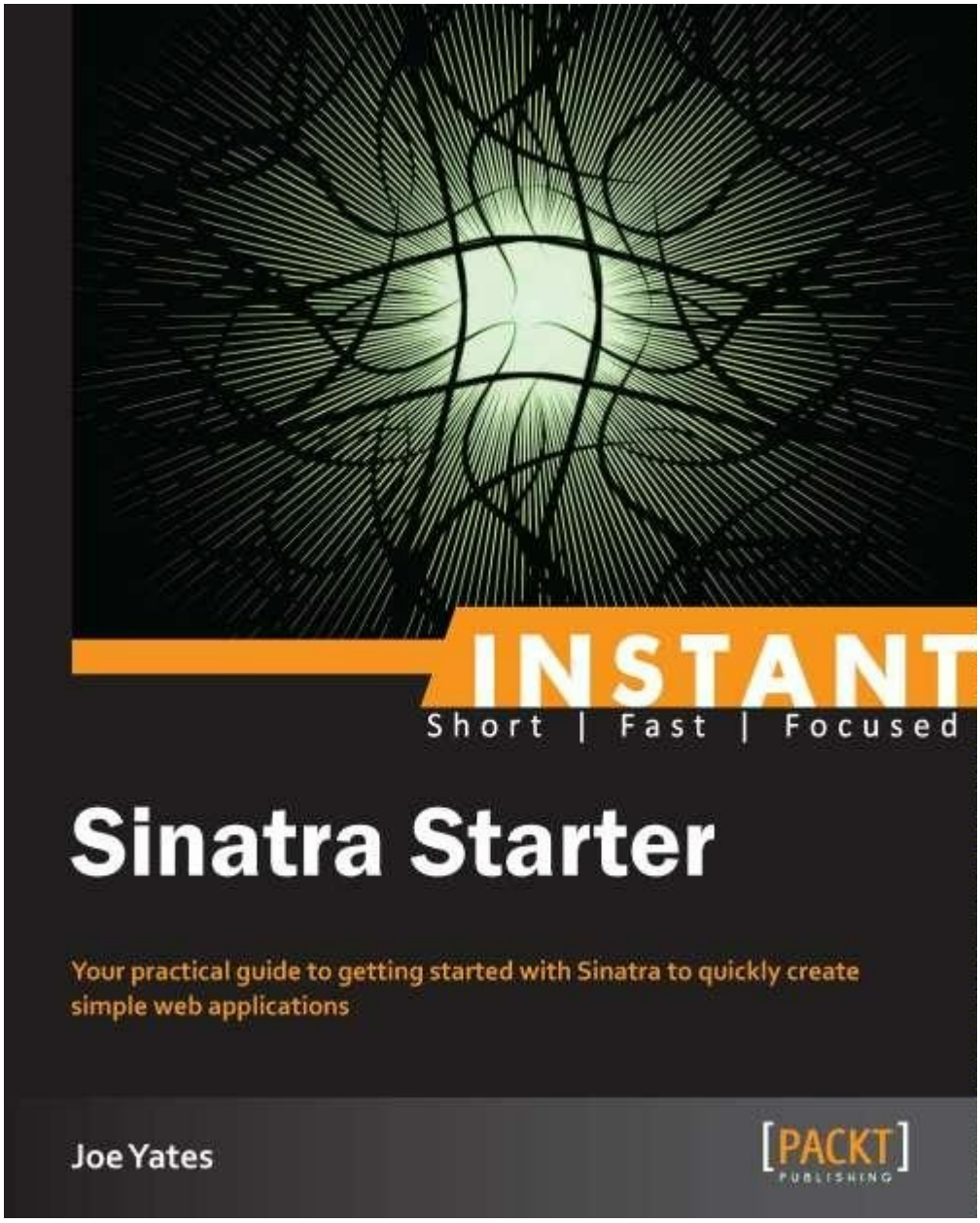

INSTANT
Short | Fast | Focused
Sinatra Starter
Your practical guide to getting started with Sinatra to quickly create simple web applications
Joe Yates
[PACKT]
PUBLISHING

Table of Contents

Instant Sinatra Starter
Credits
About the Author
About the Reviewers
www.packtpub.com
    Support files, eBooks, discount offers and more
packtlib.packtpub.com
    Why Subscribe?
    Free Access for Packt account holders
1. Sinatra Starter
    So, what is Sinatra?
        What is a web framework?
        Sinatra or Ruby on Rails?
        Who's using it?
        Minimalistic
        Education
        Performance
        Prototyping
        Source code
    Installation
        Step 1 – what do I need?
        Step 2 – installing library dependencies
            Installing a C compiler
            Installing external packages
        Step 3 – installing Ruby
            Installing a Ruby Version Manager
            Installing Ruby
        Step 4 – installing Bundler
        Step 5 – installing Git
        Step 6 – registering for online accounts
            Code
            Application deployment
        And that's it
    Quick start – your first Sinatra application
        Step 1 – creating the application
            Logging
        Step 2 – putting the application under version control with Git
        Step 3 – deploying the application
        Step 4 – page layout with Slim
        Step 5 – styling
        Step 6 – development setup
        Step 7 – testing the application
    Top 18 features you need to know about
        Sinatra application types

- Classic applications
- Modular style applications
- Runtime environments
- Project layout
  - Static files
  - Templates
- Middleware
- Sessions
  - Session security
- GET requests
- Handling forms
  - Uploading files
  - Protecting your application
  - CSRF
    - How a CSRF attack works
    - Implementing CSRF protection for your application
- Handlers, routes, and parameters
  - Handlers
  - Routes
    - First to last
  - Parameters
  - The params variable
  - block parameters
  - Query parameters
  - Wildcard or splat parameters
  - Optional parameters
  - Regular expressions
  - Route conditions
    - The provides condition
- Templating
  - Inline templates
  - HTML templates
  - Slim
- Layouts
- Sending e-mail
- Logging
  - How to write logs to a file
- Request
- Response
  - Status
- Using a database
  - Setup
  - Create a database
  - Insert an address
- The Sinatra DSL
- Helpers
- Settings

People and places you should get to know

Official sites
Articles and tutorials
Community
Blogs
Twitter

Instant Sinatra Starter

Instant Sinatra Starter

First published: June 2013

Production Reference: 1300513

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78216-821-8

www.packtpub.com

Credits

Author

Joe Yates

Reviewers

Michał Kwiatkowski

Tim Millwood

Walt Stoneburner

Acquisition Editor

Erol Staveley

Commissioning Editor

Yogesh Dalvi

Technical Editor

Dennis John

Copy Editor

Insiya Morbiwala

Project Coordinator

Sneha Modi

Proofreaders

Aaron Nash

Lydia Morris

Production Coordinator

Manu Joseph

Cover Work

Manu Joseph

Cover Image

Aditi Gajjar

About the Author

Joe Yates is a programmer with 15 years of experience in software development on the desktop and Web platforms, and develops software for areas ranging right from business to culture to research. While his main focus is on projects based on Ruby (in Sinatra and Ruby on Rails), he mixes in other languages (such as Python, JavaScript/Node.js, and Clojure) when appropriate and is developing an interest in the area of DevOps using Chef.

He is based in Italy and works as a consultant. He also develops projects for various companies across Europe.

About the Reviewers

Michał Kwiatkowski started his programming adventure in the Commodore 64, back when 1 MHz CPU and 64 KB of RAM were more than enough. When he finally got a computer with a hard drive, he went on to teach himself Linux and Perl. His programming education continued with Lisp in its many incarnations: Scheme of Wizard Book and Little Schemer, Common Lisp, and more recently, Clojure. He has been involved with open source since he got a modem and is also an active member of the Python and Ruby communities. Michał has 8 years of experience in web development and has delivered multiple RoR and Sinatra applications.

When not programming, he plays games of all kinds: *Go*, board games, tabletop RPGs, and video games. He is a co-founder of *Shelly Cloud*, a PaaS solution for Ruby applications. His website is trivas.pl and he tweets with the `@michalkw` Twitter handle.

Tim Millwood is a web developer based in the UK who has had an avid interest in computers and programming from a very young age. After completing his university education in 2007 with a BSc (Hons) in Computing, he started working with Drupal, building community sites for an educational organization. Currently Tim works at Acquia, helping enterprise-level organizations successfully launch and scale their Drupal sites.

With a keen interest in computing, as a hobby as much as a profession, Tim uses his spare time to work on freelance projects. Over the past few years, this has got him acquainted with the world of Ruby, where he mainly works with Sinatra and Ruby on Rails frameworks.

Tim has had several articles published in both the *.net* magazine and on their website. Recently this included a tutorial introducing Sinatra and focusing on building a simple to-do list application with Sinatra. Previously he has also covered many Drupal-related topics. Interested in sharing what he has learned, Tim blogs on his site (http://www.millwoodonline.co.uk) about a wide range of web development topics. Lately this has covered Sinatra, Ruby on Rails, and Drupal.

I would like to thank my wife Kelly and son Joshua for putting up with slightly less attention from me over evenings and weekends while I worked on this book.

Walt Stoneburner is a software architect with over 25 years of commercial application development and consulting experience. Fringe passions include quality assurance, configuration

management, and security. If cornered, he may actually admit to liking statistics and authoring documentation as well.

He's easily amused by programming language design, collaborative applications, big data, knowledge management, and ASCII art. Self-described as a closet geek, Walt also evaluates software products and consumer electronics, draws cartoons, runs a freelance photography studio, writes humor pieces, performs sleight of hand, enjoys game design, and can occasionally be found on ham radio.

He may be reached directly via e-mail at <wls@wwco.com>. He publishes a tech and humor blog called *Walt-O-Matic* at http://www.wwco.com/~wls/blog/.

