 3.7: The new About page (/static_pages/about).

Page	URL	Base title	Variable title
Home	/static_pages/home	`"Ruby on Rails Tutorial Sample App"`	`"Home"`
Help	/static_pages/help	`"Ruby on Rails Tutorial Sample App"`	`"Help"`
About	/static_pages/about	`"Ruby on Rails Tutorial Sample App"`	`"About"`

Table 3.2: The (mostly) static pages for the sample app.

3.4   Slightly dynamic pages

Now that we've created the actions and views for some static pages, we'll make them *slightly* dynamic by adding some content that changes on a per-page basis: we'll have the title of each page change to reflect its content. Whether a changing title represents *truly* dynamic content is debatable, but in any case it lays the necessary foundation for unambiguously dynamic content in Chapter 7.

Our plan is to edit the Home, Help, and About pages to make page titles that change on each page. This will involve using the **`<title>`** tag in our page views. Most browsers display the contents of the title tag at the top of the browser window, and it is also important for search-engine optimization. We'll be using the full "Red, Green, Refactor" cycle: first by adding simple tests for our page titles (RED), then by adding titles to each of our three pages (GREEN), and finally using a *layout* file to eliminate duplication (Refactor). By the end of this section, all three of our static pages will have titles of the form "<page name> | Ruby on Rails Tutorial Sample App", where the first part of the title will vary depending on the page (Table 3.2).

The **`rails new`** command (Listing 3.1) creates a layout file by default, but it's instructive to ignore it initially, which we can do by changing its name:

```
$ mv app/views/layouts/application.html.erb layout_file   # temporary change
```

You wouldn't normally do this in a real application, but it's easier to understand the purpose of the layout file if we start by disabling it.

3.4.1   Testing titles (Red)

To add page titles, we need to learn (or review) the structure of a typical web page, which takes the form shown in Listing 3.25. (This is covered in much more depth in *Learn Enough HTML to Be Dangerous*.)

Listing 3.25: The HTML structure of a typical web page.

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Greeting</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

The structure in Listing 3.25 includes a *document type*, or doctype, declaration at the top to tell browsers which version of HTML we're using (in this case, HTML5);[10] a **head** section, in this case with "Greeting" inside a **title** tag; and a **body** section, in this case with "Hello, world!" inside a **p** (paragraph) tag. (The indentation is optional—HTML is not sensitive to whitespace, and ignores both tabs and spaces—but it makes the document's structure easier to see.)

We'll write simple tests for each of the titles in Table 3.2 by combining the tests in Listing 3.17 with the **assert_select** method, which lets us test for the presence of a particular HTML tag (sometimes called a "selector", hence the name):[11]

```ruby
assert_select "title", "Home | Ruby on Rails Tutorial Sample App"
```

In particular, the code above checks for the presence of a **<title>** tag containing the string "Home | Ruby on Rails Tutorial Sample App". Applying this

---

[10]HTML changes with time; by explicitly making a doctype declaration we make it likelier that browsers will render our pages properly in the future. The simple doctype **<!DOCTYPE html>** is characteristic of the latest HTML standard, HTML5.

[11]For a list of common minitest assertions, see the table of available assertions in the Rails Guides testing article.

idea to all three static pages gives the tests shown in Listing 3.26.

Listing 3.26: The Static Pages controller test with title tests. RED
test/controllers/static_pages_controller_test.rb

```ruby
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

  test "should get home" do
    get static_pages_home_url
    assert_response :success
    assert_select "title", "Home | Ruby on Rails Tutorial Sample App"
  end

  test "should get help" do
    get static_pages_help_url
    assert_response :success
    assert_select "title", "Help | Ruby on Rails Tutorial Sample App"
  end

  test "should get about" do
    get static_pages_about_url
    assert_response :success
    assert_select "title", "About | Ruby on Rails Tutorial Sample App"
  end
end
```

With the tests from Listing 3.26 in place, you should verify that the test suite is currently RED:

Listing 3.27: RED

```
$ rails test
3 tests, 6 assertions, 3 failures, 0 errors, 0 skips
```

3.4.2 Adding page titles (Green)

Now we'll add a title to each page, getting the tests from Section 3.4.1 to pass in the process. Applying the basic HTML structure from Listing 3.25 to the custom Home page from Listing 3.12 yields Listing 3.28.

