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>
```

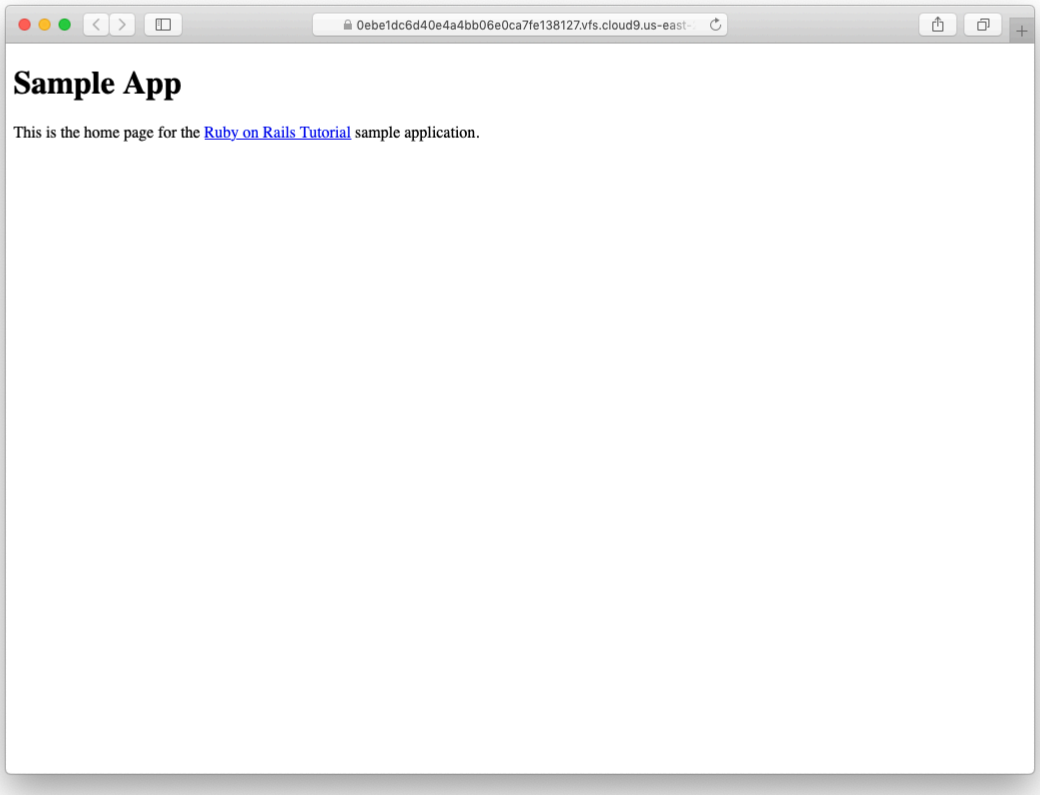
Listing 3.13: Custom HTML for the Help page.
app/views/static_pages/help.html.erb

```
<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

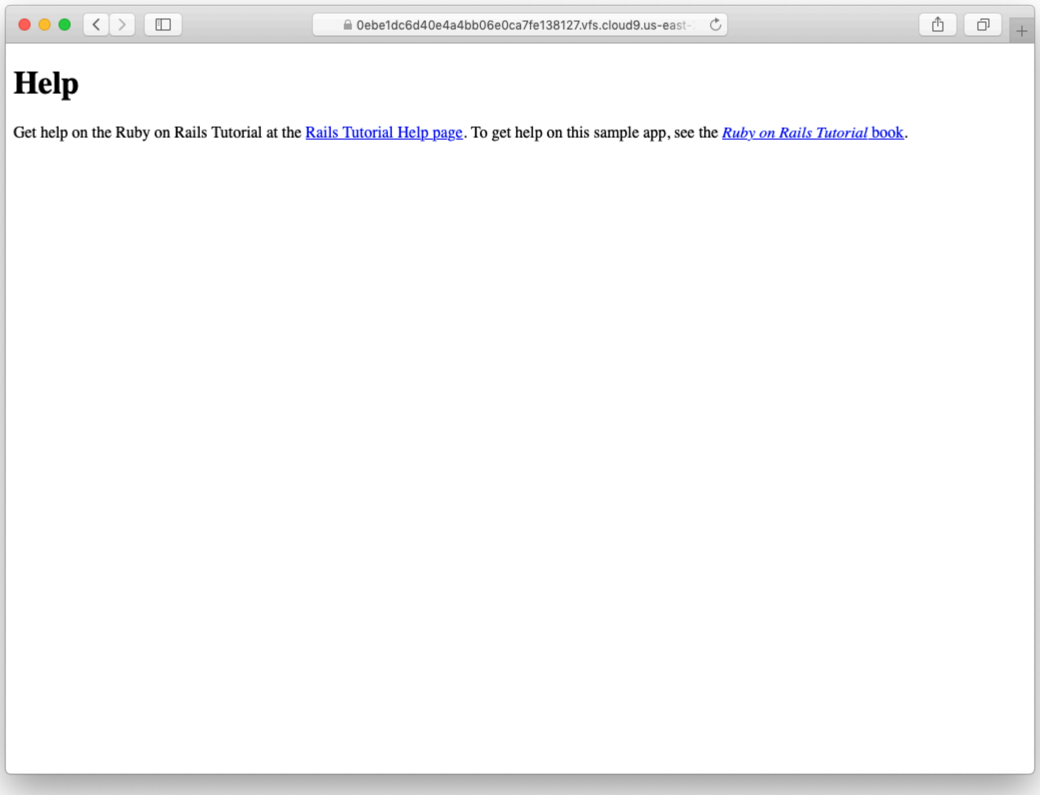
Having created and filled in the Home and Help pages for our sample app (Section 3.2.2), now we're going to add an About page as well. When making a change of this nature, it's a good practice to write an *automated test* to verify that the feature is implemented correctly. Developed over the course 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.