His other book reviews and contributions include the following:

- ***AntiPatterns and Patterns in Software Configuration Management*, *Wiley***
- ***Exploring Software: How to Break Code*, *Addison-Wesley Professional***
- ***Ruby on Rails: Web Mashup Projects*, *Packt Publishing***
- ***Building Dynamic Web 2.0 Websites with Ruby on Rails*, *Packt Publishing***
- ***South Mouth: Hillbilly Wisdom, Redneck Observations & Good Ol' Boy Logic*, *CreateSpace Independent Publishing Platform*.**

www.packtpub.com

Support files, eBooks, discount offers and more

You might want to visit www.packtpub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packtpub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

Free Access for Packt account holders

If you have an account with Packt at www.packtpub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Chapter 1. Sinatra Starter

Welcome to the *Instant Sinatra Starter*. This book has been especially created to provide you with all the information that you need to get set up with Sinatra. You will learn the basics of Sinatra and get started with building, testing, and deploying your first Sinatra application.

What this book covers:

So, what is Sinatra? explains what Sinatra actually is, what you can do with it, and why it's so great.

Installation explains how to install Sinatra with minimum fuss and then set it up so that you can use it as soon as possible.

Quick start – your first Sinatra application shows you how to perform the core tasks of Sinatra: handling a GET request and rendering a web page.

Top 18 features you need to know about teaches you how to perform 18 tasks with the most important features of Sinatra. By the end of this section, you will be able to create a simple web application that will include forms, uploads, data storage, testing, and deployment.

People and places you should get to know provides you with many useful links to the project page and forums, as well as a number of helpful articles, tutorials, blogs, and the Twitter feeds of Sinatra's super contributors.

So, what is Sinatra?

Sinatra is a minimalistic web framework written in Ruby. The project was started in 2007 by Blake Mizerany. It is currently being maintained by Konstantin Haase. You can download Sinatra from GitHub at https://github.com/sinatra/sinatra.

What is a web framework?

Web frameworks handle all the mundane tasks involved in the functioning of a website.

When a user's browser sends a request for a web page, that request eventually gets to the web server that handles the site. Here, the web framework decides which part of your code needs to run and passes the request to the code. Next, it collects the response produced by your code and sends it back to the user's browser.

Sinatra or Ruby on Rails?

Sinatra is much less well known than other web frameworks written in Ruby, such as Ruby on Rails.

Rails offers you a lot of features: built-in e-mailing, a testing framework, web templating, and much more. The price you pay for this is that Rails is very complex—you need to know a lot to use it well; and if you want to solve a smaller problem, you need to work to strip away the bloat.

On the other hand, Sinatra starts simple, but you can add on what you need as and when it becomes necessary.

Who's using it?

Sinatra's simplicity doesn't mean that it can't be used for serious applications. In fact, many big names such as the BBC, LinkedIn, and GitHub use it to power parts of their sites (see the full list at http://www.sinatrarb.com/wild.html). These companies have chosen Sinatra because it is lean and gets the job done. Alongside the big names, many smaller projects are using Sinatra. As of today, the Sinatra library has been downloaded more than 7 million times from www.RubyGems.org (https://rubygems.org/gems/sinatra).

Minimalistic

Sinatra is said to be minimalistic because you can do real work with a few lines of code. This minimalistic nature can be an advantage in many situations.

Education

For a beginner, understanding how a request is processed is not a complex task. A complete mini-application can be created in a single file, so all the code is visible at the same time. It is clear how the request is received, what processing happens, and how the response is created. All of this means that Sinatra is a great tool for teaching and learning programming both in structured courses and in self-directed study.

Performance

Sinatra also lends itself to highly performant, lightweight applications. If you want to process a lot of simple requests in parallel, you can run a number of instances of a Sinatra application without using a lot of system resources.

The high-profile sites that use Sinatra do so for this reason—for implementing a simple API to allow other systems fast access to data.

Prototyping

Another area where Sinatra shines is in prototyping; you can have a bare-bones application up and running, responding to two or three simple requests, in just a few minutes. Quickly testing out your ideas in a concrete way helps clarify problem areas and create new approaches.

Source code

The source code for Sinatra is about 1,500 lines long (https://github.com/sinatra/sinatra). It is well-structured and well-documented. Once you've gotten started with Sinatra, thanks to this book, try reading or skimming through the source code; it will give you a lot of insight into what Sinatra can do and how it does it.

Installation

In six easy steps, you can install Sinatra and get it set up on your system.

Step 1 – what do I need?

Before you install Sinatra, you will need to check that you have all of the following required elements:

- Disk space: 100 MB free (min). You'll need space for Ruby, Sinatra, and other associated libraries, and for your project itself. A Sinatra project can easily fit in 30 MB of hard disk space, but more space may be needed for images and other media files.
- Memory: 256 MB (min), 1 GB (recommended).
- Operating system: Preferably Ubuntu Linux or Mac OS X. Sinatra development on Windows is possible, but presents numerous extra problems and so is not covered in this book.

Step 2 – installing library dependencies

Your Sinatra projects will often use various software libraries for common tasks, such as HTML templating, sending e-mail, and communicating with databases. These libraries are called **gems**. If a gem is written purely in Ruby, installation is easy—you just have to download it.

Things are a little more complicated if the gem is written partly in Ruby and partly in C. These gems are said to have "native extensions"—you'll see these words in warning messages when you get around to installing your gems.

Installing a C compiler

For native gems, you'll need to have a C compiler installed, preferably **gcc**.

If you're using Ubuntu, use the following command to install the compiler:

```
sudo apt-get install build-essential
```

On Mac OS X, things are a little more complicated. Apple's Xcode software development system does provide a C compiler called **llvm**, but it's not suitable for compiling a lot of current gems.

I recommend using **Homebrew** (http://mxcl.github.com/homebrew/) as a package manager; it will install the system libraries that are not part of the

basic OS X install. Once you've got Homebrew working, install gcc using the following commands:

```
brew tap homebrew/versions
brew install gcc48
```

Installing external packages

Depending on the specific gem, you may need to install extra packages for your operating system.

A well-written gem will have a README file that will indicate the external dependencies it has.

In the application that we will be developing during the course of this book, we'll create SQLite databases. To do so, we will be using the `sqlite3` gem, which depends on SQLite's binary libraries. This means that we don't just need standard SQLite but also its development libraries and headers.

On Ubuntu, run the following command:

```
sudo apt-get install libsqlite3-dev
```

On Mac OS X, use Homebrew with the following command:

```
brew install sqlite
```

Step 3 – installing Ruby

Ruby is available in many different implementations and versions. Ruby 1.8.7 is still present on a lot of systems, but it's best avoided as it's incompatible with a lot of gems that you'll want to use.

Mac OS X comes with a preinstalled version of Ruby, but it is too old to be useful.

In such cases, you need to install a more recent version of Ruby.

At the time of writing, Ruby 1.9 is the recommended version. While Ruby 2.0 has just come out, it may cause some compatibility problems; so to start with, you should skip it.

Installing a Ruby Version Manager

As there are many different versions of Ruby available, Ruby developers find it useful to be able to switch between them to check if a project that works with one version also works with another. To aid this switch, a number of Ruby version managers have been created.

The best known Ruby version managers are **RVM** (https://rvm.io/) and **rbenv** (https://github.com/sstephenson/rbenv/).

I recommend using `rbenv` (but if you prefer RVM, that's fine; just install Ruby 1.9.3 with it and skip the next section).

On Linux, follow the instructions on the `rbenv` home page at https://github.com/sstephenson/rbenv#installation.

You'll need to install `rbenv` itself as well as the `ruby-build` plugin.

On Mac OS X, use the following command:

```
brew install rbenv
brew install ruby-build
```

Installing Ruby

Once you've got `rbenv` working, install Ruby 1.9.3 using the following command:

```
rbenv install 1.9.3-p392
rbenv global 1.9.3-p392
```

Ruby 1.9.3 will now be used by default instead of any version that might have been installed with your operating system.

Step 4 – installing Bundler

The `bundler` gem handles the collections of gems that are required by an application to make it work.

Install `bundler` using the following command:

```
gem install bundler
```

Ensure that the version of `bundler` is recent enough using the following command:

```
bundle --version
```

You will need at least version 1.2.0.

Ruby projects normally have a file called **Gemfile** that lists the gems that the project depends on. `bundler` works by reading your project's Gemfile, checking versions and dependencies, and installing what is needed.

Once you have a Gemfile set up, the basic command is as follows:

```
bundle install
```

> **Tip**
>
> Don't try this until you have a Gemfile, or you'll just get an error message.

When you start to have a number of different Ruby projects, you will have different sets of gem dependencies. With `bundler`, you can keep each project's gems in a directory within the project directory itself. You can do this explicitly, using the following command:

```
bundle install –path=./vendor/bundle
```

As this is really the behavior you will want for any project, make this the default. Do so by creating a `~/.bundle/config` file using the following content:

```
---
BUNDLE_PATH: vendor/bundle
```

Step 5 – installing Git

You'll need to keep your applications under source control. This book uses Git (http://git-scm.com/) for this purpose.

Install Git on Ubuntu using the following command:

```
sudo apt-get install git-core
```

On Mac OS X, use the following command:

```
brew install git
```

If you like, there are also applications available that provide a graphical user interface to manage Git repositories. For example, GitHub provides "GitHub for Mac."

If you are unfamiliar with Git, there are some very good online books and tutorials available, such as **Pro Git** (http://git-scm.com/book).

Step 6 – registering for online accounts

There are sites now that will host your source code and (small) applications for free. In the next section, we will be using **GitHub** to handle source code and we will use **Heroku** for deployment. There are other alternatives such as **Bitbucket** for source code, but if you want to follow the examples in this book, you'll need accounts on GitHub and Heroku.

Code

Create an account for yourself on GitHub (https://github.com/) so that you can store your applications' source code.

> **Note**
>
> Note that free GitHub accounts only provide public repositories. If you want to keep your code private without paying anything, you can create an account on Bitbucket (https://bitbucket.org/).

Application deployment

Create an account on Heroku (http://www.heroku.com/) so you can publish your first application online easily. You can run small applications on Heroku at no cost, so it's a good way to get started without having to think about hosting.

You don't necessarily have to set up a Heroku account, but it's a good way to get your first application online.

