Daniel Kehoe

Rails and Twitter Bootstrap

Ruby on Rails tutorial from the RailsApps Project

Contents

Chapter 1

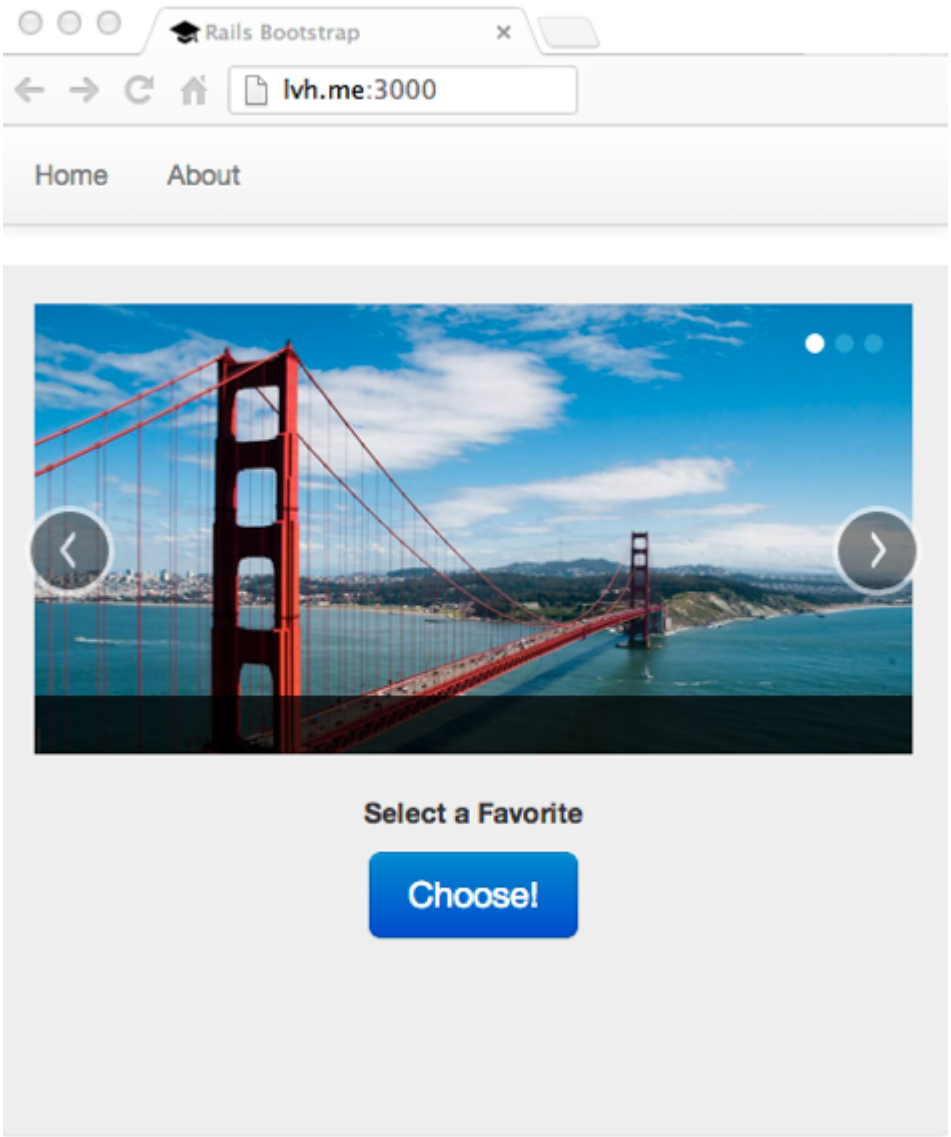
Introduction

Welcome. This tutorial shows how to integrate Twitter Bootstrap with Ruby on Rails.

Twitter Bootstrap provides CSS stylesheets and JavaScript code for the visual design of websites. CSS stylesheets are used for design and layout. JavaScript code combines user interaction elements and visual effects for features such as tabs, modal windows, and image carousels. Bootstrap is a framework for client-side (browser-based or "front end") development, much like Ruby on Rails is a framework for server-side development.

With this tutorial, you'll build a working web application and gain hands-on experience with Bootstrap and Rails.

Screenshot



Is It for You?

If you're new to Rails, start with the Learn Ruby on Rails book from the RailsApps project. The Learn Ruby on Rails book will give you a solid introduction to Rails so you are prepared for this tutorial.

For people who have never encountered HTML, CSS, or JavaScript, it's best to start elsewhere with an "Introduction to Web Design" course or online tutorial. There are thousands of such tutorials; I especially like the RailsBridge Frontend Curriculum.

This tutorial requires at least beginner-level experience with Rails, which means you should have experience building a simple Rails web application, such as the one in the Learn Ruby on Rails book.

If you are an experienced Rails developer, and you'd like to learn how to integrate Twitter Bootstrap with Rails, this tutorial is ideal. It will get you started quickly building a complete Rails starter application using Twitter Bootstrap.

The Application

We'll build a basic web application that you can use as a starter application for your own projects. The starter application integrates Twitter Bootstrap with Rails. Features include:

- Home page
- "About" page
- Navigation bar

You'll find the rails-bootstrap example application on GitHub. It is a working application that is maintained by a team of experienced developers so you can always check the "reference implementation" if you have problems. You can use the Rails Composer tool to quickly generate the complete starter application.

The starter application uses the high_voltage gem for the "About" page. Additional pages can easily be added. The high_voltage gem makes it easy to add pages with static content incorporating elements of a site-wide application layout such as header, navigation links, and footer.

After building the starter application, the tutorial goes further, introducing features of Twitter Bootstrap by implementing a simple survey website. Features include:

- Image carousel
- Survey form
- Google spreadsheet datastore
- Modal window

The tutorial uses Twitter Bootstrap to implement a carousel displaying multiple graphic images on the home page. The visitor is encouraged to vote for a favorite image. Voting is implemented with a survey form. When the visitor submits the form, the data is recorded in a Google Drive spreadsheet. Administrative access to the data is managed by Google Drive, eliminating the requirement for authentication and authorization in the application.

The application does not require a database. No pages are generated using information from a database and no user-submitted data is saved to a database. However, the sqlite3 gem must be present in the Gemfile because Rails ActiveRecord is used for form validation.

By using Google Drive for data storage, you get a fully functional survey website without the complexity of user management, session handling, or database migrations (these topics are addressed in other tutorials).

The tutorial will show you how to deploy the application to Heroku, a popular hosting platform for Rails applications.

Support the Project

The RailsApps project provides example applications that developers use as starter apps. Hundreds of developers use the apps, report problems as they arise, and propose solutions. Rails changes frequently; each application is known to work and serves as your personal "reference implementation" so you can stay up to date. Each is accompanied by a tutorial so there is no mystery code. Maintenance and development of the RailsApps applications is supported by subscriptions to the RailsApps tutorials.

Tutorials from @rails_apps take you on a guided path starting with absolute basics (the Learn Ruby on Rails
tutorial) and culminating with in-depth guides for professional Rails developers.

If you haven't done so, please consider purchasing a monthly subscription to support the project. You can purchase a subscription at tutorials.railsapps.org.

Chapter 2

Concepts

Twitter Bootstrap and other CSS front-end frameworks are toolkits that provide the kind of structure and convention that make Rails popular for back-end development. Twitter Bootstrap provides a standard grid for layout plus dozens of reusable components for common page elements such as navigation, forms, and buttons.

Similar Frameworks

Twitter Bootstrap is the most popular framework for front-end development but it is not the only one.

Web developers began putting together "boilerplate" CSS stylesheets as early as 2000, when browsers first began to fully support CSS. Boilerplate CSS made it easy to reuse CSS stylesheet rules from project to project. More importantly, designers often implemented "CSS reset" stylesheets to enforce typographic uniformity across different browsers.

In particular, front-end developers at large companies were faced with a need to establish standards for use of CSS across projects. Engineers at Yahoo! released the Yahoo! User Interface Library (YUI) as an open source project in February 2006. Inspired by an article by Jeff Croft, and reacting to the huge size of the YUI library, independent developers began releasing other CSS frameworks such as the 960 grid system and the Blueprint CSS framework.

There are dozens of CSS frameworks. In general, they all seek to implement a common set of requirements:

- An easily customizable grid
- Some default typography
- A typographic baseline
- CSS reset for default browser styles
- A stylesheet for printing

More recently, with the ubiquity of smartphones and tablets, CSS frameworks support responsive web design, accommodating differences in screen sizes across a range of devices.

Twitter Bootstrap came from an effort to document and share common design patterns and assets across projects at Twitter. It was released as an open source project in August 2011.

Perhaps because of Twitter's well-known brand identity, as well as an active developer community, Twitter Bootstrap is more popular than the other CSS frameworks.

Zurb Foundation is an excellent alternative to Twitter Bootstrap. For a comparison of the two frameworks, see Framework Fight.

Chapter 3

Resources

Before you dive into integrating Bootstrap with Rails, take a look at all the ways you can enrich your application with add-ons for Bootstrap. If you add nothing, you'll have a solid front-end framework for creating a graphic design, plus Bootstrap's dozen built-in JavaScript-based interactive elements. But because Bootstrap provides a structure for web design, and has a large and active developer community, there are hundreds of additional components, tools, and themes that can enhance your application.

Staying Up-to-Date

You can follow the official account for Bootstrap, @twbootstrap, on Twitter (of course).

Peter Cooper's HTML5 Weekly and JavaScript Weekly include occasional items about Twitter Bootstrap. I recommend his email newsletters, including Ruby Weekly.

The Bootstrap Hero site offers an occasional email newsletter announcing new Bootstrap resources.

References

The Twitter Bootstrap documentation is the best reference.

Resource Directories

The Bootstrap Hero site offers a Big Badass List of Useful Twitter Bootstrap Resources. To give you an idea of the scope of the directory, the listings include:

- Components
  - Display Components
  - Form Components
  - Input Components
- Framework Integrations (including Ruby gems)
- Interface Builders
  - Display

    - Structure
    - Theme
- Mockup Tools
- Tools and Services
- How Tos
    - Articles
    - Presentations
    - Videos
- Javascript Addons

The Bootstrap Hero list, in its magnitude, demonstrates the popularity of Bootstrap and the vitality of the open source community.

Cheatsheets

You'll find several if you google "bootstrap cheatsheets," though I haven't found any to recommend, probably because the Twitter Bootstrap documentation is more useful.

Snippets and Components

The Bootstrap documentation gives you a small selection of example code, just enough to illustrate how to use each Bootstrap component. But you can assemble Bootstrap's primitive components into complex design elements such as calendars, pricing tables, login forms, and many others. Instead of building what you need by trial and error, visit the Bootsnipp.com gallery for dozens of design elements and code examples.

For more components you can add to Bootstrap, see the extensive list of additional Bootstrap-compatible addons listed on the Bootstrap Hero site.

Bootstrap Themes

You may have visited ThemeForest or other theme galleries that offer pre-built themes for a few dollars each. These huge commercial galleries offer themes for WordPress, Tumblr, or CMS applications such as Drupal or Joomla. Many of their themes are based on Bootstrap. For example, you can search ThemeForest and find over 500 themes that are compatible with Bootstrap 2.3. In principle, it is possible to adapt any of these themes for use with Rails.

You can find Bootstrap-based themes created specifically for Rails:

- Dressed Rails Themes from Marc-André Cournoyer

Other marketplaces exist solely to offer themes for Bootstrap. Any of these themes can be adapted for Rails:

- Start Bootstrap
- WrapBootstrap

One offering from WrapBootstrap is packaged as a Rails gem:

- RailsStrap by Leon Touroutoglou

The Bootswatch site offers free themes for Twitter Bootstrap that can be adapted for Rails.

