In this chapter, you'll see:

- Running our application in a production web server
- Configuring the database for PostGreSQL
- Securely deploying secrets
- Placing all of the above into Docker containers

CHAPTER 17

Task L: Deployment and Production

Deployment is supposed to mark a happy point in the lifetime of our application. It's when we take the code that we've so carefully crafted and upload it to a server so that other people can use it. It's when the beer, champagne, and hors d'oeuvres are supposed to flow. Shortly thereafter, our application will be written about in *Wired* magazine, and we'll be overnight names in the geek community.

The reality, however, is that it often takes quite a bit of up-front planning to pull off a smooth and repeatable deployment of your application.

A bewildering number of options are available for deployment: Ansible, Capistrano, Chef, and Puppet are all popular choices. Covering all of them would be the subject of several books. We're going to focus this chapter on what effectively is the defacto standard for cloud deployment: Docker.

If you're not familiar with Docker images, they're essentially self-contained and portable runtimes that can be deployed by pretty much any cloud-hosting provider. This means you can build and test your deployment locally and then choose your cloud provider later, and even change your mind and move hosts at any time.

At the moment, we've been doing all of our work on one machine, though user interaction with our web server *could* be done on a separate machine. On our machine you've been making use of the Puma web server, SQLite 3, various gems you've installed, and your application code. Your code may or may not have also been placed in Git by this point.

For deployment, we're going to make use of two Docker containers. The web server container will be running a combination of nginx[1] and Phusion

1. https://www.nginx.com/

Passenger.[2] This code will access a PostgreSQL database running in a separate container.[3]

That's a lot of moving parts! To help us keep track of them all, we'll be using Bundler to manage our dependencies and Docker Compose as the tool to manage the containers.[4] And yet, despite all the moving parts, the overall definition is remarkably compact:

rails7/depot_td/docker-compose.yml

```
version: "3.8"

services:
  db:
    image: postgres:14
    volumes:
      - pgdata:/var/lib/postgresql/data
    environment:
      POSTGRES_PASSWORD: password

  web:
    build: .
    volumes:
      - ./log:/home/app/depot/log
    secrets:
      - source: master_key
        target: /home/app/depot/config/master.key
    ports:
      - "8001:80"
    depends_on:
      - db

secrets:
  master_key:
    file: ./config/master.key

volumes:
  pgdata:
```

Don't let the size of the file fool you—there's a lot to unpack here. We'll cover the following in subsequent sections:

- Configuring the database
- Keeping secrets
- Building a docker image
- Deploying the application

2. https://www.phusionpassenger.com/
3. https://www.postgresql.org/
4. https://docs.docker.com/compose/

Configuring the Database

The SQLite website is refreshingly honest when it comes to describing what this database is good at and what it's not good at.[5] In particular, SQLite isn't recommended for high-volume, high-concurrency websites with large datasets. And, of course, we want our website to be such a website. Plenty of alternatives to SQLite, both free and commercial, are available. We'll go with PostgreSQL.

Looking at the `docker-compose.yml` file, the first service listed is named `db`. This name serves as the host name for the container. Each service defined in the container is deployed on a private network isolated from the rest of the world except for ports that you decide to expose. Inside the network, services refer to each other by host name.

On the `db` host we run the stock postgres docker image[6] from Docker Hub.[7]

On top of that image, we mount a disk volume that we've named `pgdata`. This volume will be created as a file within the filesystem of the machine running the container. Placing the contents of the database outside of the container allows us to update the software image without affecting the data.

Finally, we define an environment variable containing the password we use to access the database. As this image has no external ports defined, and therefore no way to access it outside of the isolated environment that docker compose provides, this doesn't concern us for now. We may need to revisit once we deploy this application on a public cloud with a different configuration.