First, we need to set up communication with Heroku using the following steps:

1. Install the Heroku Toolbelt:

- On Linux, run the following command:

```
wget -qO-
https://toolbelt.heroku.com/install-
ubuntu.sh | sh
```

  This command downloads and runs a script that does all the necessary setup.

- On Mac OS X, download the Toolbelt app from https://toolbelt.heroku.com/osx.

2. Log in to Heroku using the following command:

```
heroku login
```

   You'll need to create an SSH cryptographic key pair to communicate with Heroku. If you've already got an SSH key pair, just go to https://dashboard.heroku.com/account and add the public key.

   If not, perform the following steps:

   1. Create an SSH key pair:

```
ssh-keygen
```

   2. You will get a pair of files in your `~/.ssh` directory: `id_rsa` and `id_rsa.pub`. Copy the content of `id_rsa.pub` to https://dashboard.heroku.com/account.

Complete instructions for using Heroku Toolbelt are available at https://toolbelt.heroku.com/.

And that's it

By this point, you should have everything you need to create Sinatra applications. In the next section, you'll learn how to create a simple application and deploy it to Heroku.

Quick start – your first Sinatra application

In this section, we're going to put together the basics of a simple address book application and deploy it. We'll be handling requests and returning a dynamically created page.

Step 1 – creating the application

The first thing to do is set up Sinatra itself, which means creating a Gemfile.

1. Open up a Terminal window and navigate to the directory where you're going to keep your Sinatra applications.
2. Create a directory called `address-book` using the following command:

```
mkdir address-book
```

3. Move into the new directory:

```
cd address-book
```

4. Create a file called `Gemfile`:

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

gem 'sinatra'
```

5. Install the gems via `bundler`:

```
bundle install
```

   You will notice that Bundler will not just install the `sinatra` gem but also its dependencies. The most important dependency is **Rack** (http://rack.github.com/), which is a common handler layer for web servers. Rack will be receiving requests for web pages, digesting them, and then handing them off to your Sinatra application.

   If you set up your Bundler configuration as indicated in the previous section, you will now have the following files:

   - `.bundle`: This is a directory containing the local configuration for Bundler

- `Gemfile`: As created previously
- `Gemfile.lock`: This is a list of the actual versions of gems that are installed
- `vendor/bundle`: This directory contains the gems

> **Note**
>
> You'll need to understand the `Gemfile.lock` file. It helps you know exactly which versions of your application's dependencies (gems) will get installed. When you run `bundle install`, if Bundler finds a file called `Gemfile.lock`, it will install exactly those gems and versions that are listed there. This means that when you deploy your application on the Internet, you can be sure of which versions are being used and that they are the same as the ones on your development machine. This fact makes debugging a lot more reliable. Without `Gemfile.lock`, you might spend hours trying to reproduce behavior that you're seeing on your deployed app, only to discover that it was caused by a glitch in a gem version that you haven't got on your machine.

So now we can actually create the files that make up the first version of our application.

6. Create `address-book.rb`:

```
require 'sinatra/base'

class AddressBook < Sinatra::Base
  get '/' do
    'Hello World!'
  end
end
```

This is the skeleton of the first part of our application.

Line 1 loads Sinatra, line 3 creates our application, and line 4 says we handle requests to `'/'`—the root path. So if our application is running on `myapp.example.com`, this means that this method will handle requests to `http://myapp.example.com/`.

Line 5 returns the string Hello World!. Remember that a Ruby block or a method without explicit use of the return keyword will return the result of its last line of code.

7. Create `config.ru`:

```
$: << File.dirname(__FILE__)
require 'address-book'

run AddressBook.new
```

This file gets loaded by `rackup`, which is part of the Rack gem. **Rackup** is a tool that runs rack-based applications. It reads the configuration from `config.ru` and runs our application.

Line 1 adds the current directory to the list of paths where Ruby looks for files to load, line 2 loads the file we just created previously, and line 4 runs the application.

Let's see if it works.

8. In a Terminal, run the following command:

```
bundle exec rackup -p 3000
```

Here `rackup` reads `config.ru`, loads our application, and runs it.

We use the `bundle exec` command to ensure that only our application's gems (the ones in `vendor/bundle`) get used. Bundler prepares the environment so that the application only loads the gems that were installed via our Gemfile.

The -p 3000 command means we want to run a web server on port 3000 while we're developing.

9. Open up a browser and go to `http://0.0.0.0:3000`; you should see something that looks like the following screenshot:

Illustration 1: The Hello World! output from the application

Logging

Have a look at the output in the Terminal window where you started the application.

I got the following (line numbers are added for reference):

```
1  [2013-03-03 12:30:02] INFO  WEBrick 1.3.1
2  [2013-03-03 12:30:02] INFO  ruby 1.9.3
(2013-01-
15) [x86_64-linux]
3  [2013-03-03 12:30:02] INFO
WEBrick::HTTPServer#start: pid=28551 port=3000
4  127.0.0.1 - - [03/Mar/2013 12:30:06] "GET /
HTTP/1.1" 200 12 0.0142
5  127.0.0.1 - - [03/Mar/2013 12:30:06]
"GET /favicon.ico HTTP/1.1" 404 445 0.0018
```

Like it or not, you'll be seeing a lot of logs such as this while doing web development, so it's a good idea to get used to noticing the information they contain.

Line 1 says that we are running the `WEBrick` web server. This is a minimal server included with Ruby—it's slow and not very powerful so it shouldn't be used for production applications, but it will do for now for application development.

Line 2 indicates that we are running the application on Version 1.9.3 of Ruby. Make sure you don't develop with older versions, especially the 1.8 series, as they're being phased out and are missing features that we will be using in this book.

Line 3 tells us that the server started and that it is awaiting requests on port `3000`, as we instructed.

Line 4 is the request itself: `GET /`. The number `200` means the request succeeded—it is an HTTP status code that means *Success*.

Line 5 is a second request created by our web browser. It's asking if the site has a `favicon`, an icon representing the site. We don't have one, so Sinatra responded with `404` (not found).

When you want to stop the web server, hit *Ctrl* + *C* in the Terminal window where you launched it.

Step 2 – putting the application under version control with Git

When developing software, it is very important to manage the source code with a version control system such as Git or Mercurial. **Version control** systems allow you to look at the development of your project; they allow you to work on the project in parallel with others and also to try out code development ideas (branches) without messing up the stable application.

1. Create a Git repository in this directory:

```
git init
```

2. Now add the files to the repository:

```
git add Gemfile Gemfile.lock
address-book.rb config.ru
```

3. Then commit them:

```
git commit -m "Hello World"
```

4. I assume you created a GitHub account earlier. Let's push the code up to www.github.com for safe keeping. Go to https://github.com/new.
5. Create a repo called `sinatra-address-book`.
6. Set up your local repo to send code to your GitHub account:

```
git remote add origin
git@github.com:YOUR_ACCOUNT/sinatra-
address-book.git
```

7. Push the code:

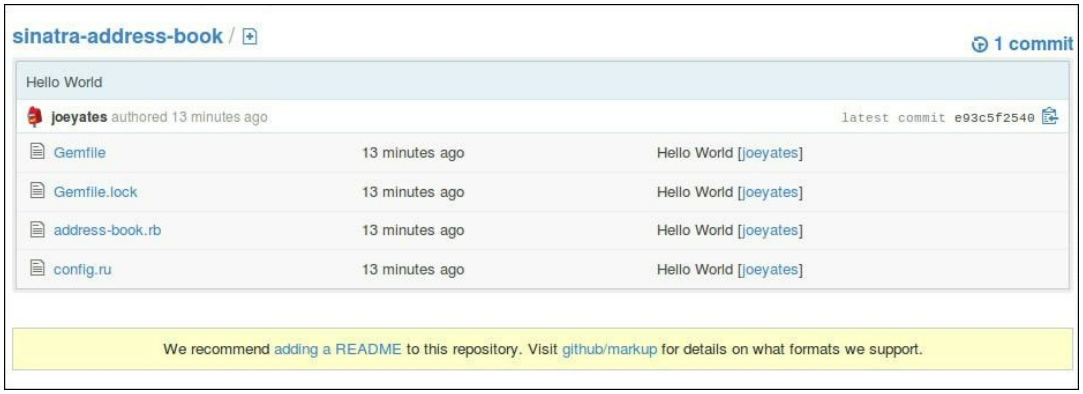
```
git push
```

You may need to sort out authentication if this is your first time pushing code. So if you get an error such as the following, you'll need to set up authentication on GitHub:

```
Permission denied (publickey)
```

Go to https://github.com/settings/ssh and add the public key that you generated in the previous section.

Now you can refresh your browser, and GitHub will show you your code as follows:



Note that the code in my GitHub repository is marked with tags. If you want to follow the changes by looking at the repository, clone my repo from //github.com/joeyates/sinatra-address-book.git into a different directory and then "check out" the correct tag (indicated by a footnote) at each stage.

To see the code at this stage, type in the following command:

```
git checkout 01_hello_world
```

If you type in the following command, Git will tell you that you have "untracked files", for example, `.bundle`:

```
git status
```

To get rid of the warning, create a file called `.gitignore` inside the project and add the following content:

```
/.bundle/
/vendor/bundle/
```

Git will no longer complain about those directories. Remember to add `.gitignore` to the Git repository and commit it.

Let's add a README file as the page is requesting, using the following steps:

1. Create the `README.md` file and insert the following text:

```
sinatra-address-book
====================
An example program of various Sinatra
functionality.
```

2. Add the new file to the repo:

```
git add README.md
```

3. Commit the changes:

```
git commit -m "Add a README explaining the
application"
```

4. Send the update to GitHub:

```
git push
```

Now that we have a README file, GitHub will stop complaining.

What's more is other people may see our application and decide to build on it. The README file will give them some information about what the application does.

Step 3 – deploying the application

We've used GitHub to host our project, but now we're going to publish it online as a working site. In the introduction, I asked you to create a Heroku account. We're now going to use that to deploy our code.

Heroku uses Git to receive code, so we'll be setting up our repository to push code to Heroku as well.

Now let's create a Heroku app:

```
heroku create
Creating limitless-basin-9090... done, stack is
cedar
http://limitless-basin-
9090.herokuapp.com/ |
git@heroku.com:limitless-basin-
9090.git
Git remote heroku added
```

My Heroku app is called `limitless-basin-9090`. This name was randomly generated by Heroku when I created the app. When you generate an app, you will get a different, randomly generated name.

My app will be available on the Web at the `http://limitless-basin-9090.herokuapp.com/` address. If you deploy your app, it will be available on an address based on the name that Heroku has generated for it.

Note that, on the last line, Git has been configured too.

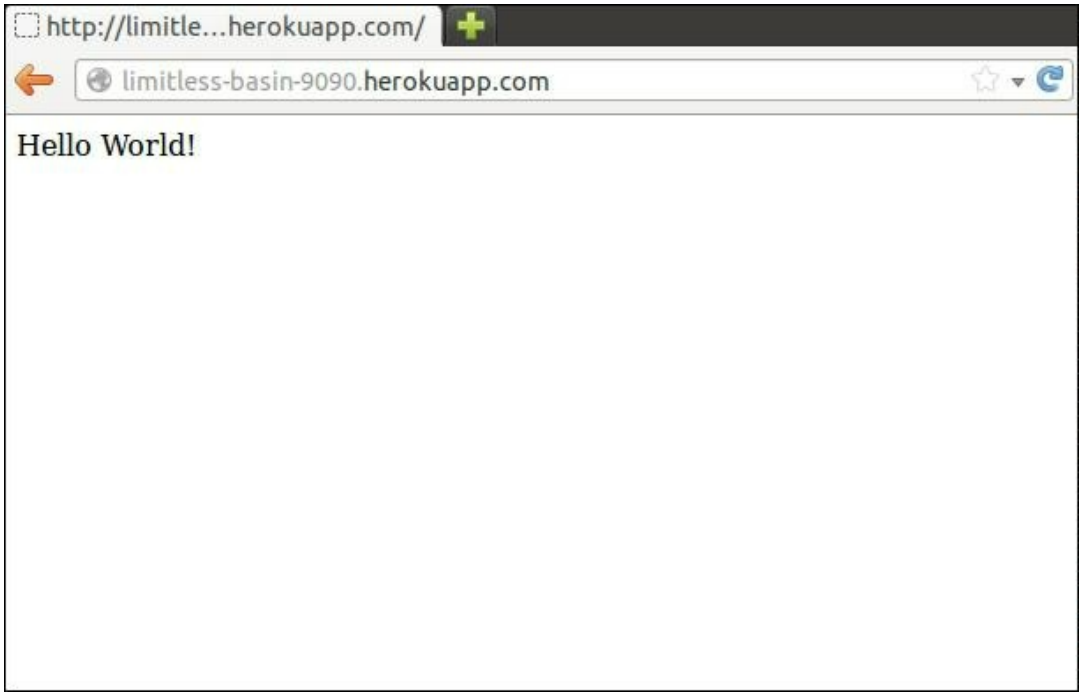
To see what has happened, use the following command:

```
git remote show heroku
* remote heroku
  Fetch URL: git@heroku.com:limitless-basin-9090.git
  Push  URL: git@heroku.com:limitless-basin-9090.git
  HEAD branch: (unknown)
```

Now let's deploy the application to the Internet:

```
git push heroku master
```

Now the application is online for all to see:



The initial version of the application, running on Heroku

Step 4 – page layout with Slim

The page looks a bit sad. Let's set up a standard page structure and use a templating language to lay out our pages.

A **templating language** allows us to create the HTML for our web pages in a clearer and more concise way.

There are many HTML templating systems available to the Sinatra developer: **erb**, **haml**, and **slim** are three popular choices. We'll be using Slim (http://slim-lang.com/).

Let's add the gem:

1. Update our Gemfile:

```
gem 'slim'
```

2. Install the gem:

```
bundle
```

We will be keeping our page templates as .slim files. Sinatra looks for these in the `views` directory.

Let's create the directory, our new home page, and the standard layout for all the pages in the application.

1. Create the `views` directory:

```
mkdir views
```

2. Create `views/home.slim`:

```
p address book - a Sinatra application
```

   When run via Sinatra, this will create the following HTML markup:

```
<p>address book - a Sinatra application</p>
```

3. Create `views/layout.slim`:

```
doctype html
html
  head
    title Sinatra Address Book
  body
    == yield
```

   Note how Slim uses indenting to indicate the structure of the web page.

   The most important line here is as follows:

```
== yield
```

This is the point in the layout where our home page's HTML markup will get inserted. The yield instruction is where our Sinatra handler gets called. The result it returns (that is, the web page) is inserted here by Slim.

Finally, we need to alter `address-book.rb`.

Add the following line at the top of the file:

```
require 'slim'
```

Replace the get '/' handler with the following:

```
get '/' do
  slim :home
end
```

4. Start the local web server as we did before:

```
bundle exec rackup -p 3000
```

The following is the new home page:

Using the Slim Templating Engine

Have a look at the source for the page. Note how the results of `home.slim` are inserted into `layout.slim`.

Let's get that deployed.

5. Add the new code to Git and then add the two new files:

```
git add views/*.slim
```

   Also add the changes made to the other files:

```
git add address-book.rb Gemfile
Gemfile.lock
```

   Commit the changes with a comment:

```
git commit -m "Generate HTML using Slim"
```

6. Deploy to Heroku:

```
git push heroku master
```

7. Check online that everything's as expected.

Step 5 – styling

To give a slightly nicer look to our pages, we can use **Bootstrap** (http://twitter.github.io/bootstrap/); it's a CSS framework made by Twitter.

Let's modify `views/layout.slim`. After the line that says `title Sinatra Address Book`, add the following code:

```
    link
href="//netdna.bootstrapcdn.com/twitter-
bootstrap/2.3.1/css/bootstrap-
combined.min.css" rel="stylesheet"
```

There are a few things to note about this line.

Firstly, we will be using a file hosted on a **Content Distribution Network** (**CDN**). Clearly, we need to check that the file we're including is actually what we think it is.

The advantage of a CDN is that we don't need to keep a copy of the file ourselves, but if our users visit other sites using the same CDN, they'll only need to download the file once.

Note also the use of `//` at the beginning of the link address; this is called a "protocol agnostic URL". This way of referencing the document will allow us later on to switch our application to run securely under HTTPS, without having to readjust all our links to the content.

Now let's change `views/home.slim` to the following:

```
div class="container"
  h1 address book
  h2 a Sinatra application
```

We're not using Bootstrap to anywhere near its full potential here. Later on we can improve the look of the app using Bootstrap as a starting point.

Remember to commit your changes and to deploy to Heroku.

Step 6 – development setup

As things stand, during local development we have to manually restart our local web server every time we want to see a change. Now we are going to set things up with the following steps so the application reloads after each change:

1. Add the following block to the Gemfile:

```
group :development do
  gem 'unicorn'
  gem 'guard'
  gem 'listen'
  gem 'rb-inotify', :require => false
  gem 'rb-fsevent', :require => false
  gem 'guard-unicorn'
end
```

   The `group` around these gems means they will only be installed and used in development mode and not when we deploy our application to the Web.

   Unicorn is a web server—it's better than **WEBrick**—that is used in real production environments. WEBrick's slowness can even become noticeable during development, while Unicorn is very fast. `rb-inotify` and `rb-fsevent` are the Linux and Mac OS X components that keep a check on your hard disk. If any of your application's files change, `guard` restarts the whole application, updating the changes.

   Finally, update your gems:

```
bundle
```

2. Now add `Guardfile`:

```
guard :unicorn, :daemonize => true do
  `git ls-files`.each_line { |s| s.chomp!;
watch s }
end
```

3. Add a configuration file for `unicorn`:

```
mkdir config
```

4. In `config/unicorn.rb`, add the following:

```
listen 3000
```

5. Run the web server:

```
guard
```

Now if you make any changes, the web server will restart and you will get a notification via a desktop message. To see this, type in the following command:

```
touch address-book.rb
```

You should get a desktop notification saying that `guard` has restarted the application.

Note that to shut `guard` down, you need to press *Ctrl* + *D*.

Also, remember to add the new files to Git.

Step 7 – testing the application

We want our application to be robust. Whenever we make changes and deploy, we want to be sure that it's going to keep working. What's more, if something does not work properly, we want to be able to fix bugs so we know that they won't come back.

This is where testing comes in. Tests check that our application works properly and also act as detailed documentation for it; they tell us what the application is intended for. Our tests will actually be called "specs", a term

that is supposed to indicate that you write tests as specifications for what your code should do.

We will be using a library called **RSpec**. Let's get it installed.

1. Add the gem to the Gemfile:

```
group :test do
  gem 'rack-test'
  gem 'rspec'
end
```

2. Update the gems so RSpec gets installed:

```
bundle
```

3. Create a directory for our specs:

```
mkdir spec
```

4. Create the `spec/spec_helper.rb` file:

```
$: << File.expand_path('../..', __FILE__)

require 'address-book'
require 'rack/test'

def app
  AddressBook.new
end

RSpec.configure do |config|
  config.include Rack::Test::Methods
end
```

5. Create a directory for the `integration` specs:

```
mkdir spec/integration
```

6. Create a `spec/integration/home_spec.rb` file for testing the home page:

```
require 'spec_helper'

describe "Sinatra App" do
  it "should respond to GET" do
    get '/'

    expect(last_response).to be_ok
```

```
    expect(last_response.body).to
match(/address book/)
  end
end
```

What we do here is call the application, asking for its home page.

We check that the application answers with an HTTP status code of `200` (be_ok).

Then we check for some expected content in the resulting page, that is, the address book page.

7. Run the spec:

```
bundle exec rspec
.