If you want the retro look of the 1990s web, use:

- Geo for Bootstrap

You've got to see it. Believe it or not, websites used to look like that.

Getting Help

What to do when you get stuck?

"Google it," of course. Often you need to eliminate outdated advice that is relevant only for older versions of Rails. Google has options under "Search tools" to show only recent results from the past year.

Go directly to Stack Overflow for answers to your questions, or look for Stack Overflow answers in Google search results. Answers from Stack Overflow are helpful if you check carefully to make sure the answers are recent. It is wise to compare several answers to find what's relevant to your own circumstances.

Rails Hotline is a free telephone hotline for Rails questions staffed by volunteers. You'll need to carefully think about and describe your problem but sometimes there's no better help than a live expert.

Chapter 4

Accounts You May Need

The tutorial will show how to use Git for version control. If you want to store your application on GitHub, you can get a free personal GitHub account. See the article Rails and Git for more information.

Google Drive

The application will use a Google Drive spreadsheet for data storage. If you have a Gmail account, your username and password are the credentials you use to access Google Drive. If you don't have a Gmail account, you can sign up for a Google account for free.

Heroku

The tutorial will show how to deploy the application to Heroku which provides Rails application hosting. It costs nothing to set up a Heroku account and deploy as many applications as you want.

To deploy an app to Heroku, you must have a Heroku account. Visit https://id.heroku.com/signup/devcenter to set up an account.

Be sure to use the same email address you used to register for GitHub. It's very important that you use the same email address for GitHub and Heroku accounts.

Chapter 5

Getting Started

Before You Start

If you follow this tutorial closely, you'll have a working application that closely matches the example app in the rails-bootstrap GitHub repository. If your application doesn't work after following the tutorial, compare the code to the example app in the GitHub repository, which is known to work.

If you find problems or wish to suggest improvements, please create a GitHub issue. It's best to download and check the example application from the GitHub repository before you report an issue, just to make sure the error isn't a result of your own mistake.

The online edition of this tutorial contains a comments section at the end of the tutorial. I encourage you to offer feedback to improve this tutorial.

Your Development Environment

The Learn Ruby on Rails tutorial explains how to set up a text editor and terminal application.

This tutorial will use Git for version control. See the article Rails and Git for more information.

Ruby 2.0

See article Installing Rails to install Ruby 2.0, the newest version of Ruby. The article will guide you to update various supporting gems and utilities as needed.

Check that the appropriate version of Ruby is installed in your development environment. Open your terminal application and enter:

```
$ ruby -v
```

You should see Ruby version 2.0.0 or newer.

If you are running older versions or Ruby on your computer, you must install a newer version to avoid unexpected problems.

Project-Specific Gemset

The instructions in the article Installing Rails recommend using RVM, the Ruby version manager. If you are an experienced Unix administrator, you can consider alternatives such as Chruby, Sam Stephenson's rbenv, or others on this list. RVM is popular, well-supported, and an excellent utility to help a developer install Ruby and manage gemsets; that's why I recommend it.

For our rails-bootstrap application, we'll create a project-specific gemset using RVM. We'll give the gemset the same name as our application.

After we create a the project-specific gemset, we'll install the Rails gem into the gemset. Enter these commands:

```
$ rvm use ruby-2.0.0@rails-bootstrap --create
$ gem install rails
```

Make sure Rails is ready to run. Open a terminal and type:

```
$ rails -v
```

You should have Rails version 4.0.0 or newer.

Chapter 6

Create the Application

You have several options for getting the code. You can *copy from the tutorial*, *fork*, *clone*, or *generate*.

Copy from the Tutorial

To create the application, you can cut and paste the code from the tutorial into your own files. It's a bit tedious and error-prone but you'll have a good opportunity to examine the code closely.

Other Options

Fork

If you'd like to add features (or bug fixes) to improve the example application, you can fork the GitHub repo and make pull requests. Your code contributions are welcome!

Clone

If you want to copy and customize the app with changes that are only useful for your own project, you can download or clone the GitHub repo. You'll need to search-and-replace the project name throughout the application. You probably should generate the app instead (see below). To clone:

```
$ git clone git://github.com/RailsApps/rails-bootstrap.git
```

You'll need git on your machine. See Rails and Git.

Generate

If you wish to skip the tutorial and build the application immediately, use the Rails Composer tool to generate the complete example app. You'll be able to give it your own project name when you generate the app. Generating the application gives you additional options.

To build the complete example application immediately, see the instructions in the README for the rails-bootstrap example application.

Building from Scratch

We'll use the Rails default starter application as the basis for our application.

We already created a project-specific gemset using RVM. Make sure it's ready to use:

```
$ rvm use ruby-2.0.0@rails-bootstrap
```

To create the Rails default starter application, type:

```
$ rails new rails-bootstrap
```

This will create a new Rails application named "rails-bootstrap."

You may give the app a different name if you are building it for another purpose. For this tutorial, we'll assume the name is "rails-bootstrap." You'll avoid extra steps and errors by using this name.

After you create the application, switch to its folder to continue work directly in the application:

```
$ cd rails-bootstrap
```

This is your project directory. It is also called the application root directory.

Setting RVM

RVM gives us a convenient technique to make sure we are always using the correct gemset when we enter the project directory. It will create hidden files to designate the correct Ruby version and project-specific gemset. Enter this command to create the hidden files:

```
$ rvm use ruby-2.0.0@rails-bootstrap --ruby-version
```

The command creates two files, **.ruby-version** and **.ruby-gemset**, that set RVM every time we `cd` to the project directory.

Replace the READMEs

Please edit the README files to add a description of the app and your contact info. Changing the README is important if your app will be publicly visible on GitHub. Otherwise, people will think I am the author of your app. If you like, add an acknowledgment and a link to the RailsApps project.

Set Up Source Control (Git)

If you're creating an application for deployment into production, you'll want to set up a source control repository at this point. If you are building a throw-away app for your own education, you may skip this step.

```
$ git init .
$ git add -A
$ git commit -m 'Initial commit'
```

See detailed instructions for Rails and Git.

If you want to store your application on GitHub, you should now set up your GitHub repository.

Test the Application

You've created a simple default web application. It's ready to run.

Launching the Web Server

You can launch the application by entering the command:

```
$ rails server
```

The `rails server` command launches the default WEBrick web server that is provided with Ruby.

Viewing in the Web Browser

To see your application in action, open a web browser window and navigate to http://localhost:3000/. You'll see the Rails default information page.

Stopping the Web Server

You can stop the server with Control-c to return to the command prompt.

Most of the time you'll keep the web server running as you add or edit files in your project. Changes will automatically appear when you refresh the browser or request a new page. There is a tricky exception, however. If you make changes to the Gemfile, or changes to configuration files, the web server must be shut down and relaunched for changes to be activated.

Chapter 7

Gems

Here are gems we'll add to the Gemfile (in alphabetical order):

- bootstrap-sass – Twitter Bootstrap for CSS and JavaScript
- figaro – configuration framework
- high_voltage – for static pages like "about"
- simple_form – forms made easy

We'll also add utilities that make development easier:

- better_errors – helps when things go wrong
- quiet_assets – suppresses distracting messages in the log

Later, when we begin to explore Twitter Bootstrap and customize our starter app, we'll add more gems. For now, these are all basic gems that are useful for any starter application.

Open your **Gemfile** and replace the contents with the following:

Gemfile

```
source 'https://rubygems.org'
ruby '2.0.0'
gem 'rails', '4.0.0'

# Rails defaults
gem 'sqlite3'
gem 'sass-rails', '~> 4.0.0'
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.0.0'
gem 'jquery-rails'
gem 'turbolinks'
gem 'jbuilder', '~> 1.2'

# rails-bootstrap
gem 'bootstrap-sass'
gem 'figaro'
gem 'high_voltage'
gem 'simple_form', '>= 3.0.0.rc'
group :development do
  gem 'better_errors'
  gem 'quiet_assets'
end
```

Check for the current version of Rails and replace `gem 'rails', '4.0.0'` accordingly. The Rails default gems may have changed as well.

Notice that we've placed the better_errors and quiet_assets gems inside a "group." Specifying a group for development or testing insures a gem is not loaded in production, reducing the application's memory footprint.

Install the Gems

Now that you've edited the Gemfile, you need to install the required gems on your computer:

```
$ bundle install
```

You can check which gems are loaded into the development environment:

```
$ gem list
```

Keep in mind that you have installed these gems locally. When you deploy the application to another server, the same gems (and versions) must be available.

Each time you edit the Gemfile, run `bundle install` and restart your web server.

Git

Let's commit our changes to the Git repository and push to GitHub:

```
$ git add -A
$ git commit -m "add gems"
$ git push origin master
```

If you get a message:

```
fatal: 'origin' does not appear to be a git repository
fatal: The remote end hung up unexpectedly
```

It shows that you can't connect to GitHub to push the changes. It is not absolutely necessary to use GitHub for this tutorial.

We're ready to configure the application.

Chapter 8

Configuration

Many applications need a way to set configuration values such as account credentials or API keys. It is a basic feature for our starter application.

You shouldn't save account credentials to a shared Git repository where others can see them. To keep account credentials safe, we use the figaro gem to set environment variables from the Unix shell or from a simple configuration file. You can read the article Rails Environment Variables if you'd like learn about the figaro gem or explore other approaches.

Configuration File

We've already installed the figaro gem in the Gemfile and run `bundle install`.

The figaro gem needs to set up files. Run:

```
$ rails generate figaro:install
```

Using the `rails generate` command, the figaro gem generates a **config/application.yml** file and modifies the **.gitignore** file. The **.gitignore** file prevents the **config/application.yml** file from being saved in the Git repository so your credentials are kept private.

You can take a look at the **config/application.yml** file:

```
# Add application configuration variables here, as shown below.
#
# PUSHER_APP_ID: "2954"
# PUSHER_KEY: 7381a978f7dd7f9a1117
# PUSHER_SECRET: abdc3b896a0ffb85d373
# STRIPE_API_KEY: EdAvEPVEC3LuaTg5Q3z6WbDVqZlcBQ8Z
# STRIPE_PUBLIC_KEY: pk_BRgD57O8fHja9HxduJUszhef6jCyS
```

These are just example configuration values. Our starter application doesn't require configuration settings, so we'll leave the file unchanged. If you customize the starter application for your own use, you can set configuration values in the file.

Later in this tutorial, when we go beyond the basic starter application and begin to explore Twitter Bootstrap in more depth, we'll set credentials for your Google account in the **config/application.yml** file.

Git

Let's commit our changes to the Git repository.

```
$ git add -A
$ git commit -m "add configuration file"
$ git push origin master
```

We're ready to create a home page for the application.

Chapter 9

Home Page

We'll create a home page for our application. We'll need a controller, view file, and routing. Initially, the home page will be a placeholder. Later we'll add a graphic carousel and survey form so we can explore features of Twitter Bootstrap.

User Story

We'll plan our work with a user story:

```
*Home page*
As a visitor to the website
I want to see the name of the website
And a link to an "About" page
so I can identify the website and learn about its purpose
```

The user story is simple and obvious. We'll modify it as we add features to the website.

Controller

We need a controller that will render our home page. We could call it the Home controller or the Welcome controller, but based on our user story, a Visitors controller is appropriate. Later, when we add a form to the home page, we'll create a Visitor model to match the Visitors controller.

We'll use the class name `VisitorsController` and **visitors_controller.rb** will be the filename.

Create a file **app/controllers/visitors_controller.rb**:

```
class VisitorsController < ApplicationController

  def new
  end

end
```

We define the class and name it `class VisitorsController`, inheriting behavior from the ApplicationController class which is defined in the Rails API.