Listing 3.28: The view for the Home page with full HTML structure. RED
app/views/static_pages/home.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Home | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Sample App</h1>
    <p>
      This is the home page for the
      <a href="https://www.railstutorial.org/">Ruby on Rails Tutorial</a>
      sample application.
    </p>
  </body>
</html>
```

The corresponding web page appears in Figure 3.8. Note that the browser used in the screenshots (Safari) displays the page title only if you include an additional tab, which explains the second tab shown in Figure 3.8.

Following this model for the Help page (Listing 3.13) and the About page (Listing 3.23) yields the code in Listing 3.29 and Listing 3.30.

Listing 3.29: The view for the Help page with full HTML structure. RED
app/views/static_pages/help.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Help | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Help</h1>
    <p>
      Get help on the Ruby on Rails Tutorial at the
      <a href="https://www.railstutorial.org/help">Rails Tutorial help
      page</a>.
      To get help on this sample app, see the
      <a href="https://www.railstutorial.org/book"><em>Ruby on Rails
      Tutorial</em> book</a>.
    </p>
  </body>
</html>
```
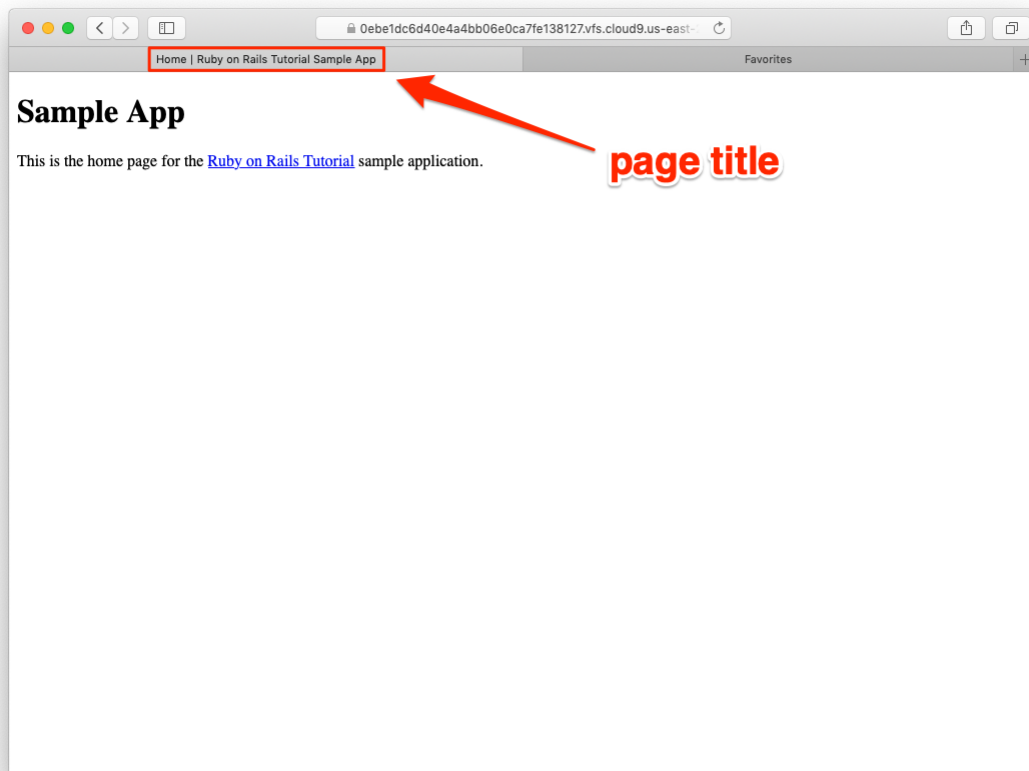
Figure 3.8: The Home page with a title.

Listing 3.30: The view for the About page with full HTML structure. GREEN
app/views/static_pages/about.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>About | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>About</h1>
    <p>
      The <a href="https://www.railstutorial.org/"><em>Ruby on Rails
      Tutorial</em></a>, part of the
      <a href="https://www.learnenough.com/">Learn Enough</a> family of
      tutorials, is a
      <a href="https://www.railstutorial.org/book">book</a> and
      <a href="https://screencasts.railstutorial.org/">screencast series</a>
      to teach web development with
      <a href="https://rubyonrails.org/">Ruby on Rails</a>.
      This is the sample app for the tutorial.
    </p>
  </body>
</html>
```

At this point, the test suite should be back to GREEN:

Listing 3.31: GREEN

```
$ rails test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

Beginning in this section, we'll start making modifications to the applications in the exercises that won't generally be reflected in future code listings. The reason is so that the text makes sense to readers who don't complete the exercises, but as a result your code will diverge from the main text if you *do*

solve them. Learning to resolve small discrepancies like this is an excellent example of technical sophistication (Box 1.2).