Finished in 0.0295 seconds
1 example, 0 failures
```

Ok, so our spec is executed without any errors.

There you have it. We've created a micro application, written tests for it, and deployed it to the Internet.

Top 18 features you need to know about

As you start to use Sinatra, you will realize that there are a wide variety of things that you can do with it. This section will teach you all about the most commonly performed tasks and most commonly used features in Sinatra.

Sinatra application types

They are of the following two types:

Classic applications

Sinatra applications can be created in a single file with very little "boilerplate"—just call `require 'sinatra'` and start writing code. The advantage of these applications is that they are easy to create and comprehend, so they reduce the difficulty for beginners.

The following is a small example. We're using the `classic_app.rb` file here.

```
require 'sinatra'

get '/' do
  'A Sinatra classic application'
end
```

Start the application with the following command:

```
$ ruby classic_app.rb
```

Now open a browser and go to `http://0.0.0.0:4567`. You will see the text **A Sinatra classic application**, so you've got a working web application in four lines of code!

To stop the application, press *Ctrl* + *C*.

Note that `get` has become a globally available method, as have `post`, `put`, and the other verbs. These are some of the verbs of Sinatra's domain-specific language. A **domain-specific language** (**DSL**) defines a set of features that help solve problems in a specific problem domain. Sinatra's DSL introduces features that relate to handling web requests.

Classic applications behave slightly differently from Modular applications (see the next section). The two most important differences are as follows:

1. The application is run automatically.
2. In the spirit of trying to keep everything in one file, the application is set up so that you can put HTML templates in the same file as the rest of the code.

The following is a list of the command-line parameters you can pass to a Classic application (the Sinatra setting that the parameter affects is shown in parentheses along with its default value):

- `-p number`: This sets the server port (setting: `port`, default: `4567`)
- `-o hostname_or_ip_address`: This sets the server host (setting: `bind`, default: `0.0.0.0`)
- `-e environment`: This sets the Sinatra environment, either one of `development`, `test`, or `production` (setting: `environment`, default: `:development`; see the *Settings* section for a fuller discussion of the environment setting)
- `-s server_name`: This sets the server to be used (setting: `server`, default: see the *Settings* section)

Modular style applications

Sinatra's Classic applications put emphasis on the minimal aspects of Sinatra by putting everything in one file. With very little extra code, you can set up a Modular application. Modular applications start out almost identical to Classic applications, but as they grow, you can start to keep your code in separate files.

Modular applications (such as the example we put together in the previous section) separate the code that responds to web requests from templates and from the code used to set up and run the application (normally in `config.ru`).

You need to declare a modular application as follows:

```
require 'sinatra/base'

class ChosenNameOfYouApplication < Sinatra::Base
  get '/' do
    …
  end
end
```

Note that you must require `sinatra/base` and not `sinatra`.

Runtime environments

The environment can be set via the `environment` setting inside the application's code. If an environment variable `RACK_ENV` is set, `environment` takes its value. If no value is explicitly set, it defaults to `:development`.

There are three default runtime environments, as follows:

- `:test`: In this mode, logging is turned off. Any errors that come up are returned to the request's `rack.errors` object (via the `dump_errors` setting). This allows your test code to detect the exact reason for any errors that occur.
- `:development`: This is the default environment. In this mode, caching is turned off (by setting the value of `reload_templates` to `true`) and errors in the application cause an error page to be shown (via the `show_exceptions` setting).
- `:production`: This mode reduces the amount of information written to the logs and activates Sinatra's template caching system so your application responds faster.

Project layout

A listing of the files commonly found in a Sinatra application is as follows:

- `sinatra_app.rb`
- `config.ru`
- `Gemfile`
- `Gemfile.lock`
- `public/`
    - `images/image1.png`
    - `javascripts/script1.js`
    - `stylesheets/stylesheet1.css`
- `views/`
    - `template1.xyz`

Static files

With the default settings enabled, Sinatra will check in the `public` directory for files that correspond to web requests before trying to match the request against the routes declared in the application.

So if you've created a file called `public/hello`, your application will respond with the contents of that file if you request `/hello` rather than call the `get '/hello'` handler.

In this way, files that do not need to be generated on the fly (such as images, CSS stylesheets, and JavaScript files) are correctly located and returned.

Templates

The `views` directory is searched for templates (see the *HTML templates* section).

Middleware

Middleware means a small piece of program code that processes your application's requests and/or responses.

Remember that Sinatra applications sit on top of Rack. **Rack** is a library that handles running Ruby web applications in a standard way. It has a standard way of presenting requests to "Rack-compatible" applications and expects these applications to respond in a standard manner. This standardization means that you can use other people's code to handle common tasks related, for example, to security, authentication, or caching.

For a full list of available middleware, see the Rack wiki on GitHub at https://github.com/rack/rack/wiki/List-of-Middleware.

You can tell Sinatra which middleware to run via the `use` command; for example, we can add the following to `config.ru`:

```
use Rack::Session::Cookie, secret:
'my_application_secret'
```

This code activates a piece of middleware that adds HTTP cookies to your responses. In this way, you can easily maintain information about a user (for example, whether they are logged on or which language they want to view the pages in).

In this way, a Sinatra application remembers the order in which each piece of middleware is activated. When a request is received by the application, it is passed to each piece of middleware in order. The middleware can modify the request or even send back its own request without it ever reaching your application's handler.

Sessions

By default, a session simply consists of the `{secret: some secret value}` hash. You can use this hash to store temporary information about your user; for example, like the language they want to view the pages in.

If the `sessions` setting is set to a hash-like value, its value will be merged into the cookie. The values in the setting will be inserted into a user's session as default values when the session is created.

Session security

For sessions to be useful, you'll need to encode them. To do so, you'll need an application-specific secret code:

```
set :session_secret, 'my_secret_code'
```

The following is a way to generate a secret code:

```
openssl rand -hex 32
```

The preceding command will generate a random string similar to the following:

```
3cada4de6819db6d4f1b13aa285771bf30348c95018af65177
85eec5e9b7fcad
```

The secret code should *not* be included in your Git repository. The best thing to do is to set an environment variable such as the following:

```
MY_APP_SECRET=3cada4de6819db6d4f1b13aa285771bf30348c95018af6517785eec5e9b
unicorn -p 8080
```

You can then use the environment variable as follows:

```
use Rack::Session::Cookie, secret:
ENV['MY_APP_SECRET']
```

GET requests

`GET` requests are the simplest type of web request which can be made to your application.

You can tell Sinatra that you want to respond to a `GET` request as follows:

```
get '/' do
  "This is the application's root"
end
```

Handling forms

Most commonly, `POST` requests are made by a web browser when sending data from an HTML form. Here's a snippet showing how to create a form:

```
<form method="post" action="/surnames">
  <input type="text" name="surname">
  <input type="submit" value="Add a surname">
</form>
```

A matching handler in your Sinatra application would be as follows:

```
post '/surnames' do
  # ...add the new name to the database
   "I've added #{params[:surname]} to the list of
surnames"
end
```

Inside the handler, `params[:surname]` holds the value that the user typed into the `name` input on the form. Don't duplicate input names or you'll only get the value from the last one!

Uploading files

If your HTML form allows users to send you files, make sure that you set up the form encoding in the correct way by specifying `enctype` as `multipart/form-data`:

```
<form method="post" action="/photos"
enctype="multipart/form-data">
  <input type="file" name="photo">
  <input type="submit" value="Add the photo">
</form>
```

When your handler receives that `POST` request, `params[:photo]` will be a hash containing information about the file being uploaded:

```
{
  filename: the name of the file,
  type: the type of file, e.g. "image/png",
  name: the name of the form field which sent
this file,
  tempfile: a file handle from which you can read
the file's contents,
  head: a String containing information about the
upload,
}
```

Create a file called `views/photos.slim`:

```
1   form method="post" action="/photos"
enctype="multipart/form-data"
2   input type="file" name="photo"
3   input type="submit" value="Add the photo"
```

Now add these handlers to `address-book.rb`:

```
 1  get '/photos' do

 2     slim :photos
 3   end
 4
 5   post '/photos' do
 6    if params[:photo].nil?
 7       redirect '/photos'
 8       return
 9     end
10     filename = File.join(settings.root,
'files', params[:photo][:filename])
11     File.open(filename, 'w') do |file|
12       file.write params[:photo][:tempfile].read
13     end
14     "OK, photo saved"
15   end
```

The first handler (lines 1-3) simply shows the view.

Lines 5-15 handle the photo upload.

On line 6, we check if the user has supplied a file—simply clicking on **Add the photo** without selecting a file will have this effect. If this happens, we simply re-present the view.

On line 10, we create a name for the file, which will end up in the `files` directory, and will have the original name that it had on the user's computer.

On lines 11-13, we save the file to the disk.

You'll need to create a `files` directory for the uploads:

```
mkdir files
```

Protecting your application

Sinatra has recently introduced the default use of a middleware called **Rack::Protection**. This software takes the necessary steps to protect your application against various malicious users.

CSRF

One common type of attack called **Cross Site Request Forgery** (**CSRF** for short) needs some configuration to be useful.

If your application handles authentication by allowing users to log on and modify data, your application can become vulnerable to CSRF attacks.

How a CSRF attack works

An attacker finds out how certain modifications are made to your site's data, for example, via `HTTP POST` or `PUT` requests. You may have a form where the user can change his or her username.

If the attacker is able to get your user to visit a specially prepared web page on another site, that web page can send a `POST` request to your server as though it was sent from your web page—they have forged a request.

So without even being aware of it, your user's session can be used by another site to make changes to your site.

The way to protect your application from CSRF is to embed some text (called the **CSRF token**) in your web pages. This text is individual to a specific user and can be checked against the user's session. Whenever you receive a request that might be vulnerable to this kind of attack, you should check that the request includes the CSRF token and that it is correct when compared to the user's session.

This is done by taking the user's session ID and combining it via a cryptographic process with the application's `secret`. If the resulting value turns out the same as the CSRF token, all is well.

As an attacker has no way of knowing what a given user's CSRF token might be, they cannot forge requests to be sent to your application.

Implementing CSRF protection for your application

Sinatra provides `Rack::Protection` that transparently protects your application against a whole host of attacks. For CSRF, you'll need to specifically enable the `AuthenticityToken` middleware in your `config.ru` file. Also, as this method requires the use of cookies, you will have to enable those first (see the *Sessions* section for a discussion on how to set `:secret`):

```
use Rack::Session::Cookie, secret:
my_cookie_secret
use Rack::Protection, use: :authenticity_token
```

In your application, you'll need to ensure that every session has its own authenticity token:

```
  before do
    session[:csrf] ||=
Rack::Protection::Base.new(self).random_string
  end
```

In your application, whenever you present a form to the user, add a hidden input containing the token:

```
input name="authenticity_token"
value=session[:csrf] type="hidden"
```

Now the CSRF protection is set up. Every time an `unsafe` method call (`POST`, `PUT`) is made, the data will be checked for the presence of a value of `authenticity_token` matching the user's session.

Let's see what happens without the correct authenticity token. We'll use the `curl` command-line tool that makes web requests.