We define the `new` method so we have a stub for features we'll add later. Hidden behavior inherited from the ApplicationController does all the work of rendering the view. Invoking the `new` method calls a `render` method supplied by the ApplicationController parent class. The `render` method searches in the **app/views/visitors** directory for a view file named **new** (the file extension **.html.erb** is assumed by default).

View

We'll need an **app/views/** directory for our view file. You can make a new folder using your file browser or text editor. Or use the Unix `mkdir` command:

```
$ mkdir app/views/visitors
```

Create a file **app/views/visitors/new.html.erb**:

```
<h3>Home</h3>
```

We've created a **visitors/** folder within the **app/views/** directory. We name our View file **new.html.erb**, adding the **.erb** file extension so that Rails will use the ERB templating engine to interpret the markup.

There are several syntaxes that can be used for a view file. In this tutorial, we'll use the ERB syntax that is most commonly used by beginners. Some experienced developers prefer to add gems that provide the Haml or Slim templating engines.

Routing

We'll need to set a route to the home page.

Open the file **config/routes.rb**. Replace the contents with this:

```
RailsBootstrap::Application.routes.draw do
  root :to => 'visitors#new'
end
```

Any request to the application root (http://localhost:3000/) will be directed to the VisitorsController `new` action.

Notice that the name of the application is contained in the **config/routes.rb** file. Earlier, I recommended using "rails-bootstrap" as the name of the application. If you didn't do so, you will need to change the code here.

For more information about routing, see the reference documentation, RailsGuides: Routing from the Outside In.

Test the Application

You've changed the routing file so you must restart your application server. Use Control-c to stop the server and restart it:

Now let's run the application.

Enter the command:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/. You'll see our new home page.

Git

Let's commit our changes to the Git repository.

```
$ git add -A
$ git commit -m "home page"
$ git push origin master
```

Now we'll look at integrating Twitter Bootstrap with our application.

Chapter 10

Layout and Views

In this chapter we'll look closely at view files, particularly the application layout, so we can organize the design of our web pages. In the next chapter we'll learn how to add CSS stylesheets to complete the graphic design of our web pages.

Application Layout

We've already created the view file for our home page. Rails will combine the Visitors#New view with the default application layout file. The default application layout is where you put HTML that you want to include on every page of your website.

Open the file **app/views/layouts/application.html.erb**:

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

<%= yield %>

</body>
</html>
```

The article Rails Default Application Layout explains more about the application layout.

The default application layout contains the Ruby keyword `yield`. The `yield` keyword is replaced with a view file that is specific to the controller and action, in this case, the **app/views/visitors/new.html.erb** view file. The content from the view is inserted where you place the `yield` keyword.

Basic Boilerplate

The default application layout contains the standard HTML `DOCTYPE`, `<head>`, and `<body>` tags.

Rails view helpers are included within the `<%= ... %>` delimiters:

- `stylesheet_link_tag` – generates an HTML `<link>` tag for CSS
- `javascript_include_tag` – generates an HTML `<script>` tag for JavaScript
- `csrf_meta_tags` – generates `<meta>` tags that prevent cross-site request forgery

The first two tags add CSS and JavaScript to the web page using the Rails *asset pipeline*.

Asset Pipeline

When building static websites, webmasters add JavaScript to a page using the `<script>` tag. For every JavaScript file, they add an additional `<script>` tag, so a page HEAD section looks like this:

```
<!DOCTYPE html>
<html>
<head>
  <title>Page that uses multiple JavaScript files</title>
  <script src="jquery.js" type="text/javascript"></script>
  <script src="jquery.plugin.js" type="text/javascript"></script>
  <script src="custom.js" type="text/javascript"></script>
</head>
```

The same is true for CSS files. When you organize stylesheets in multiple files, you add a `<link>` tag for each one.

There's a drawback to using multiple files. Requesting files from the server is a time-consuming operation for a web browser, so every extra file request slows down the browser.

Rails improves website performance by providing the *asset pipeline* utility. The Rails asset pipeline improves website performance by concatenating multiple JavaScript or CSS files into a single script, allowing the developer to organize code in separate files, but eliminating the performance penalty of multiple `<script>` or `<link>` tags. The Rails asset pipeline also compresses JavaScript and CSS files for faster page loads.

You'll use the **app/assets/** directory to fill the asset pipeline. The directory contains two folders:

- **app/assets/javascripts/**
- **app/assets/stylesheets/**

For now, just note that any JavaScript and CSS files you add to these folders will be automatically added to every page. We've already seen the tags that perform this service:

```
<%= stylesheet_link_tag "application", media: "all", "data-turbolinks-track" => true %>
<%= javascript_include_tag "application", "data-turbolinks-track" => true %>
```

Later in the tutorial, we'll use the asset pipeline to add CSS and JavaScript to our application.

Adding Boilerplate

Webmasters who build static websites are accustomed to setting up web pages with "boilerplate," or basic templates for a standard web page. The well-known HTML5 Boilerplate project has been recommending "best practice" tweaks to web pages since 2010. Very few of the HTML5 Boilerplate recommendations are relevant for Rails developers, as Rails already provides almost everything required. If you want to learn more, the article HTML5 Boilerplate for Rails Developers looks at the recommendations.

There is only one boilerplate item you should add: the `viewport` metatag.

Viewport

The `viewport` metatag improves the presentation of web pages on mobile devices. Setting a viewport tells the browser how content should fit on the device's screen. Apple's developer documentation on Configuring the Viewport provides details.

The `viewport` metatag looks like this:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0">
```

Title and Description

If you want to maximize traffic to your website, you should make your web pages search-engine friendly. That means adding *title* and *description* metatags. Google uses contents of the title tag to display titles in search results. And it will sometimes use the content of a description metatag in search results snippets. See Google's explanation of how it uses Site Title and Description. Good titles and descriptions improve clickthrough from Google searches. Later in the tutorial, we'll see how to use a `content_for` statement to set a title and description for a page.

Adding title and description looks like this:

```
<title><%= content_for?(:title) ? yield(:title) : "Rails Bootstrap" %></title>
<meta name="description" content="<%= content_for?(:description) ? yield(:description) :
"Learning Rails" %>">
```

The code uses the Ruby ternary operator. It's a fancy conditional statement that says, "if `content_for?(:title)` is present in the view file, use `yield(:title)` to include it, otherwise just display 'Rails Bootstrap'."

HTML5 Elements

To complete our application layout, we'll add a few structural elements to make it easy to apply CSS styling. These elements are not unique to a Rails application and will be familiar to anyone who has done front-end development.

We'll add a container div to the page. This is a common technique among front-end designers, particularly useful for adding margins or borders to the page layout.

The HTML5 specification provides `<header>` and `<footer>` elements that add structure to a web page. The tags don't add any new behavior but make it easier to determine the structure of the page and apply CSS styles. The `<header>` tag is typically used for branding or navigation and the `<footer>` tag typically contains links to copyright information, legal disclaimers, or contact information.

Finally we'll wrap our `yield` in a `<main role="main">` element. The `<main>` tag is among the newest HTML5 elements (see the W3C specification for details). From the specification: "The main content area of a document includes content that is unique to that document and excludes content that is repeated across a set of documents such as site navigation links, copyright information, site logos." The specification recommends, "Authors are advised to use ARIA `role="main"` attribute on the main element until user agents implement the required role mapping." We'll follow the advice of the specification and wrap our unique content in the `<main>` tag.

Let's put this all together for our updated application layout.

Updated Application Layout

Modify the file **app/views/layouts/application.html.erb** to include everything we've discussed in this chapter:

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title><%= content_for?(:title) ? yield(:title) : "Rails Bootstrap" %></title>
  <meta name="description" content="<%= content_for?(:description) ? yield(:description)
: "Rails Bootstrap" %>">
  <%= stylesheet_link_tag "application", media: "all", "data-turbolinks-track" => true %>
  <%= javascript_include_tag "application", "data-turbolinks-track" => true %>
  <%= csrf_meta_tags %>
</head>
<body>
  <div id="container">
    <header>
    </header>
    <main role="main">
      <%= yield %>
    </main>
    <footer>
    </footer>
  </div>
</body>
</html>
```

We've added the `viewport` metatag, replaced the title, and added a description.

We've also added HTML5 structural elements.

Our application layout is complete—for now. Next we'll revise it to support styling with Twitter Bootstrap.

Chapter 11

Stylesheets

This chapter shows how to add Twitter Bootstrap to your Rails application. You'll add the Twitter Bootstrap CSS files as well as the Twitter Bootstrap JavaScript libraries.

Asset Pipeline for CSS

In the previous chapter, you learned about the Rails *asset pipeline*. The asset pipeline helps web pages display faster in the browser by combining all CSS files into a single file (it does the same for JavaScript).

Your CSS stylesheets get concatenated and compacted for delivery to the browser when you add them to this directory:

- **app/assets/stylesheets/**

The default Rails starter application contains one file in the directory:

- **app/assets/stylesheets/application.css**

Open the file in your text editor and you'll see this:

```
/*
 * This is a manifest file that'll be compiled into application.css, which will include
all the files
 * listed below.
 *
 * Any CSS and SCSS file within this directory, lib/assets/stylesheets, vendor/assets/
stylesheets,
 * or vendor/assets/stylesheets of plugins, if any, can be referenced here using a
relative path.
 *
 * You're free to add application-wide styles to this file and they'll appear at the
top of the
 * compiled file, but it's generally better to create a new file per style scope.
 *
 *= require_self
 *= require_tree .
 */
```

The **app/assets/stylesheets/application.css** file serves two purposes. First, you can add any CSS rules to the file that you want to use anywhere on your website. Second, the file serves as a *manifest*, providing a list of files that should be concatenated and included in the single CSS file that is delivered to the browser.

It's bad practice to place all your CSS in the **app/assets/stylesheets/application.css** file (unless your CSS is very limited). Instead, structure your CSS in multiple files. CSS that is used on only a single page can go in a file with a name that matches the page. Or, if sections of the website share common elements, such as themes for landing pages or administrative pages, make a file for each theme. How you organize your CSS is up to you; the asset pipeline lets you organize your CSS so it is easier to develop and maintain. Just add the files to the **app/assets/stylesheets/** folder.

It's not obvious from the name of the **app/assets/stylesheets/application.css** file that it serves as a *manifest file* as well as a location for miscellaneous CSS rules. For most websites, you can ignore its role as a manifest file. In the comments at the top of the file, the `*= require_self` directive indicates that any CSS in the file should be delivered to the browser. The `*= require_tree .` directive (note the Unix "dot operator") indicates any files in the same folder, including files in subfolders, should be combined into a single file for delivery to the browser.

If your website is large and complex, you can remove the `*= require_tree .` directive and specify individual files to be included in the file that is generated by the asset pipeline. This gives you the option of reducing the size of the application-wide CSS file that is delivered to the browser. For example, you might segregate a file that includes CSS that is used only in the site's administrative section. In general, only large and complex sites need this optimization. The speed of rendering a single large CSS file is faster than fetching multiple files.

Rename the application.css File

It's a good idea to rename the **app/assets/stylesheets/application.css** file as **app/assets/stylesheets/application.css.scss**. You'll add the **.scss** file extension. This will allow you to use the advantages of the Sass syntax for your application stylesheet.

Rename the file now:

```
$ mv app/assets/stylesheets/application.css app/assets/stylesheets/application.css.scss
```

CSS Styling with Sass

Ordinary CSS is not a programming language so CSS rules are verbose and often repetitive. Sass is a preprocessor for CSS so your stylesheets can use variables, *mixins*, and nesting of CSS rules. You can create a variable such as `$blue: #3bbfce` and specify colors anywhere

using the variable, for example, `border-color: $blue`. Mixins are like variables that let you use snippets of reusable CSS. Nesting eliminates repetition by layering CSS selectors.

You can use Sass in any file by adding the file extension **.scss**. The asset pipeline will preprocess the file and expand it as standard CSS.

For more on the advantages of Sass and how to use it, see the the Sass website or the Sass Basics RailsCast from Ryan Bates.

Twitter Bootstrap Gem

Several options are available for installing Twitter Bootstrap in a Rails application. Gems are available for either the native Bootstrap LESS language or the Sass language that is the default for CSS files in Rails. Sass is a default for CSS development in Rails so I recommend using Thomas McDonald's bootstrap-sass gem to add Twitter Bootstrap.

If you want to explore the differences between LESS and Sass, see the article Sass vs. LESS. In general, it recommends Sass.

In your **Gemfile**, you've already added:

```
gem 'bootstrap-sass'
```

and previously run `$ bundle install`.

Add Twitter Bootstrap CSS

Add a file **app/assets/stylesheets/bootstrap_and_overrides.css.scss** containing:

```
@import "bootstrap";
body { padding-top: 60px; }
@import "bootstrap-responsive";
```

The file **app/assets/stylesheets/bootstrap_and_overrides.css.scss** is automatically included and compiled into your Rails application.css file by the `*= require_tree .` statement in the **app/assets/stylesheets/application.css.scss** file.

The file will import both basic Bootstrap CSS rules and the Bootstrap rules for responsive design (allowing the layout to resize for various devices and screen sizes).

The CSS rule `body { padding-top: 60px; }` applies an additional CSS rule to accommodate the Bootstrap navigation bar heading created by the `navbar-fixed-top class` in the `header` tag in the layout we'll use.

You could add the Bootstrap `@import` code to the **app/assets/stylesheets/application.css.scss** file. However, I recommend adding the separate **app/assets/stylesheets/bootstrap_and_overrides.css.scss** file. You may wish to modify the Twitter Bootstrap CSS rules; placing changes to Twitter Bootstrap CSS rules in the **bootstrap_and_overrides.css.scss** file will keep your CSS better organized.

Add Twitter Bootstrap JavaScript

Include the Twitter Bootstrap JavaScript files by modifying the file **app/assets/javascripts/application.js**:

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

You'll see you need to add the directive `//= require bootstrap` before `//= require_tree .`.

Like the **application.css.scss** file, the **application.js** file is a manifest that allows a developer to designate the JavaScript files that will be combined for delivery to the browser.

CSS Example

We'll add a nice gray box as a background to page content. This gives us an example of adding a CSS rule that will be used on every page of the application.

Add this to your **app/assets/stylesheets/application.css.scss** file for a gray background:

```
.content {
  background-color: #eee;
  padding: 20px;
  margin: 0 -20px;
  -webkit-border-radius: 0 0 6px 6px;
  -moz-border-radius: 0 0 6px 6px;
  border-radius: 0 0 6px 6px;
  -webkit-box-shadow: 0 1px 2px rgba(0,0,0,.15);
  -moz-box-shadow: 0 1px 2px rgba(0,0,0,.15);
  box-shadow: 0 1px 2px rgba(0,0,0,.15);
}
```

The CSS code applies styling to a `.content` class. It sets background color, a border and shadow, padding and margin. It's complicated by accommodating differences among web browsers.

Set up SimpleForm with Twitter Bootstrap

Our tutorial application will contain a survey form. We could set up styling for the form when we add the form to the home page, but it is convenient to set up form styling now, as we would if we were adding multiple forms to the site.

Rails provides a set of view helpers for forms. They are described in the RailsGuides: Rails Form Helpers document. Most developers use an alternative set of form helpers named SimpleForm, provided by the SimpleForm gem. The SimpleForm helpers are more powerful, easier to use, and offer an option for styling with Twitter Bootstrap.

The SimpleForm and Twitter Bootstrap Demo shows what you can expect.

In your **Gemfile**, you've already added:

```
gem 'simple_form', '>= 3.0.0.rc'
```

and previously run `$ bundle install`.

Run the generator to install SimpleForm with a Twitter Bootstrap option:

```
$ rails generate simple_form:install --bootstrap
```

which installs several configuration files:

```
config/initializers/simple_form.rb
config/initializers/simple_form_bootstrap.rb
config/locales/simple_form.en.yml
lib/templates/erb/scaffold/_form.html.erb
```

Here the SimpleForm gem uses the generator command to create files for initialization and localization (language translation). SimpleForm can be customized with settings in the initialization file. We'll use the defaults.

Chapter 12

Navigation and Flash Messages

Previously we looked at the application layout and used the asset pipeline to add CSS and JavaScript to our application. In this chapter, we'll add navigation links and a Rails flash message to our application layout.

Navigation Links

Every website needs navigation links. For our application, we'll want links for Home and About.

You can add navigation links directly to your application layout but many Rails developers prefer to create a partial template – a "partial" – to better organize the default application layout.

A *partial* is similar to any view file, except the filename begins with an underscore character. Save the file in any view folder and use the `render` keyword to insert the partial. Here's an example of using the `render` keyword with a hypothetical file named **app/views/layouts/_footer.html.erb**:

```
<%= render 'layouts/footer' %>
```

Notice that you specify the folder within the **app/views/** directory with a truncated version of the filename. The `render` method doesn't want the `_` underscore character or the `.html.erb` file extension. That can be confusing; it makes sense when you remember that Rails likes "convention over configuration" and economizes on extra characters when possible.

There's no rule against using raw HTML in our view files, so we could create a partial for navigation links that uses the HTML `<a>` anchor tag like this:

```
<ul class="nav">
  <li><a href="/">Home</a></li>
  <li><a href="/about">About</a></li>
</ul>
```

Rails gives us another option, however. We can use the Rails `link_to` view helper instead of the HTML `<a>` anchor tag. The Rails `link_to` helper eliminates the crufty `<>` angle brackets and the unnecessary `href=""`. More importantly, it adds a layer of abstraction, using the routing configuration file to form links. This is advantageous if we make changes to the location of the link destinations. If we used the raw HTML `<a>` anchor tag, we'd have to change the raw HTML everywhere we had a link to the home page. Using the Rails `link_to` helper, we name a route and make any changes once, in the **config/routes.rb** file.

Using the `link_to` helper, create a file **app/views/layouts/_navigation.html.erb**:

```
<ul class="nav">
  <li><%= link_to 'Home', root_path %></li>
  <li><%= link_to 'About', page_path('about') %></li>
</ul>
```

Here the `link_to` helper takes two parameters. The first parameter is the string displayed as the anchor text (`'Home'`). The second parameter is the route. In this case, the route `root_path` has been set in the **config/routes.rb** file. The second route, `page_path('about')` , is set automatically by the high_voltage gem.

We can add the navigation links partial to our application layout with the expression:

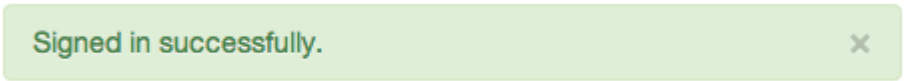
```
<%= render 'layouts/navigation' %>
```

We'll add this to the application layout at the end of this chapter.

Flash Messages

Rails provides a standard convention to display alerts (including error messages) and other notices (including success messages), called a *flash message_. The name comes from the term "flash memory" and should not be confused with the "Adobe Flash" web development platform that was once popular for animated websites. The flash message is documented in the* RailsGuides: Action Controller Overviewoverview.html#the-flash.

Here's a flash message you might see after logging in to an application:



It is called a "flash message" because it appears on a page temporarily. When the page is reloaded or another page is visited, the message disappears.

Flash messages are created in a controller. For example, we could add messages to the home page by modifying the VisitorsController like this:

```
class VisitorsController < ApplicationController

  def new
    flash[:notice] = 'Welcome!'
    flash[:alert] = 'My birthday is soon.'
  end

end
```

Rails provides the `flash` object so that messages can be created in the controller and displayed on the rendered web page.

Flash and Flash Now

You can control the persistence of the flash message by choosing from two variants of the Flash directive.

Use `flash.now` in the controller when you immediately render a page, for example with a `render :new` directive. With `flash.now`, the message will vanish after the user clicks any links.

Use the simple variant, `flash`, in the controller when you redirect to another page, for example with a `redirect_to root_path` directive. If you use `flash.now` before a redirect, the user will not see the flash message because `flash.now` does not persist through redirects or links. If you use the simple `flash` directive before a `render` directive, the message will appear on the rendered page and reappear on a subsequent page after the user clicks a link.

In our example above, we really need to use the `flash.now` variant because the controller provides a hidden `render` method.

```
class VisitorsController < ApplicationController

  def new
    @owner = Owner.new
    flash.now[:notice] = 'Welcome!'
    flash.now[:alert] = 'My birthday is soon.'
  end

end
```

Using `flash.now` will make sure the message only appears on the rendered page and will not persist after a user follows a link to a new page.

If this is confusing, don't worry. Just make a mental note that if you see a "sticky" flash message that won't go away, you need to use `flash.now` instead of `flash`.

Explaining the Ruby Code

If you're new to programming in Ruby, it may be helpful to learn how the `flash` object works.

The `flash` object is a Ruby *hash* (also known as an *associative array*). A hash is a data structure that associates a key to some value. You retrieve the value based upon its key. This construct is called a *dictionary* in other languages, which is appropriate because you use the key to "look up" a value, as you would look up a definition for a word in a dictionary.

In this example, we create a flash message by associating the key `flash[:notice]` with the string `'Welcome!'`. We can assign other messages, such as `flash[:alert]` or even `flash[:warning]`. In practice, Rails uses only `:notice` and `:alert` as flash message keys so it is wise to stick with just these.

Hash is a type of *collection*. In Ruby, all collections support an *iterator* method named `each`. Iterators return all the elements of a collection, one after the other. The iterator returns each key-value pair, item by item, to a *block*. In Ruby, a block is delimited by `do` and `end` or `{ }` braces. You can add any code to a block to process each item from the collection.

Here is simple Ruby code to iterate through a `flash` object, outputting each flash message in an HTML `div` tag and applying a CSS class for styling:

```
flash.each do |key, value|
  puts '<div class="' + key + '">' + value + '</div>'
end
```

In this simple example, we use `each` to iterate through the flash hash, retrieving a `key` and `value` that are passed to a block to be output as a string. We've chosen the variable names `key` and `value` but the names are arbitrary. In the next example, we'll use `name` and `msg` as variables for the key-value pair. The output string will appear as HTML like this:

```
<div class="notice">Welcome!</div>
<div class="alert">My birthday is soon.</div>
```

Create a Partial for Flash Messages

Code for flash messages can go directly in your application layout file or you can create a partial.

Add a partial for flash messages by creating a file **app/views/layouts/_messages.html.erb** like this:

```
<% flash.each do |name, msg| %>
  <% if msg.is_a?(String) %>
    <div class="alert alert-<%= name == :notice ? "success" : "error" %>">
      <a class="close" data-dismiss="alert">&#215;</a>
      <%= content_tag :div, msg, :id => "flash_#{name}" %>
    </div>
  <% end %>
<% end %>
```

We've improved on our simple Ruby example in several ways. First, we've added `if msg.is_a?(String)` as a test to make sure we only display messages that are strings. Second, we've used the Rails `content_tag` view helper to create the HTML `div`. The `content_tag` helper eliminates the messy soup of angle brackets and quote marks we used to create the HTML output in the example above. Finally, instead of applying a CSS `class` for styling, we've applied a CSS `id` and combined the word "flash" with "notice" or "alert" to make the CSS ID. It's appropriate to apply an ID instead of a class because there should be only be a single flash message on the page (classes are used for multiple elements).

Twitter Bootstrap provides a jQuery plugin named bootstrap-alert that makes it easy to dismiss flash messages with a click. The data-dismiss property displays an "x" that enables the close functionality. Note that Twitter Bootstrap uses the HTML entity "×" instead of the keyboard letter "x".

At the end of this chapter, we'll add the partial to the application layout.

Beyond Red and Green with Twitter Bootstrap

Rails uses `:notice` and `:alert` as flash message keys. Using the Twitter Bootstrap styling we've applied above, a "notice" appears in green and an "alert" in red.

Twitter Bootstrap provides a base class `.alert` with additional classes `.alert-success` and `.alert-error` (see the Bootstrap documentation on alerts). This is how Twitter Bootstrap applies a green background to the CSS class `.alert-success` and a red background to `.alert-error`.

Twitter Bootstrap also provides a third class `.alert-info` with a blue background. With a little hacking, it's possible to create a Rails flash message with a custom name, such as `:info`, that will display with the Bootstrap `.alert-info` class. However, it's wise to stick with the Rails convention of using only "alert" and "notice."

Using the code in the partial above, a Rails "notice" message will be styled with the Twitter Bootstrap "alert-success" style. Any other message, including a Rails "alert" message, will be styled with the Twitter Bootstrap "alert-error" style.

For an alternative that accommodates four different flash message types (success, error, alert, notice), see an example of Rails Flash Messages using Twitter Bootstrap.

Default Application Layout with Twitter Bootstrap

In the last chapter, we configured the default application layout with additional boilerplate and HTML5 elements. Now that we've installed Twitter Bootstrap, we can style the page with Bootstrap's built-in CSS components. You can choose from dozens of Bootstrap CSS components; we'll use the navigation components with the Bootstrap 12-column grid.

Replace the contents of the file **app/views/layouts/application.html.erb** with this:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title><%= content_for?(:title) ? yield(:title) : "Rails Bootstrap" %></title>
    <meta name="description" content="<%= content_for?(:description) ?
yield(:description) : "Rails Bootstrap" %>">
    <%= stylesheet_link_tag "application", :media => "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <header class="navbar navbar-fixed-top">
      <nav class="navbar-inner">
        <div class="container">
          <%= render 'layouts/navigation' %>
        </div>
      </nav>
    </header>
    <main role="main">
      <div class="container">
        <div class="content">
           <div class="row">
            <div class="span12">
              <%= render 'layouts/messages' %>
              <%= yield %>
            </div>
          </div>
          <footer>
          </footer>
        </div>
      </div>
    </main>
  </body>
</html>
```

We wrap the navigation partial `render 'layouts/navigation'` with the Bootstrap navigation classes.

We wrap both the flash messages partial `render 'layouts/messages'` and the main page content rendered by `yield` with Bootstrap `row` and `span` classes. We also apply the `content` class for the nice gray box around the main page content.

Test the Application

Let's see how the application looks with Twitter Bootstrap. The web server may already be running. If not, enter the command:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

The links are broken because we haven't yet created an "About" page. We'll add that next.

Git

Let's commit our changes to the Git repository.

```
$ git add -A
$ git commit -m "stylesheets"
$ git push origin master
```

Chapter 13

Adding Pages

Let's add a page to our web application.

Most websites contain pages such as:

- "About" page
- Contact page
- Legal page
- FAQ page

We call these "static pages" because the content does not vary (they would be "dynamic pages" if the content was obtained from a database or varied depending on a visitor's interaction).

It's possible to place these pages in the **public/** folder and copy the HTML and CSS from the default application layout but this leads to duplicated code and maintenance headaches. And dynamic elements such as navigation links can't be included. For these reasons, developers seldom create static pages in the **public/** folder.

We can use Rails to create a page that uses no model, a nearly-empty controller, and a view that contains no instance variables. This solution is implemented so frequently that many developers create a gem to encapsulate the functionality. We'll use the best-known of these gems to create an "About" Page, the high_voltage gem created by the Thoughtbot consulting firm.

HighVoltage Gem

We can add a page using the HighVoltage gem almost effortlessly. The gem implements Rails "convention over configuration" so well that there is nothing to configure. There are alternatives to its defaults which can be useful but we won't need them; visit the GitHub page for the high_voltage gem if you want to explore all the options.

In your **Gemfile**, you've already added:

```
gem 'high_voltage'
```

and previously run `$ bundle install`.

Views Folder

Create a folder **app/views/pages**:

```
$ mkdir app/views/pages
```

Any view files we add to this directory will automatically use the default application layout and appear when we use a URL that contains the filename.

The HighVoltage gem contains all the controller and routing magic required for this to happen.

Let's try it out.

"About" Page

Create a file **app/views/pages/about.html.erb**:

```
<% content_for :title do %>About<% end %>
<h3>About the Website</h3>
<p>Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna aliqua.</p>
```

Our simple "About" view will be combined with the default application layout by the HighVoltage gem.

We include a `content_for` Rails view helper that passes a page title to the application layout.

Routing for the HighVoltage Gem

The HighVoltage gem provides a PagesController. You'll never see it; it is packaged inside the gem.

In addition to providing a controller, the HighVoltage gem provides default routing so any URL with the form http://localhost:3000/pages/about will obtain a view from the **app/views/pages** directory.

Like the PagesController, the code that sets up the route is packaged inside the gem. If we wanted to add the route explicitly to the file **config/routes.rb**, the file would look like this:

```
LearnRails::Application.routes.draw do
  get "/pages/*id" => 'pages#show'
  root :to => 'visitors#new'
end
```

Again, you don't need to add the code above because the HighVoltage gem already provides the route.

You can use a Rails route helper to create a link to any view in the **app/views/pages** directory like this:

```
link_to 'About', page_path('about')
```

We already have a link to the "About" page in the navigation partial so we can test the application immediately.

Test the Application

The web server may already be running. If not, enter the command:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

The link to the "About" page should work.

Git

Let's commit our changes to the Git repository.

```
$ git add -A
$ git commit -m "add 'about' page"
$ git push origin master
```

There is nothing more we need for our "About" page.

At this point, you've built a basic starter application that integrates Twitter Bootstrap and Rails. You can use this starter application as a foundation for your own projects that require Twitter Bootstrap and Rails.

You can use the Rails Composer tool to generate the starter application. In less than a few minutes, the Rails Composer tool will automatically create the application you've just built. You'll be able to give it your own project name when you generate the app. See the instructions in the README for the rails-bootstrap example application.

Chapter 14

Exploring Bootstrap

Twitter Bootstrap is a rich toolkit for front-end design. Explore the Twitter Bootstrap documentation to see all the design elements that are available.

So far, our tutorial has shown how to integrate Twitter Bootstrap with Rails but we've only created a home page, an "About" page, and used Twitter Bootstrap to style a navigation bar. It's a good starter app you can use for any number of projects. But we haven't explored many of the design elements offered by Twitter Bootstrap.

In the chapters to come, we'll go beyond the starter application and implement a simple survey website. Features include:

- Graphic carousel
- Survey form
- Google spreadsheet datastore
- Modal window

With the survey website example, we'll only scratch the surface of all the possibilities offered by Twitter Bootstrap, but you'll see how a few interesting design elements are added to your starter application, making it easier to add others on your own.

Additional Gems

We'll need additional gems to implement our example. These gems are not needed in the starter application, so we didn't add them earlier. But we'll add them now so we can learn more about Twitter Bootstrap.

Here are gems we'll add to the Gemfile:

- activerecord-tableless – makes Rails easier without a database
- google_drive – use Google Drive spreadsheets for data storage

Open your **Gemfile** and replace the contents with the following:

Gemfile

```
source 'https://rubygems.org'
ruby '2.0.0'
gem 'rails', '4.0.0'

# Rails defaults
gem 'sqlite3'
gem 'sass-rails', '~> 4.0.0'
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.0.0'
gem 'jquery-rails'
gem 'turbolinks'
gem 'jbuilder', '~> 1.2'

# rails-bootstrap
gem 'activerecord-tableless'
gem 'bootstrap-sass'
gem 'figaro'
gem 'google_drive'
gem 'high_voltage'
gem 'simple_form', '>= 3.0.0.rc'
group :development do
  gem 'better_errors'
  gem 'quiet_assets'
end
```

Check for the current version of Rails and replace `gem 'rails', '4.0.0'` accordingly. The Rails default gems may have changed as well.

Install the Gems

Now that you've edited the Gemfile, run `bundle install`:

```
$ bundle install
```

You can check which gems are loaded into the development environment:

```
$ gem list
```

You'll need to restart your web server before you can use the new gems.

Git

Let's commit our changes to the Git repository.

```
$ git add -A
$ git commit -m "add gems"
$ git push origin master
```

We're ready to configure the application.

Configuration File

To implement our survey website and save data submitted by the visitors, we'll need to configure an account name and password for Google Drive access.

We've already installed the figaro gem in the Gemfile and run `bundle install`.

Earlier, using the `rails generate` command, we set up a configuration file. You should already have a file named **config/application.yml**. If not, you'll need to set it up:

```
$ rails generate figaro:install
```

The figaro gem generates a **config/application.yml** file and modifies the **.gitignore** file. The **.gitignore** file prevents the **config/application.yml** file from being saved in the Git repository so your credentials are kept private.

We'll set credentials for your Google account in the **config/application.yml** file.

Use your text editor to replace the file **config/application.yml** with this:

```
# Add account credentials and API keys here.
# See http://railsapps.github.io/rails-environment-variables.html
# This file should be listed in .gitignore to keep your settings secret!
# Each entry sets a local environment variable and overrides ENV variables in the Unix
shell.
GMAIL_USERNAME: changeme
GMAIL_PASSWORD: changeme
```

Replace the placeholders in the **config/application.yml** file with real credentials.

If you have a Gmail account, your username and password are the credentials you use to access Google Drive. If you don't have a Gmail account, you can sign up for a Google account for free.

Git

Let's commit our changes to the Git repository.

```
$ git add -A
$ git commit -m "add configuration"
$ git push origin master
```

We're ready to improve the application home page.

Chapter 15

Carousel

A *carousel* is a design element that is ideal for displaying a series of images.

The carousel is an impressive effect that can be implemented in a few lines of code using the Twitter Bootstrap JavaScript library.

We've already created a home page with a Visitors controller and a view file, as well as routing. In this chapter, we'll modify the Visitors#New view file to add an image carousel.

User Story

We'll plan our work with a user story:

```
*Home page*
As a visitor to the website
I want to see a slideshow of images
so I can pick my favorite
```

In the next chapter we'll add a survey form for the visitor to vote on favorite images.

Add an Image Carousel

The Twitter Bootstrap JavaScript library provides more than a dozen design elements that can be implemented in a few lines of HTML. You can see the documentation for JavaScript in Bootstrap. One of the elements is the Bootstrap Carousel.

Adding the carousel is as easy as copying the Bootstrap example and adding images.

Replace the contents of the file **app/views/visitors/new.html.erb**:

```
<% content_for :title do %>Rails Bootstrap<% end %>
<% content_for :description do %>Rails Bootstrap Example<% end %>
<div id="myCarousel" class="carousel slide">
  <ol class="carousel-indicators">
    <li data-target="#myCarousel" data-slide-to="0" class="active"></li>
    <li data-target="#myCarousel" data-slide-to="1"></li>
    <li data-target="#myCarousel" data-slide-to="2"></li>
  </ol>
  <div class="carousel-inner">
    <div class="item active">
      <img src="http://lorempixel.com/1170/800/city/9">
      <div class="carousel-caption text-center">
        <h4>San Francisco</h4>
      </div>
    </div>
    <div class="item">
      <img src="http://lorempixel.com/1170/800/city/7">
      <div class="carousel-caption text-center">
        <h4>Sydney</h4>
      </div>
    </div>
    <div class="item">
      <img src="http://lorempixel.com/1170/800/city/6">
      <div class="carousel-caption text-center">
        <h4>Paris</h4>
      </div>
    </div>
  </div>
  <a class="left carousel-control" href="#myCarousel" data-slide="prev">&lsaquo;</a>
  <a class="right carousel-control" href="#myCarousel" data-slide="next">&rsaquo;</a>
</div>
```

We include `content_for` view helpers that pass a title and description to the application layout.

