Chapter 3

Mostly static pages

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

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

3.1 Sample app setup

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

Listing 3.1: Generating a new sample app.

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

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

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

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

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

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

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

Listing 3.2: A `Gemfile` for the sample app.

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

gem 'rails',      '6.0.1'
gem 'puma',       '3.12.1'
gem 'sass-rails', '5.1.0'
gem 'webpacker',  '4.0.7'
gem 'turbolinks', '5.2.0'
gem 'jbuilder',   '2.9.1'
gem 'bootsnap',   '1.4.4', require: false

group :development, :test do
  gem 'sqlite3', '1.4.1'
  gem 'byebug',  '11.0.1', platforms: [:mri, :mingw, :x64_mingw]
end

group :development do
  gem 'web-console',           '4.0.1'
  gem 'listen',                '3.1.5'
  gem 'spring',                '2.1.0'
  gem 'spring-watcher-listen', '2.0.1'
end

group :test do
  gem 'capybara',                  '3.28.0'
  gem 'selenium-webdriver',        '3.142.4'
  gem 'webdrivers',                '4.1.2'
  gem 'rails-controller-testing', '1.0.4'
  gem 'minitest',                  '5.11.3'
  gem 'minitest-reporters',        '1.3.8'
  gem 'guard',                     '2.15.0'
  gem 'guard-minitest',            '2.4.6'
end

group :production do
  gem 'pg', '1.1.4'
end

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw, :jruby]
```

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

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

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

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

```
$ bundle update
```

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

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

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

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

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

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

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

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

```
# Ruby on Rails Tutorial sample application

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

## License

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

## Getting started

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

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

Next, migrate the database:

```
$ rails db:migrate
```

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

```
$ rails test
```

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

```
$ rails server
```

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

Then commit the changes as follows:

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

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

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

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

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

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

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

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

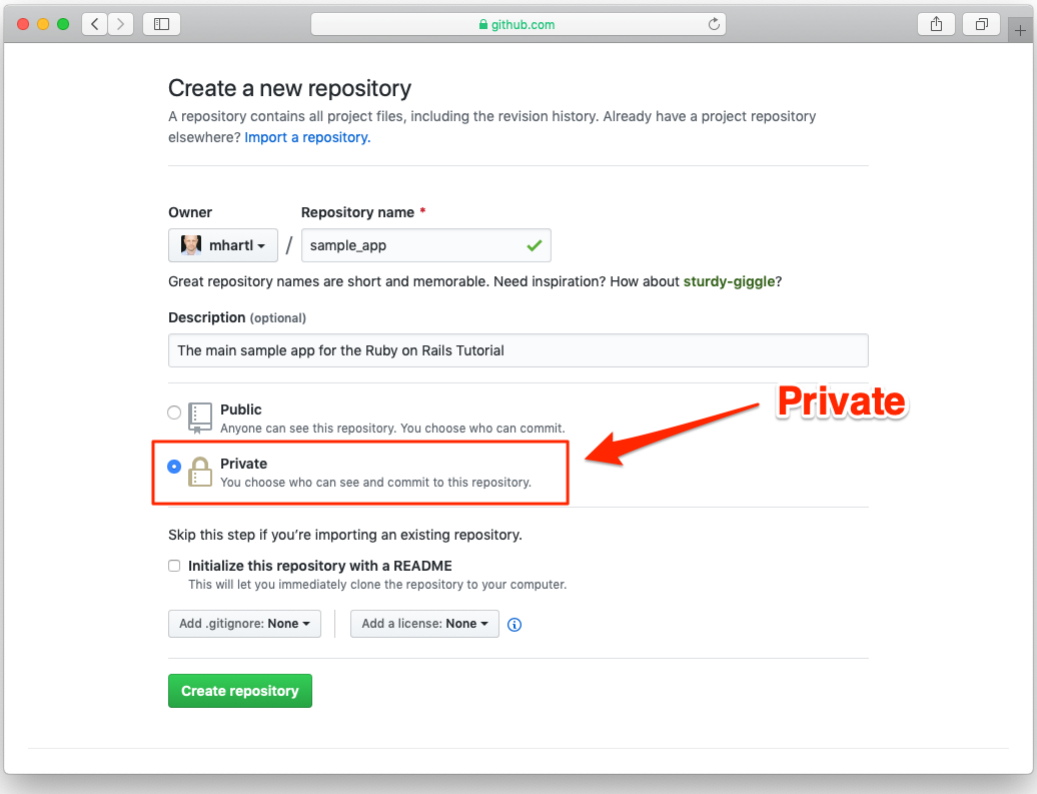

Figure 3.1: Creating the main sample app repository at GitHub.

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

```
class ApplicationController < ActionController::Base

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

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

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

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

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

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

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

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

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

Exercises

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

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

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

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

3.2 Static pages

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

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

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