1. You may have noticed some repetition in the Static Pages controller test (Listing 3.26). In particular, the base title, "Ruby on Rails Tutorial Sample App", is the same for every title test. Using the special function **setup**, which is automatically run before every test, verify that the tests in Listing 3.32 are still GREEN. (Listing 3.32 uses an *instance variable*, seen briefly in Section 2.2.2 and covered further in Section 4.4.5, combined with *string interpolation*, which is covered further in Section 4.2.1.)

Listing 3.32: The Static Pages controller test with a base title. GREEN
test/controllers/static_pages_controller_test.rb

```ruby
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

  def setup
    @base_title = "Ruby on Rails Tutorial Sample App"
  end

  test "should get home" do
    get static_pages_home_url
    assert_response :success
    assert_select "title", "Home | #{@base_title}"
  end

  test "should get help" do
    get static_pages_help_url
    assert_response :success
    assert_select "title", "Help | #{@base_title}"
  end

  test "should get about" do
    get static_pages_about_url
    assert_response :success
    assert_select "title", "About | #{@base_title}"
  end
end
```

3.4.3 Layouts and embedded Ruby (Refactor)

We've achieved a lot already in this section, generating three valid pages using Rails controllers and actions, but they are purely static HTML and hence don't show off the power of Rails. Moreover, they suffer from terrible duplication:

- The page titles are almost (but not quite) exactly the same.

- "Ruby on Rails Tutorial Sample App" is common to all three titles.

- The entire HTML skeleton structure is repeated on each page.

This repeated code is a violation of the important "Don't Repeat Yourself" (DRY) principle; in this section we'll "DRY out our code" by removing the repetition. At the end, we'll re-run the tests from Section 3.4.2 to verify that the titles are still correct.

Paradoxically, we'll take the first step toward eliminating duplication by first adding some more: we'll make the titles of the pages, which are currently quite similar, match *exactly*. This will make it much simpler to remove all the repetition at a stroke.

The technique involves using *embedded Ruby* in our views. Since the Home, Help, and About page titles have a variable component, we'll use a special Rails function called **provide** to set a different title on each page. We can see how this works by replacing the literal title "Home" in the **home.html.erb** view with the code in Listing 3.33.

Listing 3.33: The view for the Home page with an embedded Ruby title. GREEN
app/views/static_pages/home.html.erb

```erb
<% provide(:title, "Home") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Sample App</h1>
    <p>
```

```
      This is the home page for the
      <a href="https://www.railstutorial.org/">Ruby on Rails Tutorial</a>
      sample application.
    </p>
  </body>
</html>
```

Listing 3.33 is our first example of embedded Ruby, also called *ERb* (or ERB).
(Now you know why HTML views have the file extension **.html.erb**.) ERb
is the primary template system for including dynamic content in web pages.[12]
The code

```
<% provide(:title, "Home") %>
```

indicates using **<% ... %>** that Rails should call the **provide** function and
associate the string **"Home"** with the label **:title**.[13] Then, in the title, we
use the closely related notation **<%= ... %>** to insert the title into the template
using Ruby's **yield** function:[14]

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

(The distinction between the two types of embedded Ruby is that **<% ... %>**
executes the code inside, while **<%= ... %>** executes it *and inserts* the result
into the template.) The resulting page is exactly the same as before, only now
the variable part of the title is generated dynamically by ERb.

We can verify that all this works by running the tests from Section 3.4.2 and
see that they are still GREEN:

---

[12]There is a second popular template system called Haml (note: not "HAML"), which I personally love, but it's
not *quite* standard enough for use in an introductory tutorial.

[13]Experienced Rails developers might have expected the use of **content_for** at this point, but it doesn't work
well with the asset pipeline. The **provide** function is its replacement.

[14]If you've studied Ruby before, you might suspect that Rails is *yielding* the contents to a block, and your
suspicion would be correct. But you don't need to know this to develop applications with Rails.

