```
  .
  .
  # Force all access to the app over SSL, use Strict-Transport-Security,
  # and use secure cookies.
  config.force_ssl = true
  .
  .
  .
end
```

At this stage, we need to set up SSL on the remote server. Setting up a production site to use SSL involves purchasing and configuring an *SSL certificate* for your domain. That's a lot of work, though, and luckily we won't need it here: for an application running on a Heroku domain (such as the sample application), we can piggyback on Heroku's SSL certificate. As a result, when we deploy the application in Section 7.5.2, SSL will automatically be enabled. (If you want to run SSL on a custom domain, such as www.example.com, refer to Heroku's documentation on SSL.)

7.5.2 Production webserver

Having added SSL, we now need to configure our application to use a webserver suitable for production applications. By default, Heroku uses a pure-Ruby webserver called WEBrick, which is easy to set up and run but isn't good at handling significant traffic. As a result, WEBrick isn't suitable for production use, so we'll replace WEBrick with Puma, an HTTP server that is capable of handling a large number of incoming requests.

To add the new webserver, we simply follow the Heroku Puma documentation. The first step is to include the `puma` gem in our **`Gemfile`**, but as of Rails 5 Puma is included by default (Listing 3.2). This means we can skip right to the second step, which is to replace the default contents of the file **`config/puma.rb`** with the configuration shown in Listing 7.35. The code in Listing 7.35 comes straight from the Heroku documentation,[14] and there is no need to understand it (Box 1.2).

[14] Listing 7.35 changes the formatting slightly so that the code fits in the standard 80 columns.

Listing 7.35: The configuration file for the production webserver.
config/puma.rb

```
# Puma configuration file.
max_threads_count = ENV.fetch("RAILS_MAX_THREADS") { 5 }
min_threads_count = ENV.fetch("RAILS_MIN_THREADS") { max_threads_count }
threads min_threads_count, max_threads_count
port        ENV.fetch("PORT") { 3000 }
environment ENV.fetch("RAILS_ENV") { ENV['RACK_ENV'] || "development" }
pidfile ENV.fetch("PIDFILE") { "tmp/pids/server.pid" }
workers ENV.fetch("WEB_CONCURRENCY") { 2 }
preload_app!
plugin :tmp_restart
```

We also need to make a so-called **`Procfile`** to tell Heroku to run a Puma process in production, as shown in Listing 7.36. The **`Procfile`** should be created in your application's root directory (i.e., in the same directory as the **`Gemfile`**).

Listing 7.36: Defining a **`Procfile`** for Puma.
./Procfile

```
web: bundle exec puma -C config/puma.rb
```

7.5.3 Production database configuration

The final step in our production deployment is properly configuring the production database, which (as mentioned briefly in Section 2.3.5) is PostgreSQL. My testing indicates that PostgreSQL actually works on Heroku without any configuration, but the official Heroku documentation recommends explicit configuration nonetheless, so we'll err on the side of caution and include it.

The actual change is easy: all we have to do is update the **`production`** section of the database configuration file, **`config/database.yml`**. The result, which I adapted from the Heroku docs, is shown in Listing 7.37.