```
$ curl -i -d 'username=newname'
http://localhost:8080/username
HTTP/1.1 403 Forbidden
```

```
Date: Sun, 17 Mar 2013 11:38:31 GMT
Status: 403 Forbidden
Connection: close
Content-
Type: text/plain
Set-
Cookie:
rack.session=BAh7B0kiD3Nlc3Npb25faWQGOgZFRiJFOWQyODAxMGI0ZmEzMDY0ZTg3NWE0
%0ACWNzcmYGOwBGIkU1ZGJkZTJiYzI0MGU5OWI0MGVmNjA4MzlkZGRjYWVkOGQ4%0AYjE5OWV
%0A--
8ea7d8577d4d9d7d44fc1ecd2ca1d581de344670; path=/;
HttpOnly
Transfer-Encoding: chunked

Forbidden
```

We called `curl` with the `-i` parameter, indicating we want to see the headers that are returned accompanying the text of the response. We also submitted the value `newname` for `username` using the `-d` parameter.

So, the request was not accepted and the `Rack::Protection:AuthenticityToken` middleware sent back a response status of `403 Forbidden`.

When this happens, you'll get the following in the logs:

```
W, [2013-03-17T12:38:31.957791 #18735]  WARN -- :
attack prevented by
Rack::Protection::AuthenticityToken
127.0.0.1 - - [17/Mar/2013 12:38:31]
"POST /username HTTP/1.1" 403 - 0.0018
```

Handlers, routes, and parameters

Handlers are the blocks of code that are run when matching web requests are received. This matching is done based on routes.

Routes are the paths that your application responds to, paired with the HTTP verb, such as `GET` or `POST`, that a handler is defined to respond to. So if your application is on `www.example.com` and responds when you type in `http://example.com/some/page` into the browser's address bar, `GET /some/page` is one of your application routes.

Parameters are the values that are passed to your handlers according to the request that is received.

Parameters may be part of the path itself, often a number indicating a single item as in `/books/5`.

They may follow a `?` symbol, often indicating a specific way the user wants content prepared for the page; these are called **query parameters**. One famous example is in Google searches. Consider `/search?q=sinatra`, where the search term `sinatra` is indicated by a query parameter `q`.

Handlers

As we have seen throughout the book, handlers are defined in the following way:

```
get '/photos' do
  ...
end
```

Routes

A route is the part of the handler definition that indicates the HTTP verb (for example, `get`) and the path that the handler responds to.

When your Sinatra application receives a request, it tries to find a handler with a route that matches the request's HTTP verb and path.

For example, if your application is hosted on `www.mydomain.com` and responds to the calls to `http://mydomain.com/photos`, `/photos` is one of its routes.

The route `/photos` matches HTTP requests, such as `http://mydomain.com/photos` or even `http://mydomain.com/photos?since=20121225` using the `GET` verb. It does not match `POST` requests, and it does not match `GET` requests to `http://mydomain.com/photos/1` either.

First to last

The routes that an application defines are searched in the order they are defined in. Sinatra searches through the list, and the first one that matches the request is executed.

Parameters

Your routes will often not be as simple as `GET /photos`; for example, you will want to handle requests for specific items via its ID, that is, `GET /photos/935`. In order to handle these variable elements in routes, we use parameters.

What follows is a list of the possible ways you can handle extracting parameters from routes.

The params variable

The variable `params` holds the values of the parameters named in the route.

An example of this is as follows:

```
get '/examples/params/:id' do
  params[:id]
end
```

This handler will return the number 12 when called with `/examples/params/12`.

block parameters

Consider the following example:

```
get '/examples/block_parameters/:id' do |id|
  ...
end
```

The block parameter `id` holds the value of the parameter. Whether you use the `params` variable or block parameters is mainly a matter of personal taste —they both achieve the same end result. Having the block parameter in the preceding example is just like putting the following at the beginning of the block:

```
id = params[:id]
```

Query parameters

The query part of a request path is any section after `?`.

Query parameters do not affect route matching, so a request to `/search?q=hello&lang=en` would match the following handler:

```
get '/search' do
  ...
end
```

And `params` would be a hash such as `{'q' => 'hello', 'lang' => 'en'}`.

Wildcard or splat parameters

Route definitions that are strings and contain the `*` symbol allow us to match groups of characters.

```
get '/example/*/part/*' do |section, part|
  ..
end
```

The block parameters `section` and `part` will hold the first and second matched elements.

You can get the same result without the block parameters:

```
get '/example/*/part/*' do
  ..
end
```

Here `params[:splat][0]` and `params[:splat][1]` will hold the two values. Note that this means you should avoid using `:splat` as a parameter name.

Optional parameters

Optional parameters are indicated by `?` in the route matcher. You may use these to match a number of different formats, such as HTML and JSON.

```
get '/document/:id.?:format?' do
  ...
end
```

This will match `/document/1` and also match the same route with a dot and a format; for example, `/document/1.json`.

When the request is for `/document/1.json`, the value of `params[:format]` is `json`; when it is for `/document/1`, the value of `params[:format]` is `nil`.

Regular expressions

If the preceding methods for describing route matches are not sufficient, you can use regular expressions.

> **Note**
>
> Note that regular expression literals created with `/.../` may become hard to read when matching against paths containing `/`, so it is

recommended that you use Ruby's `%r{...}` literal syntax for route matching expressions.

The following is an example that matches `/words/hello` but not `/words/999`:

```
get %r{/words/([a-z]+)} do
  params[:captures]
end
```

As we used parentheses, `params[:captures]` holds the matched word. So when called with `/words/thing`, this example produces `thing`.

Note that, like `splat`, `captures` should be avoided as a parameter name.

Route conditions

Route matching can also be controlled by matching conditions.

The provides condition

The `provides` condition checks the `Accept` header sent with the web request.

For example, in these routes, only the first responds to requests for JSON:

```
get '/provides', provides: 'json' do
  '{result: "You asked for JSON"}'
end

get '/provides' do
  'This is the default response'
end
```

If we declared those two routes in the inverse order, the one without the condition would respond to all the requests and the one with the `provides` condition would never get called.

Note that a browser will normally issue requests indicating that it will accept any format, so to test the previous route, we can use `curl` from the command line:

```
$ curl -H 'Accept: application/json'
http://0.0.0.0:3000/provides
{result: "You asked for JSON"}
```

```
$ curl -H 'Accept: text/html'
http://0.0.0.0:3000/provides
```

This is the default response

Sinatra comes with the following conditions predefined:

- `:agent/:user_agent`: The request's user agent must match
- `:host_name`: The request's host must match
- `:provides`: The request's `preferred_type` must match

Templating

There are two ways of returning content in your response: returning strings of text (or HTML) and using templates.

Inline templates

If your application is very simple, you can keep your templates in the same file as the rest of the application.

Here's an example of an inline template:

```
require 'sinatra/base'
require 'slim'

class MyApplication < Sinatra::Base
  configure do
    enable :inline_templates
  end

  get '/inline' do
    slim :my_inline_template
  end
end
__END__

@@my_inline_template
h1 This is an inline template
```

Note that the template is defined after `__END__`. The section of a Ruby file after the `__END__` marker can be accessed by the program but is not interpreted as Ruby code. Sinatra makes use of this fact to "hide" templates inside the application files.

HTML templates

Templates are used to create the HTML content that you send back in response to a web request. There are a number of competing templating

engines, from ERB (which is the oldest and most tightly related to the history of Ruby) to Slim, which is recent and very popular due to its simplicity.

In Sinatra (by default), templates are kept in the `views` directory.

All templating engines provide a way of laying out the structure of a web page, substituting values from variables, including blocks of HTML from other files (called **partials**), and looping through lists and conditionally including blocks of text.

In this book, we have been using Slim as the templating engine in all examples.

Slim

You can add Slim to your project by adding the following line to your Gemfile:

```
gem 'slim'
```

As Sinatra defaults to using ERB, you'll need to indicate that you want to use Slim in your application file:

```
require 'slim'
```

Slim indicates the structure of the page via indentation. An example page called `views/home.slim` is as follows:

```
div class="container"
  h1 My Slim Template
```

In your application file in a handler, you tell Sinatra to use this template as follows:

```
slim :home
```

When you need to pass values to a template, you can use the `:locals` Hash. Here's an example where we pass a name to the template and display it.

Include the following in your application file:

```
get '/hello'
  slim :hello, locals: {name: "Joe"}
end
```

Next, create a template called `views/hello.slim`:

```
h1 "Hello #{name}!"
```

When you call this handler, you'll see the following output:

```
Hello Joe!
```

Layouts

Normally you will want to have a consistent look for all of the pages in your application. To achieve this, you will want each page to be wrapped in some standard HTML called the `layout`.

To achieve this you simply need to create a file called `layout` with the correct extension `layout.slim` in the `views` directory.