We add photos using the `<img>` tag. We're taking a shortcut and using placeholder photos from the lorempixel.com service.

Each photo has a caption set by the `<h4>` tag and the `<div class="carousel-caption">` style. We've added the `text-center` class to the example code we copied from the Bootstrap example code.

The best way to understand the code is to remove elements and refresh the page to see the changes. Removing the `<ol class="carousel-indicators">` section will remove the little dots in the upper right corner of the image that provide image selection.

You can add another image by adding a `<div class="item">` section. Additional placeholder images are available by changing the last number in the image URL.

You can remove the navigation arrows by removing the section:

```
<a class="left carousel-control" href="#myCarousel" data-slide="prev">&lsaquo;</a>
<a class="right carousel-control" href="#myCarousel" data-slide="next">&rsaquo;</a>
```

Photo Options

You might wish to modify the placeholder photo. If you don't like the photos I've selected, try http://lorempixel.com/1170/600/cats/1 or any other categories from the lorempixel.com service. You can change the size by modifying the dimensions from 1170 (pixel width) by 800 (pixel height).

You can replace the placeholder photo with your own by creating an **app/assets/images** folder and adding images. Instead of the HTML `<img>` tag, use the Rails `image_tag` view helper, like this:

```
<%= image_tag "myphoto.jpg" %>
```

Test the Application

The web server may already be running. If not, enter the command:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

You should see the image carousel.

Git

Let's commit our changes to the Git repository.

```
$ git add -A
$ git commit -m "home page carousel"
$ git push origin master
```

In the next chapter, we'll add the survey form.

Chapter 16

Survey Form

Aside from links, forms are the most common way for users to interact with web applications. The combination of Rails form handling and Bootstrap form styling greatly improves the website user experience.

In this chapter we'll add a survey form to the home page. Our primary purpose is to explore how Twitter Bootstrap applies styling to form elements.

You'll recall that we set up the SimpleForm gem when we added Twitter Bootstrap to our application. The SimpleForm gem gives us view helpers to generate the HTML required by forms. Forms contain some of the most complex HTML a developer will encounter, so the SimpleForm gem is truly worthwhile. You can see a demonstration of SimpleForm and Twitter Bootstrap for all the possible form elements.

User Story

Let's plan our work with a user story:

```
*Survey Form*
As a visitor to the website
I want to select a favorite image
And submit a comment
so my opinion is known
```

There are many ways to implement a selection system, such as clicking a "like" icon or selecting from a "star" matrix. We'll simply implement a form with a set of radio buttons and a comment field.

Visitor Model

We'll need a Visitor model before we create the form. Strictly speaking, we could create a form without a model, but the model gives us the benefits of Rails form validation provided by the ActiveRecord API.

Create a file **app/models/visitor.rb**:

```
class Visitor < ActiveRecord::Base
  has_no_table
  column :favorite, :string
  column :comment, :string
  validates_presence_of :favorite

  IMAGE_LABELS = ['San Francisco', 'Sydney', 'Paris']

end
```

We inherit behavior from the ActiveRecord parent class. We are not using a database for our tutorial application, so we use the `has_no_table` keyword from the activerecord-tableless gem to disable the database features of ActiveRecord.

Note: There's another way to create a model without a database using only the ActiveModel class, described in the RailsCasts: ActiveModel screencast. Either approach is fine; we're using the activerecord-tableless gem because a tableless implementation using ActiveModel requires an understanding of Ruby modules, get and set methods, and object initialization. It's easier to use the activerecord-tableless gem.

We create attributes for `favorite` and `comment`.

Validation requirements for our survey form are simple. We want to make sure a "favorite" is selected and we don't care if a comment is missing.

Finally, we create a constant named `IMAGE_LABELS` that returns an array containing a name for each image the visitor will see on the home page. We'll use this array in several places, to provide a caption for each image, and to populate a collection of radio buttons, so it makes sense to keep our code DRY (*Don't Repeat Yourself*) by setting a constant in the model. There are other ways to approach this, including hard coding the variables in the view, or setting an instance variable in the controller, but the preferred Rails approach is to define data values in a model.

By convention, Ruby constants are styled in uppercase with underscores to separate words. To obtain the constant value elsewhere in our code, we'll use the syntax `Visitor::IMAGE_LABELS`.

Modify the Visitors Controller

We already have a Visitors controller that contains a simple `new` method. We'll change the `new` method, add a `create` method, and provide a `secure_params` private method to secure the controller from mass assignment exploits.

Replace the contents of the file **app/controllers/visitors_controller.rb**:

```
class VisitorsController < ApplicationController

  def new
    @visitor = Visitor.new
  end

  def create
    @visitor = Visitor.new(secure_params)
    if @visitor.valid?
      flash[:notice] = "Chose #{@visitor.favorite}."
      render :new
    else
      render :new
    end
  end

  private

  def secure_params
    params.require(:visitor).permit(:favorite, :comment)
  end

end
```

The controller `new` action will instantiate an empty Visitor model, assign it to the `@visitor` instance variable, and render the **app/views/visitors/new.html.erb** view.

SimpleForm will set a destination URL that corresponds to the VisitorsController#create action. The `create` method will instantiate a new Visitor model using the data from the form (we take steps to avoid security vulnerabilities first—more on that below).

The ActiveRecord parent class provides a method `valid?` which we can call on the Visitor model. Our conditional statement `if @visitor.valid?` checks the validation requirements we've set in the model.

If the validation check succeeds, we set a flash notice and redisplay the home page.

If the the validation check fails, we redisplay the home page and the form will highlight errors.

Mass-Assignment Vulnerabilities

Rails protects us from a class of security exploits called "mass-assignment vulnerabilities." Rails won't let us initialize a model with just any parameters submitted on a form. Suppose we were creating a new user and one of the user attributes was a flag allowing administrator access. A malicious hacker could create a fake form that provides a user name and sets the administrator status to "true." Rails forces us to "white list" each of the parameters used to initialize the model.

We create a private method named `secure_params` to screen the parameters sent from the browser. The `params` hash contains two useful methods we use for our screening:

- `require(:visitor)` – makes sure that `params[:visitor]` is present
- `permit(:favorite, :comment)` – our "white list"

With this code, we make sure that `params[:visitor]` only contains `::favorite, :comment)`. If other parameters are present, they are stripped out. Rails will raise an error if a controller attempts to pass params to a model method without explicitly permitting attributes via `permit`.

For our simple survey form, we don't need to be concerned with mass-assignment vulnerabilities, as there is nothing to compromise. But our example follows Rails recommended practice because you may decide to adapt the form if you use the tutorial application as a starter app for your own project.

Add a Form to the Home Page

Earlier we built a home page that contained an image carousel. We'll add a form to the home page so a visitor can vote for a favorite image.

Replace the contents of the file **app/views/visitors/new.html.erb**:

```
<% content_for :title do %>Rails Bootstrap<% end %>
<% content_for :description do %>Rails Bootstrap Example<% end %>
<div id="myCarousel" class="carousel slide">
  <ol class="carousel-indicators">
    <li data-target="#myCarousel" data-slide-to="0" class="active"></li>
    <li data-target="#myCarousel" data-slide-to="1"></li>
    <li data-target="#myCarousel" data-slide-to="2"></li>
  </ol>
  <div class="carousel-inner">
    <div class="item active">
      <img src="http://lorempixel.com/1170/800/city/9">
      <div class="carousel-caption text-center">
        <h4><%= Visitor::IMAGE_LABELS[0] %></h4>
      </div>
    </div>
    <div class="item">
      <img src="http://lorempixel.com/1170/800/city/7">
      <div class="carousel-caption text-center">
        <h4><%= Visitor::IMAGE_LABELS[1] %></h4>
      </div>
    </div>
    <div class="item">
      <img src="http://lorempixel.com/1170/800/city/6">
      <div class="carousel-caption text-center">
        <h4><%= Visitor::IMAGE_LABELS[2] %></h4>
      </div>
    </div>
  </div>
  <a class="left carousel-control" href="#myCarousel" data-slide="prev">&lsaquo;</a>
  <a class="right carousel-control" href="#myCarousel" data-slide="next">&rsaquo;</a>
</div>
<div class="text-center">
  <h5>Select a Favorite</h5>
  <%= simple_form_for @visitor do |f| %>
    <%= f.error_notification %>
    <%= f.input :favorite, :collection => Visitor::IMAGE_LABELS, label: false, as:
:radio_buttons, :item_wrapper_class => 'inline' %>
    <%= f.input :comment, label: false, :placeholder => 'Add a comment...' %>
    <%= f.submit "Choose!", :class => "btn btn-primary"%>
  <% end %>
</div>
```

We've added a Visitors model so we obtain the caption for each image from the `IMAGE_LABELS` constant we've created in the Visitors model. The syntax `Visitor::IMAGE_LABELS[0]` gives us the value for the first item in the `IMAGE_LABELS` array (Ruby counts array items starting from zero).

We add a div and apply the Boostrap `text-center` class for our form. We set a `<h5>` headline above the form.

The code for the form is compact but complex. We see several elements:

- `simple_form_for` is the view helper for the form

The `simple_form_for` view helper instantiates a form object which we assign to a variable named `f`. SimpleForm offers many standard form elements, such as text fields and submit buttons. Each element is available as a method call on the form object.

The view helper `simple_form_for` requires *parameters* and a *block*.

We initialize the form with the `@visitor` instance variable. SimpleForm uses this parameter to name the form, set a destination for the form data (the VisitorsController#create action), and initialize each field in the form using attributes from the Visitor model. Setting the values for the form fields from the attributes in the model is called "binding the form to the object" and you can read about it in the RailsGuides: Form Helpers article.

The `simple_form_for` view helper accommodates a Ruby block. The block begins with `do` and closes with `end`.

Inside the block, the `form` object methods generate HTML for:

- error notifications
- radio buttons
- a field for "comment"
- a submit button

Each of the form methods takes various parameters.

The first input field creates radio buttons from parameters:

- `:favorite` – the field name matching an attribute in the Visitor model
- `:collection => Visitor::IMAGE_LABELS` – choices from an array in the Visitor model
- `label: false`, `as: :radio_buttons` – don't display a label for the field
- `:item_wrapper_class => 'inline'` – display the choices horizontally

The second input field creates a comment field from parameters:

- `:comment` – the field name matching an attribute in the Visitor model
- `label: false` – don't display a label for the field
- `:placeholder` – a user prompt

The final method creates a submit button, setting the text to display in the button and applying the Bootstrap classes `btn btn-primary`.

The structure of the form is clearly visible in the code. The form begins with a `simple_form_for` helper and closes with the `end` keyword. Each line of code produces an element in the form such as a field or a button.

Routing

We'll modify the routing to accommodate the new controller actions.

In addition to rendering the Visitors#new view as the application root (the home page), we need to handle the `create` action. We can use Rails resourceful routing to define two new routes in a single line of code.