Listing 3.34: GREEN

```
$ rails test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

Then we can make the corresponding replacements for the Help and About pages (Listing 3.35 and Listing 3.36).

Listing 3.35: The view for the Help page with an embedded Ruby title. GREEN
app/views/static_pages/help.html.erb

```erb
<% provide(:title, "Help") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Help</h1>
    <p>
      Get help on the Ruby on Rails Tutorial at the
      <a href="https://www.railstutorial.org/help">Rails Tutorial help
      section</a>.
      To get help on this sample app, see the
      <a href="https://www.railstutorial.org/book"><em>Ruby on Rails
      Tutorial</em> book</a>.
    </p>
  </body>
</html>
```

Listing 3.36: The view for the About page with an embedded Ruby title.
GREEN
app/views/static_pages/about.html.erb

```erb
<% provide(:title, "About") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>About</h1>
    <p>
```

```
      The <a href="https://www.railstutorial.org/"><em>Ruby on Rails
      Tutorial</em></a>, part of the
      <a href="https://www.learnenough.com/">Learn Enough</a> family of
      tutorials, is a
      <a href="https://www.railstutorial.org/book">book</a> and
      <a href="https://screencasts.railstutorial.org/">screencast series</a>
      to teach web development with
      <a href="https://rubyonrails.org/">Ruby on Rails</a>.
      This is the sample app for the tutorial.
    </p>
  </body>
</html>
```

Now that we've replaced the variable part of the page titles with ERb, each of our pages looks something like this:

```
<% provide(:title, "Page Title") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    Contents
  </body>
</html>
```

In other words, all the pages are identical in structure, including the contents of the title tag, with the sole exception of the material inside the **body** tag.

In order to factor out this common structure, Rails comes with a special *layout* file called **application.html.erb**, which we renamed in the beginning of this section (Section 3.4) and which we'll now restore:

```
$ mv layout_file app/views/layouts/application.html.erb
```

To get the layout to work, we have to replace the default title with the embedded Ruby from the examples above:

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

The resulting layout appears in Listing 3.37.

Listing 3.37: The sample application site layout. GREEN
app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>

    <%= stylesheet_link_tag 'application', media: 'all',
                                            'data-turbolinks-track': 'reload' %>
    <%= javascript_pack_tag 'application', 'data-turbolinks-track': 'reload' %>
  </head>

  <body>
    <%= yield %>
  </body>
</html>
```

Note here the special line

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

This code is responsible for inserting the contents of each page into the layout. It's not important to know exactly how this works; what matters is that using this layout ensures that, for example, visiting the page /static_pages/home converts the contents of **home.html.erb** to HTML and then inserts it in place of **<%= yield %>**.

It's also worth noting that the default Rails layout includes several additional lines:

```
<%= csrf_meta_tags %>
<%= csp_meta_tag %>
<%= stylesheet_link_tag ... %>
<%= javascript_pack_tag "application", ... %>
```

This code arranges to include the application stylesheet and JavaScript, which are part of the asset pipeline (Section 5.2.1), together with the Rails method **csp_meta_tag**, which implements Content Security Policy (CSP) to mitigate cross-site scripting (XSS) attacks, and **csrf_meta_tags**, which mitigates cross-site request forgery (CSRF) attacks. (One huge advantage of using a mature framework like Rails is that it worries about such things so that we don't have to.)

Even though the tests are passing, there one detail left to deal with: the views in Listing 3.33, Listing 3.35, and Listing 3.36 are still filled with all the HTML structure included in the layout. Since it's redundant (and indeed leads to invalid HTML markup) we should remove it and leave only the interior contents. The resulting cleaned-up views appear in Listing 3.38, Listing 3.39, and Listing 3.40.

Listing 3.38: The Home page with HTML structure removed. GREEN
app/views/static_pages/home.html.erb

```erb
<% provide(:title, "Home") %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="https://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

Listing 3.39: The Help page with HTML structure removed. GREEN
app/views/static_pages/help.html.erb

```erb
<% provide(:title, "Help") %>
<h1>Help</h1>
<p>
  Get help on the Ruby on Rails Tutorial at the
```

```
  <a href="https://www.railstutorial.org/help">Rails Tutorial Help page</a>.
  To get help on this sample app, see the
  <a href="https://www.railstutorial.org/book"><em>Ruby on Rails Tutorial</em>
  book</a>.
</p>
```

Listing 3.40: The About page with HTML structure removed. GREEN
app/views/static_pages/about.html.erb