Sample App

This is the home page for the Ruby on Rails Tutorial sample application.

Figure 3.5: A custom Home page.



Help

Get help on the Ruby on Rails Tutorial at the Rails Tutorial Help page. To get help on this sample app, see the *Ruby on Rails Tutorial* book.

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

```
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.

```
  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:

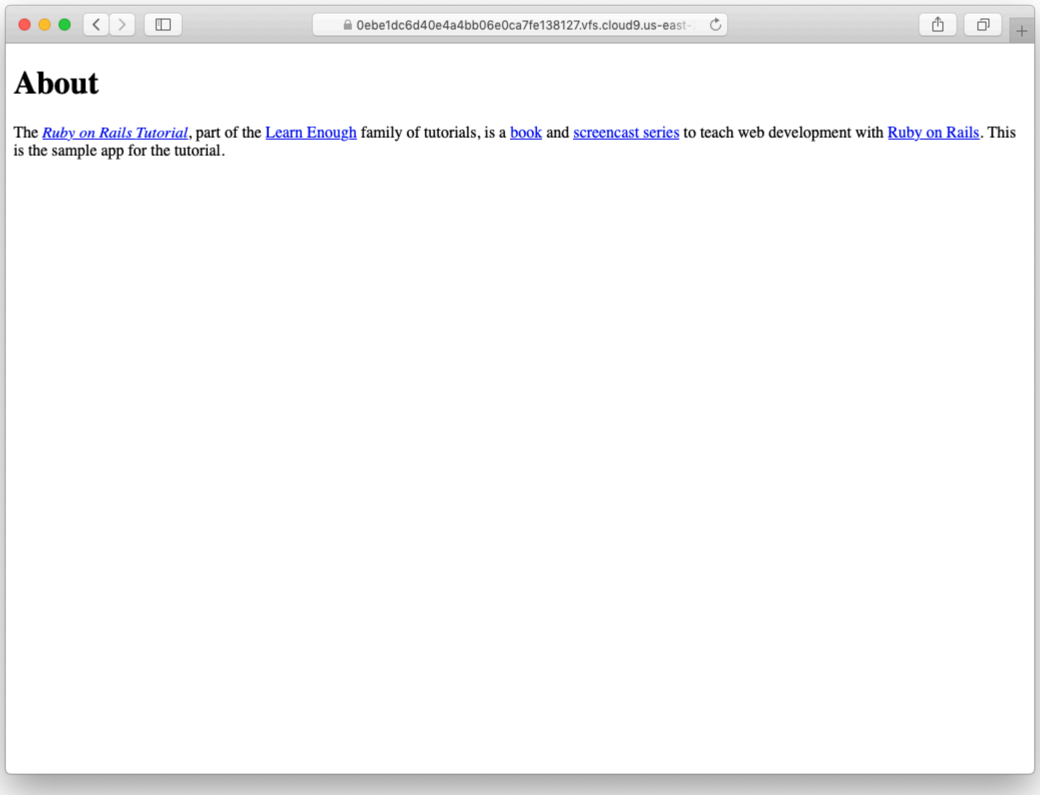
Listing 3.24: GREEN

```
$ 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.



About

The *Ruby on Rails Tutorial*, part of the Learn Enough family of tutorials, is a book and screencast series to teach web development with Ruby on Rails. This is the sample app for the tutorial.

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.