We'll use two of the seven "resourceful" controller actions:

- *new* – display the home page with the empty visitor form
- *create* – validate and process the submitted form

Open the file **config/routes.rb**. Replace the contents with this:

```
RailsBootstrap::Application.routes.draw do
  resources :visitors, only: [:new, :create]
  root :to => 'visitors#new'
end
```

The root path remains `visitors#new`. Order is significant in the **config/routes.rb** file. As the final designated route, the root path will only be active if nothing above it matches the route.

We've added `visitors :contacts, only: [:new, :create]`.

We only want two routes so we've added the restriction `only: [:new, :create]`.

The `new` route has these properties:

- `new_visitor_path` – route helper
- `visitors` – name of the controller (VisitorsController)
- `new` – controller action
- http://localhost:3000/visitors/new – URL generated by the route helper
- `GET` – HTTP method to display a page

The `create` route has these properties:

- `visitors_path` – route helper

- `visitors` – name of the controller (VisitorsController)
- `create` – controller action
- http://localhost:3000/visitors – URL generated by the route helper
- `POST` – HTTP method to submit form data

You can run the `rake routes` command to see these in the console:

```
$ rake routes
     Prefix Verb URI Pattern             Controller#Action
   visitors POST /visitors(.:format)     visitors#create
new_visitor GET  /visitors/new(.:format) visitors#new
       root GET  /                       visitors#new
       page GET  /pages/*id              high_voltage/pages#show
```

The output of the `rake routes` command shows we've created the routes we need.

Test the Application

You've modified the routing, so stop the web server with Control-c and restart:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

You'll see our new home page with the survey form.

Git

Let's commit our changes to the Git repository.

```
$ git add -A
$ git commit -m "add form to home page"
$ git push origin master
```

Next, let's implement a feature to save data to a spreadsheet.

Chapter 17

Spreadsheet Connection

In the last chapter, we added a survey form to the application home page. When a visitor submits the form, we display an acknowledgment message. To complete the implementation, we need a way to capture the data for later analysis or review.

We've purposely chosen not to implement a database application so we can keep this tutorial to a manageable length and focus on Twitter Bootstrap. Though many Rails applications are backed by databases, a database adds complexity to a Rails application. One of the requirements that adds complexity is authentication and authorization. If data is stored in an application database, we have to implement access control so only an administrator can view it.

Fortunately, Google Drive (formerly known as Google Docs) gives us an easy way to store and access our visitor data without a database. It's an elegant solution. We can use the Google Drive API (application programming interface) to save form data to a spreadsheet that is stored in Google Drive. We don't have to implement authentication and authorization in our application because Google Drive already manages user access. Our application will send the data to a spreadsheet and you (or colleagues or clients, if you wish) can access the data on Google Drive.

Most computer-literate people have some experience with spreadsheets. Making data available in a spreadsheet makes it easy for an administrator or a website owner to analyze or review the data.

Google Drive Gem

We'll use the google_drive gem to connect to the spreadsheet and save data.

The google_drive gem is a Ruby library to read and write data to spreadsheets in Google Drive. It provides convenient Ruby methods to wrap the Google Spreadsheets API. You can see all the features of the google_drive gem by reviewing the google_drive gem API.

In your **Gemfile**, you've already added:

```
gem 'google_drive'
```

and previously run `$ bundle install`.

Implementation

The code that saves data to the spreadsheet could be added to the controller. After all, the form data is received by the controller. It would be easy to add a private method to the controller that sends the data to the spreadsheet. In practice, Rails developers prefer to add code that manipulates data to the model.

Controllers should contain only enough code to instantiate a model and handle rendering. In principle, all data manipulation should be handled by a model. Saving to a spreadsheet is a data operation.

Modify the Visitor Model

In keeping with the Rails mantra, "skinny controller, fat model," we'll add our spreadsheet code to the Visitor model.

Replace the contents of the file **app/models/visitor.rb**:

```
class Visitor < ActiveRecord::Base
  has_no_table
  column :favorite, :string
  column :comment, :string
  validates_presence_of :favorite

  IMAGE_LABELS = ['San Francisco', 'Sydney', 'Paris']

  def update_spreadsheet
    connection = GoogleDrive.login(ENV["GMAIL_USERNAME"], ENV["GMAIL_PASSWORD"])
    ss = connection.spreadsheet_by_title('Rails-Bootstrap-Example')
    if ss.nil?
      ss = connection.create_spreadsheet('Rails-Bootstrap-Example')
    end
    ws = ss.worksheets[0]
    last_row = 1 + ws.num_rows
    ws[last_row, 1] = Time.now
    ws[last_row, 2] = self.favorite
    ws[last_row, 3] = self.comment
    ws.save
  end

end
```

We'll call the new `update_spreadsheet` method from the controller.

The google_drive gem gives us a `GoogleDrive` object.

Create a connection to Google Drive by passing your credentials to the `login` method. Here's where we use the environment variables we set in the **config/application.yml** file using the figaro gem.

We look for a spreadsheet named "Rails-Bootstrap-Example." The first time we attempt to save data, the spreadsheet will not exist, so we use the `create_spreadsheet` method to create it. If it already exists, the `spreadsheet_by_title` method will find it.

A single spreadsheet file can contain multiple *worksheets*. We'll use only one worksheet to store our data, designated as "worksheet 0" (we count from zero).

Here the code gets a little tricky. You might expect the API to provide an "append row" method. In fact, we have to retrieve a count of rows, and then add one, to calculate the row number of the last empty row.

We add data on a cell-by-cell basis, by designating the row number and column number of a cell. We add the current date and time using the Ruby API method `Time.now` to the first cell in the last row. Then we add the `favorite` and `comment` attributes to the second and third columns (we refer to the current instance of the class by using the keyword "self").

Setting the cell value doesn't save the data. We explicity call the worksheet `save` method to update the worksheet.

Modify the Visitors Controller

Our Visitor model now has a method to save data to a spreadsheet.

We'll update the Visitors controller to save the data.

Replace the contents of the file **app/controllers/visitors_controller.rb**:

```
class VisitorsController < ApplicationController

  def new
    @visitor = Visitor.new
  end

  def create
    @visitor = Visitor.new(secure_params)
    if @visitor.valid?
      @visitor.update_spreadsheet
      flash[:notice] = "Chose #{@visitor.favorite}."
      render :new
    else
      render :new
    end
  end

  private

  def secure_params
    params.require(:visitor).permit(:favorite, :comment)
  end

end
```

We've added only one line, the `@visitor.update_spreadsheet` statement.

When the visitor submits the form, the VisitorsController#create action is called. The `create` method will instantiate a new Visitor model using the data from the form after laundering the parameters. If the validation check succeeds, we save data to the spreadsheet, set a flash notice, and redisplay the home page.

In only a few lines of code, we've added data storage using Google Drive.

Test the Application

Make sure the web server is running:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

You'll see the home page with the survey form.

Fill in the form by selecting a radio button, adding a comment, and clicking the submit button. You'll see an acknowledgment message.

Visit your Google Drive account ("Drive" is in the navigation bar when you visit the Gooogle Search or Gmail home pages). You'll see a list of Google Drive files. The newest one will be a **Rails-Bootstrap-Example** spreadsheet. Open the file and you will see the data from the survey form. Whenever a visitor submits the survey form, the spreadsheet will update within seconds.

Git

Let's commit our changes to the Git repository.

```
$ git add -A
$ git commit -m "save data to a spreadsheet"
$ git push origin master
```

We've got a fully functional survey website that stores data in a Google Drive spreadsheet.

We could deploy the application now. But let's explore another Twitter Bootstrap design element, the modal window.

Chapter 18

Modal Window

The functionality of the survey form is complete. Let's add another Twitter Bootstrap design element.

We can use Twitter Bootstrap to hide the survey form and reveal it when a button is clicked. We'll use a lightbox effect: The form will appear in a bright overlay with the page darkened behind it. Because no further action can be taken until the form is submitted or cancelled, we say the form appears in a "modal window." A modal window is a user interface element in desktop applications and on the web, commonly used to block action until the user responds to the message in the window.

Add a Modal Window

Our "call to action" will be a big button that says, "Choose!" Clicking the button will open a modal window and invite the visitor to select a favorite and click a submit button.

Twitter Bootstrap gives us everything we need to implement this in a few lines of code.

Replace the contents of the file **app/views/visitors/new.html.erb**:

```
<% content_for :title do %>Rails Bootstrap<% end %>
<% content_for :description do %>Rails Bootstrap Example<% end %>
<div id="myCarousel" class="carousel slide">
  <ol class="carousel-indicators">
    <li data-target="#myCarousel" data-slide-to="0" class="active"></li>
    <li data-target="#myCarousel" data-slide-to="1"></li>
    <li data-target="#myCarousel" data-slide-to="2"></li>
  </ol>
  <div class="carousel-inner">
    <div class="item active">
      <img src="http://lorempixel.com/1170/800/city/9">
      <div class="carousel-caption text-center">
        <h4><%= Visitor::IMAGE_LABELS[0] %></h4>
      </div>
    </div>
    <div class="item">
      <img src="http://lorempixel.com/1170/800/city/7">
      <div class="carousel-caption text-center">
        <h4><%= Visitor::IMAGE_LABELS[1] %></h4>
      </div>
    </div>
    <div class="item">
      <img src="http://lorempixel.com/1170/800/city/6">
      <div class="carousel-caption text-center">
        <h4><%= Visitor::IMAGE_LABELS[2] %></h4>
      </div>
    </div>
  </div>
  <a class="left carousel-control" href="#myCarousel" data-slide="prev">&lsaquo;</a>
  <a class="right carousel-control" href="#myCarousel" data-slide="next">&rsaquo;</a>
</div>
<div class="text-center">
  <h5>Select a Favorite</h5>
  <div id="modal-form" class="modal" style="display: <%= @visitor.errors.any? ? 'block'
: 'none';%>">
    <%= simple_form_for @visitor do |f| %>
      <div class="modal-header">
        <a class="close" data-dismiss="modal">&#215;</a>
        <h3>Select a Favorite</h3>
      </div>
      <div class="modal-body">
        <%= f.error_notification %>
        <%= f.input :favorite, :collection => Visitor::IMAGE_LABELS, label: false, as:
:radio_buttons, :item_wrapper_class => 'inline' %>
        <%= f.input :comment, label: false, :placeholder => 'Add a comment...' %>
      </div>
      <div class="modal-footer">
        <%= f.submit "Choose!", :class => "btn btn-primary" %>
        <a class="btn" data-dismiss="modal" href="#">Close</a>
```

```
      </div>
    <% end %>
  </div>
  <div id="call-to-action">
    <a class="btn btn-primary btn-large" data-toggle="modal"
href="#modal-form">Choose!</a>
  </div>
</div>
```

The additional code includes HTML and CSS classes from Twitter Bootstrap that implement the modal window and sets up a button to reveal the hidden modal window.

Let's look at the button that reveals the window:

```
<div id="call-to-action">
  <a class="btn btn-primary btn-large" data-toggle="modal" href="#modal-form">Vote!</a>
</div>
```

Clicking the button reveals a form with the ID `#modal-form` because the attribute `data-toggle="modal"` is applied to the link.

It doesn't matter where the code for the form is placed in the file as the form will be hidden when the page is initially displayed and will appear as an overlay when revealed by a click on the appropriate link. For convenience, I've appended the code at the end of the file.

Let's look closely at the div that encloses the form:

```
<div id="modal-form" class="modal" style="display: <%= @visitor.errors.any? ? 'block' :
'none';%>">
```

We wrap the form in a div named `modal-form`. The name can be anything but it must match the `href="#modal-form"` found in the button that reveals the modal window.