```
<% provide(:title, "About") %>
<h1>About</h1>
<p>
  The <a href="https://www.railstutorial.org/"><em>Ruby on Rails
  Tutorial</em></a>, part of the
  <a href="https://www.learnenough.com/">Learn Enough</a> family of
  tutorials, is a
  <a href="https://www.railstutorial.org/book">book</a> and
  <a href="https://screencasts.railstutorial.org/">screencast series</a>
  to teach web development with
  <a href="https://rubyonrails.org/">Ruby on Rails</a>.
  This is the sample app for the tutorial.
</p>
```

With these views defined, the Home, Help, and About pages are the same as before, but they have much less duplication.

Experience shows that even fairly simple refactoring is error-prone and can easily go awry. This is one reason why having a good test suite is so valuable. Rather than double-checking every page for correctness—a procedure that isn't too hard early on but rapidly becomes unwieldy as an application grows—we can simply verify that the test suite is still GREEN:

Listing 3.41: GREEN

```
$ rails test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

This isn't a *proof* that our code is still correct, but it greatly increases the probability, thereby providing a safety net to protect us against future bugs.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

   To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Make a Contact page for the sample app.[15] Following the model in Listing 3.17, first write a test for the existence of a page at the URL /static_-pages/contact by testing for the title "Contact | Ruby on Rails Tutorial Sample App". Get your test to pass by following the same steps as when making the About page in Section 3.3.3, including filling the Contact page with the content from Listing 3.42.

Listing 3.42: Code for a proposed Contact page.
app/views/static_pages/contact.html.erb

```erb
<% provide(:title, "Contact") %>
<h1>Contact</h1>
<p>
  Contact the Ruby on Rails Tutorial about the sample app at the
  <a href="https://www.railstutorial.org/contact">contact page</a>.
</p>
```

3.4.4   Setting the root route

Now that we've customized our site's pages and gotten a good start on the test suite, let's set the application's root route before moving on. As in Section 1.2.4 and Section 2.2.2, this involves editing the `routes.rb` file to connect / to a page of our choice, which in this case will be the Home page. (At this point, I also recommend removing the `hello` action from the Application controller if you added it in Section 3.1.) As shown in Listing 3.43, this means changing the `root` route from

---

[15]This exercise is solved in Section 5.3.1.

```
root 'application#hello'
```

to

```
root 'static_pages#home'
```

This arranges for requests for / to be routed to the **home** action in the Static Pages controller. The resulting routes file is shown in Figure 3.9.

Listing 3.43: Setting the root route to the Home page.
config/routes.rb

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get  'static_pages/home'
  get  'static_pages/help'
  get  'static_pages/about'
end
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.
To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Adding the root route in Listing 3.43 leads to the creation of a Rails helper called **root_url** (in analogy with helpers like **static_pages_home_-url**). By filling in the code marked **FILL_IN** in Listing 3.44, write a test for the root route.

2. Due to the code in Listing 3.43, the test in the previous exercise is already GREEN. In such a case, it's harder to be confident that we're actually testing what we think we're testing, so modify the code in Listing 3.43 by commenting out the root route to get to RED (Listing 3.45). (We'll talk more about Ruby comments in Section 4.2.) Then uncomment it (thereby restoring the original Listing 3.43) and verify that you get back to GREEN.
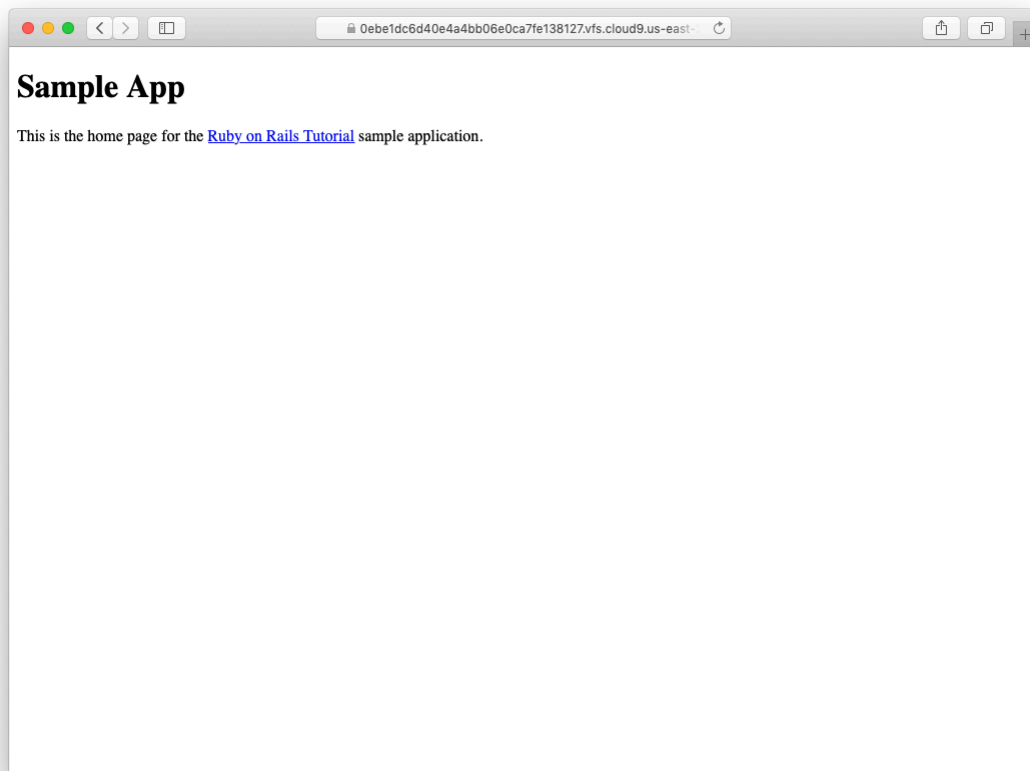
Figure 3.9: The Home page at the root route.