```
views/layout.slim
doctype html
html
  head
    title My Sinatra Application
  body
    p My header
    == yield
    p My footer
```

Notice the line `== yield`. As we saw in the previous section, the HTML returned by your handler will be inserted at that point in the layout.

Sending e-mail

One common task you will need to perform in an application is to send e-mail.

Let's create a contact form.

First, you need to make sure that you can send e-mail from your web server. How you do so varies according to the type of web server you have. If you have command-line access to your server (for example, via the `ssh` command), try the following command:

```
echo 'Email works' | sendmail YOUR_EMAIL ADDRESS
```

If you get e-mail to your account, sending e-mail works; otherwise, contact whoever provides your web server to ask how to set it up.

Here's how you can achieve this using the `pony` gem.

Add the following to your Gemfile:

```
gem 'pony'
```

Now let's add a view that accepts contact information.

The following is the view; save it as `views/contact.slim`:

```
1 form method="post" action="/email"
2   input name="email" type="email"
placeholder="Your email..."
3   br
4   input type="submit" value="Contact me!"
```

When the user clicks on the **Contact me!** button, the application will receive an HTTP `POST` request to the path `/email`.

Now edit `address-book.rb`.

Include the following at the top:

```
require 'pony'
```

Now we need two handlers; we need the following one to show the new view:

```
1   get 'contact'
2     slim :contact
3   end
```

The other handler receives the data from the form and sends the e-mail (substitute the `from` and `to` e-mail addresses with your own):

```
01   post '/email' do
02     Pony.mail(
03       to: 'YOUR_EMAIL_ADDRESS',
04       from: 'YOUR_EMAIL_ADDRESS',
05       subject: 'Contact request',
06       body: "Email: #{params[:email]}"
07     )
08
09     "Email received, we'll be contacting you
shortly!"
```

```
10    end
11 end
```

Lines 2-7 set up and send an e-mail.

Note that in line 6, we create the body of the e-mail by substituting in the e-mail that the user typed into the form.

Logging

Sinatra logging is done by `Rack::Logger`, which in turn wraps the standard Ruby Logger.

By default, logging output is sent to the Terminal.

When the `:logging` setting is set to a `true` value, an instance of `Rake::Logger` is used to write log entries. The amount of logging is controlled by the logging level; this defaults to `::Logger::INFO`.

You can control the amount of output by setting `:logging` to one of the following values:

- `::Logger::FATAL`: This shows only those errors that cause the program to crash
- `::Logger::ERROR`: This shows all errors
- `::Logger::WARN`: This shows errors and warnings
- `::Logger::INFO` (default): This shows general program information as well as errors and warnings
- `::Logger::DEBUG`: This shows all loggable messages

As a rule of thumb, production systems usually have logging set to `INFO`, while under development, `DEBUG` is preferred.

How to write logs to a file

In `config.ru`, add the following:

```
require 'logger'
class MyLogger < ::Logger
  alias :write :<<
end
log = MyLogger.new('sinatra-app.log')
use Rack::CommonLogger, log
```

This will cause all log entries to be written to the `sinatra-app.log` file.

Note

The Sinatra logging system expects to call a method `write` on `Logger`. As the standard Ruby Logger does not have this method, we subclass `Logger` as `MyLogger` and add the method by making `write` an alias of `<<`, which Logger already implements.

Request

Inside handlers, `request` gives you access to a `Sinatra::Request` object containing the details of the web request; for example, cookies, the URL, and the host.

So this example will respond with `/some/path`:

```
get '/some/path' do
  request.env['REQUEST_PATH']
end
```

Response

Also available inside handlers is `response`, which holds the response that Sinatra will return to Rack. Useful attributes of the response are `body`, `header`, and `status`.

In this example, we'll first set the `Content-type` header and then send it back as our actual response:

```
get '/response' do
  headers 'Content-type' => 'text/plain'
  response.header['Content-type']
end
```

So the response will be `text/plain`.

Status

A response status is a standard HTTP numeric status code; see the Wikipedia reference at http://en.wikipedia.org/wiki/List_of_HTTP_status_codes. Understanding a few status codes is essential for web application development.

Let's look at a few examples using `curl` from the command line so we can see the response status.

If Sinatra handles a request and returns a web page without error, the status will be `200`, also known as `OK`.

So with the following handler:

```
get '/this_works' do
  'It worked!'
end
```

The following is what we get with `curl`:

```
1 curl -i http://0.0.0.0:3000/this_works
2 HTTP/1.1 200 OK
3 Date: Sun, 28 Apr 2013 16:16:43 GMT
4 Status: 200 OK
5 Connection: close
6 Content-Type: text/html;charset=utf-8
7 Content-Length: 10
8
9 It worked!
```

The bits to notice are as follows:

- Line 1: Here we make the request for `/this_works`
- Line 2: Here the response comes back with the status `200`, which means `OK`
- Line 9: Here it shows `It worked!`, which is the actual text that would get displayed by a browser

Now if we request something that the application doesn't have a handler for, we get a different response status: `404 Not Found`:

```
1 curl -i http://0.0.0.0:3000/this_doesnt_exist
2 HTTP/1.1 404 Not Found
3 Date: Sun, 28 Apr 2013 17:13:21 GMT
4 Status: 404 Not Found
5 Connection: close
6 Content-Type: text/html;charset=utf-8
7 Content-Length: 451
8
9 <snip/>
```

Notice line 2: `404 Not Found`. I have snipped the actual web page that Sinatra sends back. In the development mode, this is a helpful page that indicates what's wrong and how to fix it.

Using a database

When your application needs to store information, you'll need to create a database, connect to it, and read and write data.

There are a number of database connectors available. Often, libraries are used that go beyond simply connecting to the database; they create a mapping between rows in the database and objects in your program. These are called **object-relational mappings**. The most commonly used are **ActiveRecord**, which was created as part of the Ruby on Rails project, and **DataMapper**. The following examples will use DataMapper.

Setup

Add the following lines to your Gemfile:

```
gem 'data_mapper'
gem 'dm-sqlite-adapter'
```

Next, update the gems as usual:

```
bundle
```

Create a database

We will be creating a SQLite database. We want one table called `addresses`.

We'll edit `address-book.rb`. Add the following line at the top with the other `require` statements:

```
require 'data_mapper'
```

Next, before the declaration of the class `AddressBook`, add the following class:

```
01 class Address
02   include DataMapper::Resource
03   property :id,     Serial
04   property :name,   String
05   property :street, Text
06 end
```

In line 1, we call the class `Address`; DataMapper will transform that name (making it lowercase and pluralizing it) into `addresses` and will look for a table by that name in the database.

In line 2, we add in all the code that will allow us to use objects of the `Address` class to transparently read and alter the data in our database table.

In line 3 we add an `id` column; the database will automatically assign unique numeric values to `id`. This is the mechanism that will allow us to keep tabs on the various addresses we create.

Lines 4 and 5 create the actual values of the address, the person's name, and the street where he/she lives. Feel free to flesh this out by adding `city`, `zipcode`, `region`, and so on.

Now add the following code inside the `AddressBook` class:

```
configure do
    DataMapper.setup(:default,
"sqlite3://#{settings.root}/address-book.sqlite3")
    DataMapper.auto_upgrade!
end
```

This code will create (if necessary) and connect to our database. The database will be created in the root directory of our project and called `address-book.sqlite3`.

Insert an address

In order to insert data, we need an HTML form to insert the values and a handler to receive the HTTP `POST` request.

Create a directory `views/addresses`; inside it create `views/addresses/new.slim`:

```
01 form  method="post" action="/addresses"
02   fieldset
03     legend Add an address
04     input name="authenticity_token"
value=session[:csrf] type="hidden"
05     p
06       input type="text" name="address[name]"
placeholder="Name..."
07     p
08       input type="text" name="address[street]"
placeholder="Street..."
09     p
10       input type="submit" value="Create"
```

In line 1, we create an HTML form. When the user submits the form (by pressing the *Enter* key or clicking on the **Create** button), the form will be sent to our application as a `POST` request to the path `/addresses`.

Add the following handlers to `address-book.rb`:

```
1   get '/address/new' do
2      slim :'addresses/new'
3   end
4
5   post '/addresses' do
6     address = Address.new(params[:address])
7     address.save
8   end
```

Lines 1-3 simply call the `new` view when we go to `/addresses/new`.

In line 5, we set up the handler to respond to the form's `POST` request.

The values sent with the `POST` request are as always contained in the `params` variable. In `params[:address]`, there exist the values the user typed into the form. It will be a hash, for example:

```
{:name => "Some name", :street => "15 Green
Street"}
```

In line 6, we take these values and create an `Address` object with them.

At this point, `address.name` will contain `Some name` and `address.street` will contain `15 Green Street`.

In line 7, we save the new address to the database.

For a fuller example, look at the source code in the sample `address-book` application.

The Sinatra DSL

What follows is a list of the methods that Sinatra provides to define your application. Note that entries marked (internal) are normally not used by applications.

- `add_filter` (internal): This is used by `before` and `after` to create filters.
- `after`: This is a block of code to be run after handlers.
- `before`: This is a block of code to be run before handlers.
- `build` (internal): This is used to set up the application.

- `call` (internal): This is the method that Rack calls when passing a request to the application.
- `caller_files`: This is a simplified application call stack, listing just the filenames.
- `caller_locations`: This is like `caller_files`, but lists only filenames and line numbers.
- `configure`: This is a block of code to be run once on application startup.
- `delete`: This matches HTTP `DELETE` requests. See the *Routes* section.
- `development?`: This is true if the application is in the development mode (that is, the `environment` setting is set to `:development`).
- `disable`: See the *Settings* section.
- `enable`: See the *Settings* section.
- `error`: This defines the code to be called when the application returns an error. This is not the same as the `error` helper method, which is callable within handlers (see the *Helpers* section).
- `extensions`: This is a list of extension modules (see the `register` command).
- `find_template` (internal): This is used for locating a template.
- `forward` (middleware): This passes the request on to the application.
- `get { route_block }`: This defines a handler matching HTTP `GET` requests. Note that `get` handlers automatically define an equivalent `head` handler. See the *Routes* section.
- `halt response`: This stops the processing and returns the given response.
- `head`: This matches HTTP `HEAD` requests. See the *Routes* section.
- `helpers`: This defines a set of helper methods for handlers and templates.
- `layout`: This is a block of code returning the layout markup.
- `link`: This matches HTTP `LINK` requests. See the *Routes* section.
- `mime_type` (internal): This contains get or set Rack MIME types.
- `mime_types`: This is a query for the MIME types matching a value.
- `not_found`: This stops the processing and returns a `Not Found` error.
- `options`: This matches HTTP `OPTIONS` requests. See the *Routes* section.
- `pass`: This skips the current handler and searches for another handler that matches the request.
- `patch`: This matches HTTP `PATCH` requests. See the *Routes* section.
- `post`: This matches HTTP `POST` requests. See the *Routes* section.
- `production?`: This shows if we are in the production mode (see the definition for `development?`).
- `prototype` (internal): This is the application/middleware instance.
- `put`: This matches HTTP `PUT` requests. See the *Routes* section.

- `quit!`: This shuts the application down.
- `register`: This registers an extension module.
- `request`: This is the request object.
- `reset!` (internal): This resets the application or middleware's internal state.
- `run!` (internal): This starts the application.
- `set`: See the *Settings* section.
- `settings`: See the *Settings* section.
- `template`: This defines a named template.
- `test?`: This shows if we are in the test mode (see the definition for `development?`).
- `use`: This adds middleware to the application.

Helpers

Helpers are methods that you can call inside handlers and templates. A list of helpers is as follows:

- `attachment`: This returns the given file, prompting the user to save it.
- `back`: This returns the address the request referred, allowing us to say `redirect back`.
- `body` (string/block): This, when passed a value or a block, sets the body of the response. When called without a value, it returns the response body.
- `cache_control`: This sets the `Cache-Control` header for the response.
- `client_error?`: This shows if we are returning a 4xx error (see the *Response* section).
- `content_type`: This sets or gets the response's `Content-Type` header.
- `error`: This stops processing the request and returns an HTTP error.
- `etag`: This sets the response's `ETag` header and behaves correctly according to relevant `HTTP_IF(_NONE)_MATCH` headers in the request.
- `expires`: This sets the response's `Expires` header.
- `headers`: This sets response headers.
- `informational?`: This shows if we are returning a 1xx response (see the *Response* section).
- `last_modified`: This sets the response's `Last-Modified` header.
- `logger`: This returns the currently configured logger.
- `mime_type`: This queries Rack for a MIME type.
- `not_found?`: This shows if we are returning a 4xx response (see the *Response* section).
- `redirect`: This stops the processing and sends a response to the client, telling it to go to another location.

- `redirect?`: This shows if we are returning a 3xx status (see the *Response* section).
- `send_file`: This sends the file indicated as the body of the response.
- `server_error?`: This shows if we are returning a 5xx status (see the *Response* section).
- `session`: This returns the Rack session object.
- `status`: This sets or gets the response status (see the *Response* section).
- `stream`: This starts streaming data to the client.
- `success?`: This shows if we are returning a 2xx status (see the *Response* section).
- `time_for`: This converts various objects into `Time` values.
- `to`: This is an alias for `uri`.
- `uri`: This returns an Internet address for a given path.
- `url`: This is an alias for `uri`.

Settings

A running Sinatra application has a collection of settings. Many settings will be the standard ones defined by Sinatra (see the following list), but your application can also define its own settings using the `set`, `enable`, and `disable` commands.

Settings can be accessed via the `settings` keyword. So, the following will give you access to the current value of the `environment` setting:

```
settings.environment
```

Settings can be set to any value with `set`:

```
set :setting, value
```

Boolean settings can be set to `true` with `enable`:

```
enable :setting1, :setting2, …
```

Likewise, they can be set to `false` with `disable`:

```
disable :setting1, ...
```

`configure` can be used to create groups of settings that differ according to the environment.

```
configure :development do
  enable :dump_errors
end
```

This means that the `:dump_errors` setting will only be set to `true` for the `development` environment.

The following is a list of standard Sinatra settings and their effects. Adjusting these settings allows us to modify the behavior of our application.

- `absolute_redirects`: This tells us if the domain should be prepended when doing redirects.

  When true, the location header of the response will be of the form:

  ```
  Location: http://0.0.0.0/some/path
  ```

  When false:

  ```
  Location: /some/path
  ```

  Default value: true.

- `add_charset`: This, if true, add the request's character set to `params` as `params[:charset]`. See also `default_encoding`.
- `app_file`: This holds the name of the file that starts the application.
- `bind`: This holds the IP address the server has to bind to. The default value is `0.0.0.0`.
- `default_encoding`: This is the default character encoding for requests. The default value is `utf-8`. See the definition for `add_charset`.
- `dump_errors`: This, if true, reports any errors back to Rack via the `rack.errors` value. The default value is `false` for the test environment, otherwise it is `true`.
- `empty_path_info`: This allows handling requests for the path `/` via the definition of a route with an empty path (that is, `get do; end`).
- `environment`: This modifies various behaviors of the application. The default value is `:development`, but it takes the value of the environment variable `RACK_ENV` if set. It should only be set to one of the following: `:development`, `:test`, or `:production`.
- `inline_templates`: If this has a value of `true`, the section of the application file after `__END__` is searched for templates. If a file

path is supplied, the given file is searched instead of the application file. The default value is `false`; it is set to `true` for Classic applications.

- `lock`: This, if true, stops more than one request being run via the application at a time. This setting should be set to true if some part of your application has unpredictable behavior when modified by more than one process at a time. The default value is `false`.
- `logging`: In Classic applications, `logging` is set to `true` except under the `test` environment. See the *Logging* section. The default value is `false`.
- `method_override`: This setting exists to handle requests from old browsers that cannot send any requests except `GET` or `POST`. In post requests, it interprets a `_method` parameter as an indication of the HTTP verb. When active, you can add a hidden input field called `_method` to your forms. In this way, you can issue `PUT`, `DELETE`, and other requests. The default value is `false`.
- `port`: This is the port the server should listen to (also see `bind`).
- `prefixed_redirects`: This, if true, adds the script name to the `Location` header when redirecting. The default value is `false`.
- `protection`: This, if true, uses `Rack::Protect` to protect the application against common attacks. The default value is `true`.
- `public_folder`: This is the directory where Sinatra looks for static files (for example, images and stylesheets). If a static file corresponding to a request is not found, your application is invoked. The default value is `root + /public`.
- `raise_errors`: If this is false, Sinatra's error handlers are run when an error occurs; if `true`, errors are returned directly to Rack. The default value is `true` for the test environment, otherwise it is `false`.
- `reload_templates`: This clears the cache of templates before each request. The default value is `true` for development, otherwise it is `false`.
- `root`: This the root path of the application. The default value is the directory containing `app_file`.
- `run`: This starts a web server when the application is run.
- `running`: This this is set to `true` when Sinatra starts the web server. The default value is `false`.
- `server`: This is a list of possible servers that you can use. Sinatra tests for the presence of each in turn and uses the first that it finds. The default value is `thin`, `mongrel`, and `webrick`. (`thin` is not available on Jruby, so `mongrel` is used.)
- `sessions`: This creates a session and returns it with the response as a cookie. The default value is `false`.
- `show_exceptions`: This returns details as a web page when errors occur. The default value is `true` for the development environment, otherwise it is `false`.

- `static`: If this is set to `true`, Sinatra checks if the requests are for the files found in the public directory—the directory that has to be searched for static files. The default value is `true` if the directory indicated by `:public_folder` exists.
- `static_cache_control`: This when set to a `true` value adds the supplied values to the response headers when responding with static files. The default value is `false`.
- `threaded`: This tells the web server to run in the threaded mode. This setting currently only affects the `thin` server where up to 20 requests can be handled concurrently. The setting is ignored by other `servers`. The default value is `true`.
- `views`: This is the path where Sinatra looks for templates. The default value is `root directory + /views`.

This section has presented many of the key concepts of Sinatra. You will have learned to set up templates, handle file uploads, and write data to a database.

People and places you should get to know

If you need help with Sinatra, here are some people and places that will prove invaluable.

Official sites

The following is a list of sites that will give you useful reference information about Sinatra:

- Homepage: http://www.sinatrarb.com/.
- Manual and documentation:
    - http://www.sinatrarb.com/documentation.
    - http://rubydoc.info/gems/sinatra.
    - http://sinatra-book.gittr.com/—a step-by-step presentation of the framework and a lot of code examples
- Wikipedia article: http://en.wikipedia.org/wiki/Sinatra_%28software%29.
- Blog: http://www.sinatrarb.com/blog.html.
- Source code: https://github.com/sinatra/sinatra—Sinatra's source code is short and clear. Try reading through it and you'll get a lot of insight into how the framework works.
- Blake Mizerany is the original creator of Sinatra and continues to work on related projects. His GitHub code can be found at https://github.com/bmizerany.

Articles and tutorials

- There is a series of nice tutorials by Net Tuts+ on Sinatra. This example shows how to use Sinatra on the server with the `Knockout.js` client-side framework: http://net.tutsplus.com/tutorials/javascript-ajax/building-single-page-web-apps-with-sinatra-part-1/.
- A useful but incomplete guide is available at http://sinatra.restafari.org/book.html.

Community

- Official mailing list: https://groups.google.com/forum/?fromgroups#!forum/sinatrarb.
- Official IRC channel: irc://irc.freenode.net/#sinatra.
- User FAQ: http://www.sinatrarb.com/faq.html.

- StackOverflow has a growing number of questions about Sinatra: http://stackoverflow.com/questions/tagged/sinatra.
- Read some Sinatra code on GitHub: https://github.com/search?q=sinatra.

Blogs

Konstantin Haase's blog can be found at http://rkh.im/. Unfortunately, the site doesn't seem to have a list of blog articles, but the most recent ones are listed on the front page.

Twitter

- The Sinatra project has an official Twitter account: https://twitter.com/sinatra
- Sinatra's maintainer Konstantin Haase's account: https://twitter.com/konstantinhaase
- Follow Blake Mizerany on Twitter: https://twitter.com/bmizerany
- For more open source information, follow Packt Publishing at http://twitter.com/#!/packtopensource