The div named `modal-form` contains some complex code to handle validation errors. Normally, the form will be hidden when the page is rendered in the browser. That's fine for a first visit. But our Rails controller may detect a validation error after the visitor submits the form, in which case it will redisplay the page. This is a special case where we want the model window to be forced open as soon as the page is rendered. We use the Ruby ternary operator as a fancy conditional statement that says, "if the `@visitor` object contains errors, display the div as a 'block', otherwise, set the display property to 'none'."

Continuing our code inspection:

```
<div class="modal-header">
  <a class="close" data-dismiss="modal">&#215;</a>
  <h3>Select Your Favorite</h3>
</div>
```

The form has a section named “modal-header” that contains a link to close the modal window. We follow Twitter Bootstrap’s example and use HTML entity #215 (an “x” character) for the link.

The next section is named “modal-body” and contains the form input fields.

Following the input fields is a section named “modal-footer” that contains a submit button and another link to close the modal window:

```
<div class="modal-footer">
  <%= f.submit "Vote!", :class => "btn btn-primary" %>
  <a class="btn" data-dismiss="modal" href="#">Close</a>
</div>
```

You can use similar code for a modal window for other projects by duplicating this structure.

You can read the Bootstrap documentation for more information about the Bootstrap modal.

Test the Application

Make sure the web server is running:

```
$ rails server
```

Open a web browser window and navigate to http://localhost:3000/.

You’ll see the home page but the survey form will be hidden.

Click the big blue button and the modal window will be revealed.

You can fill out and submit the form with the same result as before.

Git

Let’s commit our changes to the Git repository.

```
$ git add -A
$ git commit -m "add modal to home page"
$ git push origin master
```

We've explored design elements provided by Twitter Bootstrap and seen how we can integrate the Bootstrap code with Rails. We've used two elements, the carousel and the modal, that are suitable for our simple survey website. Bootstrap has many other design elements that you can read about in the Bootstrap documentation.

Our survey website is complete. Let's deploy it.

Chapter 19

Deploy

Heroku provides the best known and most popular hosting for Rails applications. Using Heroku or another *platform-as-a-service* provider means you'll have experts maintaining the production environment, tuning system performance, and keeping the servers running.

Heroku is ideal for hosting our application:

- no system administration expertise is required
- hosting is free
- performance is excellent

Our Rails and Heroku article goes into more detail, describing costs and deployment options.

Let's deploy!

Test the Application

We haven't used test-driven development to build this application so no test suite is available. You've tested the application manually at each stage.

Preparing for Heroku

You'll need to prepare your Rails application for deployment to Heroku.

Gemfile

We need to modify the Gemfile for Heroku.

We add a `group :production` block for gems that Heroku needs:

- pg – PostgreSQL gem
- thin – web server
- rails_12factor – logging and static assets

Heroku doesn't support the SQLite database; the company provides a PostgreSQL database. Though we won't need it for our tutorial application, we must include the PostgreSQL gem for Heroku. We'll mark the sqlite3 gem to be used in development only.

The Thin web server is easy to use and requires no configuration. Note that Heroku recommends Unicorn for handling higher levels of traffic efficiently. Unicorn can be difficult to setup and configure, so we're using Thin for our tutorial application.

On Heroku, Rails 4.0 needs an extra gem to handle logging and serve CSS and JavaScript assets. The rails_12factor gem provides these services.

Open your **Gemfile** and replace the contents with the following:

Gemfile

```
source 'https://rubygems.org'
ruby '2.0.0'
gem 'rails', '4.0.0'

# Rails defaults
gem 'sass-rails', '~> 4.0.0'
gem 'uglifier', '>= 1.3.0'
gem 'coffee-rails', '~> 4.0.0'
gem 'jquery-rails'
gem 'turbolinks'
gem 'jbuilder', '~> 1.2'

# rails-bootstrap
gem 'activerecord-tableless'
gem 'bootstrap-sass'
gem 'figaro'
gem 'google_drive'
gem 'high_voltage'
gem 'simple_form', '>= 3.0.0.rc'
group :development do
  gem 'sqlite3'
  gem 'better_errors'
  gem 'quiet_assets'
end
group :production do
  gem 'pg'
  gem 'thin'
  gem 'rails_12factor'
end
```

We have to run `bundle install` because we've changed the Gemfile. The gems we've added are only needed in production so we don't install them on our local machine. When we deploy, Heroku will read the Gemfile and install the gems in the production environment.

We'll run `bundle install` with the `--without production` argument so we don't install the new gems locally:

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

Commit your changes to the Git repository:

```
$ git add -A
$ git commit -m "gems for Heroku"
$ git push origin master
```

Precompile Assets

In development mode, the Rails asset pipeline "live compiles" all CSS and JavaScript files and makes them available for use. Compiling assets adds processing overhead. In production, a web application would be slowed unneccesarily if assets were compiled for every web request. Consequently, we must precompile assets before we deploy our application to production.

When you precompile assets for production, the Rails asset pipeline will automatically produce concatenated and minified **application.js** and **application.css** files from files listed in the manifest files **app/assets/javascripts/application.js** and **app/assets/stylesheets/application.css.scss**. You must commit the compiled files to your git repository before deploying.

Here's how to precompile assets and commit to the Git repo:

```
$ RAILS_ENV=production rake assets:precompile
$ git add -A
$ git commit -m "assets compiled for Heroku"
$ git push origin master
```

The result will be several files added to the **public/assets/** folder. The filenames will contain a long unique identifier that prevents caching when you change the application CSS or JavaScript.

If you don't precompile assets for production, all web pages will look strange. They won't use the Twitter Bootstrap CSS styling.

Option to Ban Spiders

Do you want your website to show up in Google search results? If there's a link anywhere on the web to your site, within a few days (sometimes hours) the Googlebot spider will visit

your site and add it to the database for the Google search engine. Most webmasters want their sites to be found in Google search results. If that's not what you want, you may want to add the file **public/robots.txt** to prevent indexing by search engines.

Only add this file if you want to prevent your website from appearing in search engine listings:

```
# public/robots.txt
# To allow spiders to visit the entire site comment out the next two lines:
User-Agent: *
Disallow: /
```

You can learn more about the format of the robots exclusion standard.

Humans.txt

Many websites include a **robots.txt** file for nosy bots so it's only fair that you offer a **humans.txt** file for nosy people. Few people will look for it but you can add a file **public/humans.txt** to credit and identify the creators and software behind the website. The HTML5 Boilerplate project offers an example file or you can borrow from RailsApps.

Sign Up for a Heroku Account

In the chapter, "Accounts You May Need," I suggested you sign up for a Heroku account.

To deploy an app to Heroku, you must have a Heroku account. Visit https://id.heroku.com/signup/devcenter to set up an account.

You'll use a `git push` command to deploy your application to Heroku.

Be sure to use the same email address you used to configure Git locally. You can check the email address you used for Git with:

```
$ git config --get user.email
```

Heroku Toolbelt

Heroku provides a command line utility for creating and managing Heroku apps.

Visit https://toolbelt.heroku.com/ to install the Heroku Toolbelt. A one-click installer is available for Mac OS X, Windows, and Linux.

The installation process will install the Heroku command line utility. It also installs the Foreman gem which is useful for duplicating the Heroku production environment on a local machine. The installation process will also make sure Git is installed.

To make sure the Heroku command line utility is installed, try:

```
$ heroku version
heroku-toolbelt/...
```

You'll see the heroku-toolbelt version number.

You should be able to login using the email address and password you used when creating your Heroku account:

```
$ heroku login
Enter your Heroku credentials.
Email: adam@example.com
Password:
Could not find an existing public key.
Would you like to generate one? [Yn]
Generating new SSH public key.
Uploading ssh public key /Users/adam/.ssh/id_rsa.pub
```

The Heroku command line utility will create SSH keys if necessary to guarantee a secure connection to Heroku.

Heroku Create

Be sure you are in your application root directory and you've committed the tutorial application to your Git repository.

Use the Heroku create command to create and name your application.

```
$ heroku create myapp
```

Replace `myapp` with something unique. Heroku demands a unique name for every hosted application. If it is not unique, you'll see an error, "name is already taken." Chances are, "rails-bootstrap" is already taken.

If you don't specify your app name (`myapp` in the example above), Heroku will supply a placeholder name. You can easily change Heroku's placeholder name to a name of your choice with the `heroku apps:rename command` (see Renaming Apps from the CLI).

Don't worry too much about getting the "perfect name" for your Heroku app. The name of your Heroku app won't matter if you plan to set up your Heroku app to use your own domain name. You'll just use the name for access to the instance of your app running on the Heroku servers; if you have a custom domain name, you'll set up DNS (*domain name service*) to point your domain name to the app running on Heroku.

The `heroku create` command sets your Heroku application as a Git remote repository. That means you'll use the `git push` command to deploy your application to Heroku.

Next we'll set Heroku environment variables.

Set Heroku Environment Variables

You'll need to set the configuration values from the **config/application.yml** file as Heroku environment variables.

With the figaro gem, just run:

```
$ rake figaro:heroku
```

Alternatively, you can set Heroku environment variables directly.

Here's how to set environment variables directly on Heroku with `heroku config:add`.

```
$ heroku config:add GMAIL_USERNAME='myname@gmail.com' GMAIL_PASSWORD='secret'
```

You can check that the environment variables are set with:

```
$ heroku config
```

See the Heroku documentation on Configuration and Config Vars and the article Rails Environment Variables for more information.

Push to Heroku

After all this preparation, you can finally push your application to Heroku.

Be sure you've run `RAILS_ENV=production rake assets:precompile`. Run it each time you change your CSS or JavaScript files.

Be sure to commit your code to the Git local repository before you push to Heroku.

You commit your code to Heroku just like you push your code to GitHub.

Here's how to push to Heroku:

```
$ git push heroku master
```

The push to Heroku takes several minutes. You'll see a sequence of diagnostic messages in the console, beginning with:

```
-----> Ruby/Rails app detected
```

and finishing with:

```
-----> Launching... done
```

Visit Your Site

Open your Heroku site in your default web browser:

```
$ heroku open
```

Your application will be running at http://my-app-name.herokuapp.com/.

If you've configured everything correctly, you should be able to sign up for the newsletter and send a contact request.

Customizing

For a real application, you'll likely want to use your own domain name for your app.

See Heroku's article about custom domains for instructions.

You may also want to improve website responsiveness by adding apge caching with a content delivery network such as CloudFlare. CloudFlare can also provide an SSL connection for secure connections between the browser and server.

Heroku offers many add-on services. These are particularly noteworthy:

- Adept Scale – automated scaling of Heroku dynos
- New Relic – performance monitoring

For an in-depth look at your options, see the Rails Heroku Tutorial.

Troubleshooting

When you get errors, troubleshoot by reviewing the log files:

```
$ heroku logs
```

If necessary, use the Unix `tail` flag to monitor your log files. Open a new terminal window and enter:

```
$ heroku logs -t
```

to watch the server logs in real time.

Where to Get Help

Your best source for help with Heroku is Stack Overflow. Your issue may have been encountered and addressed by others.

You can also check the Heroku Dev Center or the Heroku Google Group.

Chapter 20

Comments

Credits

Daniel Kehoe implemented the application and wrote the tutorial.

Photos

Images provided by the lorempixel.com service are used under the Creative Commons license (CC BY-SA). Visit the Flickr accounts of the photographers to learn more about their work:

- photo of San Francisco by Eneas De Troya
- photo of Sydney by Jimmy Harris
- photo of Paris by Archie Ballantine

The photo of San Francisco by Eneas De Troya appears in the screenshot in the Introduction chapter and on the tutorial cover page.

Did You Like the Tutorial?

Was this useful to you? Follow rails_apps on Twitter and tweet some praise. I'd love to know you were helped out by the tutorial.

Any issues? Please create an issue on GitHub. Reporting (and patching!) issues helps everyone.