Listing 3.44: A test for the root route. GREEN
test/controllers/static_pages_controller_test.rb

```ruby
require 'test_helper'

class StaticPagesControllerTest < ActionDispatch::IntegrationTest

  test "should get root" do
    get FILL_IN
    assert_response FILL_IN
  end

  test "should get home" do
    get static_pages_home_url
    assert_response :success
  end

  test "should get help" do
    get static_pages_help_url
    assert_response :success
  end

  test "should get about" do
    get static_pages_about_url
    assert_response :success
  end
end
```

Listing 3.45: Commenting out the root route to get a failing test. RED
config/routes.rb

```ruby
Rails.application.routes.draw do
#   root 'static_pages#home'
  get  'static_pages/home'
  get  'static_pages/help'
  get  'static_pages/about'
end
```

3.5 Conclusion

Seen from the outside, this chapter hardly accomplished anything: we started with static pages, and ended with… *mostly* static pages. But appearances are

deceiving: by developing in terms of Rails controllers, actions, and views, we are now in a position to add arbitrary amounts of dynamic content to our site. Seeing exactly how this plays out is the task for the rest of this tutorial.

Before moving on, let's take a minute to commit the changes on our topic branch and merge them into the master branch. Back in Section 3.2 we created a Git branch for the development of static pages. If you haven't been making commits as we've been moving along, first make a commit indicating that we've reached a stopping point:

```
$ git add -A
$ git commit -m "Finish static pages"
```

Then merge the changes back into the master branch using the same technique as in Section 1.3.4:[16]

```
$ git checkout master
$ git merge static-pages
```

Once you reach a stopping point like this, it's usually a good idea to push your code up to a remote repository (which, if you followed the steps in Section 1.3.3, will be GitHub):

```
$ git push
```

I also recommend deploying the application to Heroku:

```
$ rails test
$ git push heroku
```

Here we've taken care to run the test suite before deploying, which is a good habit to develop.

---

[16]If you get an error message saying that the Spring process id (pid) file would be overwritten by the merge, just remove the file using `rm -f *.pid` at the command line.

3.5.1 What we learned in this chapter

- For a third time, we went through the full procedure of creating a new Rails application from scratch, installing the necessary gems, pushing it up to a remote repository, and deploying it to production.

- The **rails** script generates a new controller with **rails generate controller ControllerName <optional action names>**.

- New routes are defined in the file **config/routes.rb**.

- Rails views can contain static HTML or embedded Ruby (ERb).

- Automated testing allows us to write test suites that drive the development of new features, allow for confident refactoring, and catch regressions.

- Test-driven development uses a "Red, Green, Refactor" cycle.

- Rails layouts allow the use of a common template for pages in our application, thereby eliminating duplication.

3.6 Advanced testing setup

This optional section describes the testing setup used in the Ruby on Rails Tutorial screencast series. There are two main elements: an enhanced pass/fail reporter (Section 3.6.1), and an automated test runner that detects file changes and automatically runs the corresponding tests (Section 3.6.2). The code in this section is advanced and is presented for convenience only; you are not expected to understand it at this time.

The changes in this section should be made on the master branch:

```
$ git checkout master
```

3.6.1   minitest reporters

Although many systems, including the cloud IDE, will show the appropriate colors for RED and GREEN test suites, adding *minitest reporters* lends a degree of pleasant polish to the test outputs, so I recommend adding the code in Listing 3.46 to your test helper file,[17] thereby making use of the `minitest-reporters` gem included in Listing 3.2.

Listing 3.46: Configuring the tests to show RED and GREEN.
test/test_helper.rb

```ruby
ENV['RAILS_ENV'] ||= 'test'
require_relative '../config/environment'
require 'rails/test_help'
require "minitest/reporters"
Minitest::Reporters.use!

class ActiveSupport::TestCase
  # Run tests in parallel with specified workers
  parallelize(workers: :number_of_processors)

  # Setup all fixtures in test/fixtures/*.yml for all tests in alphabetical order.
  fixtures :all

  # Add more helper methods to be used by all tests here...
end
```

The resulting transition from RED to GREEN in the cloud IDE appears as in Figure 3.10.

3.6.2   Automated tests with Guard

One annoyance associated with using the `rails test` command is having to switch to the command line and run the tests by hand. To avoid this inconvenience, we can use *Guard* to automate the running of the tests. Guard monitors changes in the filesystem so that, for example, when we change the

---

[17]The code in Listing 3.46 mixes single- and double-quoted strings. This is because `rails new` generates single-quoted strings, whereas the minitest reporters documentation uses double-quoted strings. This mixing of the two string types is common in Ruby; see Section 4.2.1 for more information.

```
ubuntu:~/environment/sample_app (master) $ rails test
Running via Spring preloader in process 12327
Started with run options --seed 64190

 FAIL["test_should_get_about", #<Minitest::Reporters::Suite:0x000055efd7c1d690 @name="StaticPagesControllerTest">, 0.9476043219983694]
 test_should_get_about#StaticPagesControllerTest (0.95s)
        <About | Ruby on Rails Tutorial Sample App> expected but was
        <| Ruby on Rails Tutorial Sample App>..
        Expected 0 to be >= 1.
        test/controllers/static_pages_controller_test.rb:20:in `block in <class:StaticPagesControllerTest>'

  3/3: [=================================================================================] 100% Time: 00:00:00, Time: 00:00:00

Finished in 0.95923s
3 tests, 6 assertions, 1 failures, 0 errors, 0 skips
ubuntu:~/environment/sample_app (master) $ rails test
Running via Spring preloader in process 12353
Started with run options --seed 28649

  3/3: [=================================================================================] 100% Time: 00:00:00, Time: 00:00:00

Finished in 0.95176s
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
ubuntu:~/environment/sample_app (master) $ ▌
```

Figure 3.10: Going from RED to GREEN in the cloud IDE.

`static_pages_controller_test.rb` file, only those tests get run. Even better, we can configure Guard so that when, say, the `home.html.erb` file is modified, the `static_pages_controller_test.rb` automatically runs.

The `Gemfile` in Listing 3.2 has already included the guard gem in our application, so to get started we just need to initialize it:

```
$ bundle exec guard init
Writing new Guardfile to /home/ec2-user/environment/sample_app/Guardfile
00:51:32 - INFO - minitest guard added to Guardfile, feel free to edit it
```

We then edit the resulting `Guardfile` so that Guard will run the right tests when the integration tests and views are updated, which will look something like Listing 3.47. For maximum flexibility, I recommend using the version of the `Guardfile` listed in the reference application, which if you're reading this online should be identical to Listing 3.47:

- Reference `Guardfile` at railstutorial.org/guardfile

Listing 3.47: A custom `Guardfile`.

```
# Defines the matching rules for Guard.
guard :minitest, spring: "bin/rails test", all_on_start: false do
```

```ruby
  watch(%r{^test/(.*)/?(.*)_test\.rb$})
  watch('test/test_helper.rb') { 'test' }
  watch('config/routes.rb') { interface_tests }
  watch(%r{app/views/layouts/*}) { interface_tests }
  watch(%r{^app/models/(.*?)\.rb$}) do |matches|
    "test/models/#{matches[1]}_test.rb"
  end
  watch(%r{^app/controllers/(.*?)_controller\.rb$}) do |matches|
    resource_tests(matches[1])
  end
  watch(%r{^app/views/([^/]*?)/.*\.html\.erb$}) do |matches|
    ["test/controllers/#{matches[1]}_controller_test.rb"] +
    integration_tests(matches[1])
  end
  watch(%r{^app/helpers/(.*?)_helper\.rb$}) do |matches|
    integration_tests(matches[1])
  end
  watch('app/views/layouts/application.html.erb') do
    'test/integration/site_layout_test.rb'
  end
  watch('app/helpers/sessions_helper.rb') do
    integration_tests << 'test/helpers/sessions_helper_test.rb'
  end
  watch('app/controllers/sessions_controller.rb') do
    ['test/controllers/sessions_controller_test.rb',
     'test/integration/users_login_test.rb']
  end
  watch('app/controllers/account_activations_controller.rb') do
    'test/integration/users_signup_test.rb'
  end
  watch(%r{app/views/users/*}) do
    resource_tests('users') +
    ['test/integration/microposts_interface_test.rb']
  end
end

# Returns the integration tests corresponding to the given resource.
def integration_tests(resource = :all)
  if resource == :all
    Dir["test/integration/*"]
  else
    Dir["test/integration/#{resource}_*.rb"]
  end
end

# Returns all tests that hit the interface.
def interface_tests
  integration_tests << "test/controllers/"
end

# Returns the controller tests corresponding to the given resource.
```

```ruby
def controller_test(resource)
  "test/controllers/#{resource}_controller_test.rb"
end

# Returns all tests for the given resource.
def resource_tests(resource)
  integration_tests(resource) << controller_test(resource)
end
```

On the cloud IDE, there's one additional step, which is to run the following rather obscure commands to allow Guard to monitor all the files in the project:

```
$ echo fs.inotify.max_user_watches=524288 | sudo tee -a /etc/sysctl.conf
$ sudo sysctl -p
```

Once Guard is configured, you should open a new terminal (as with the Rails server in Section 1.2.2) and run it at the command line as follows (Figure 3.11):

```
$ bundle exec guard
```
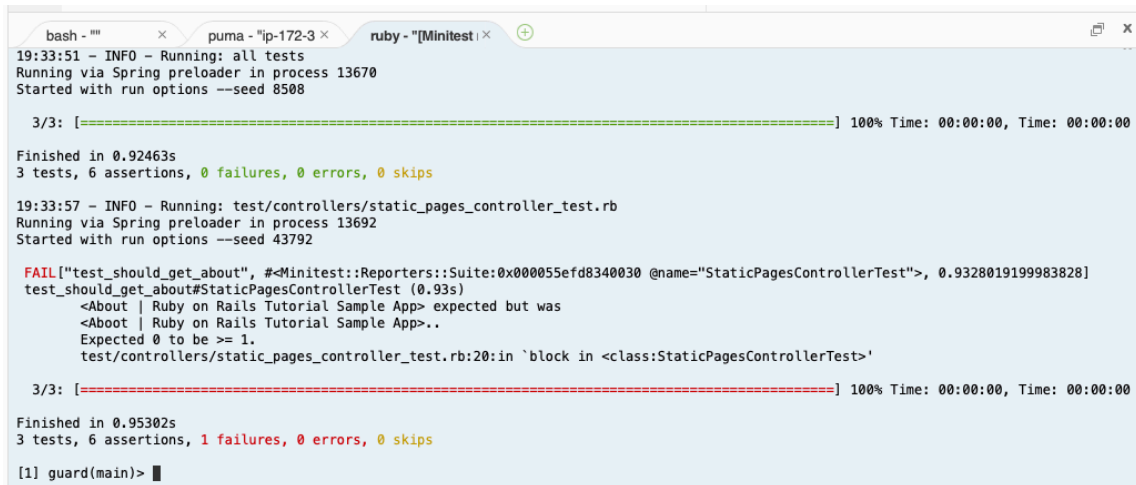
The rules in Listing 3.47 are optimized for this tutorial, automatically running (for example) the integration tests when a controller is changed. To run *all* the tests, simply hit return at the **guard>** prompt.

To exit Guard, press Ctrl-D. To add additional matchers to Guard, refer to the examples in Listing 3.47, the Guard README, and the Guard wiki.

If the test suite fails without apparent cause, try exiting Guard, stopping Spring (which Rails uses to pre-load information to help speed up tests), and restarting:

```
$ bin/spring stop    # Try this if the tests mysteriously start failing.
$ bundle exec guard
```

Before proceeding, you should add your changes and make a commit:

Figure 3.11: Using Guard on the cloud IDE.

```
$ git add -A
$ git commit -m "Complete advanced testing setup"
```