Now that we've defined the database container, we change the configuration of the production database from using the sqlite3 adapter to using the postgresql adapter:

rails7/depot_td/config/database.yml

```
# SQLite. Versions 3.8.0 and up are supported.
#   gem install sqlite3
#
#   Ensure the SQLite 3 gem is defined in your Gemfile
#   gem "sqlite3"
#
default: &default
  adapter: sqlite3
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
  timeout: 5000
```

5. http://www.sqlite.org/whentouse.html
6. https://hub.docker.com/_/postgres
7. https://hub.docker.com/

```
development:
  <<: *default
  database: db/development.sqlite3

# Warning: The database defined as "test" will be erased and
# re-generated from your development database when you run "rake".
# Do not set this db to the same as development or production.
test:
  <<: *default
  database: db/test.sqlite3

production:
  database: depot
  adapter: postgresql
  encoding: unicode
  host: db
  username: postgres
  password: password
  pool: 5
```

In addition to specifying the adapter name, we define the database name, the host where the database can be found, the username and password used to authenticate access, and two configuration options: the character encoding to be used and the number of database clients to allocate to service requests.

We need one more step to prepare our deployment to use the postgresql adapter, namely to install the adapter itself:

```
$ bundle add pg --group production
```

Conceptually we've defined how to provision a complete virtual machine to host our data needs—all with a few lines of YAML. We don't need to worry about what operating system that container is running or any other platform details. Welcome to the world of containers!

Managing Secrets

When we defined our seed data back in Iteration A2: Making Prettier Listings, on page 77, our database didn't have any users. After we added our first user in Chapter 14, Task I: Logging In, on page 207, we added code to require a valid login to access pages that update the database.

What that means is that if we were to deploy a new installation starting with the seed data alone, we would be locked out of our own application. That's not good. So lets fix it!

Adding an initial user to our seed data solves the problem, but checking in a password into our version control and deploying it is hardly secure.

Fortunately Rails has provided a way to encrypt secrets such as this one. Rails calls such secrets credentials.

We get started by editing our credentials:

```
$ EDITOR='code --wait' rails credentials:edit
```

Feel free to replace the editor with vim or another editor of your choice.

You'll see that Rails has already defined one credential that's used to encrypt cookies, which is how Rails implements sessions such as the one used to track a user's cart. Leave that credential alone, and add another one to the file:

```
dave_password: secret
```

When you save the file, Rails will update config/credentials.yml.enc using the master key defined in config/master.key. The encoded file can be checked into version control and shared publicly. The key, however, needs to be kept private.

If you look into the .gitignore file, you'll see that /config/master.key is already listed there. For similar reasons, we won't want the key to be placed into the docker image that we'll be creating shortly, so we'll want to create a .dockerignore file with this in it. As the .gitignore is a good starting point, we can simply copy it:

```
$ cp .gitignore .dockerignore
```

Now that we have a credential defined, let's make use of it by adding the following to db/seeds.rb

rails7/depot_td/db/seeds.rb

```
User.create! name: 'dave',
        password: Rails.application.credentials.dave_password
```

So far, we've defined a credential, made use of it, and ensured that the master key won't be committed to version control or placed in the image. The one task remaining is to set things up to deploy the master key at runtime. The following lines in docker-compose.yml take care of this:

rails7/depot_td/docker-compose.yml

```
services:
  web:
    secrets:
      - source: master_key
        target: /home/app/depot/config/master.key
secrets:
  master_key:
    file: ./config/master.key
```

The general pattern of placing a secret in a file, listing all of the secrets you'll be using in one place, and then referencing individual secrets by the containers that use them is common in cloud deployments. Rails makes it easy in that there's only one secret you need for Rails applications, namely a master key. That key can be used to unlock all of the credentials that you'll need.

Building a Docker Image

Before proceeding, let's take one last look at the web service defined in the docker-compose.yml:

rails7/depot_td/docker-compose.yml

```
web:
  build: .
  volumes:
    - ./log:/home/app/depot/log
  secrets:
    - source: master_key
      target: /home/app/depot/config/master.key
  ports:
    - "8001:80"
  depends_on:
    - db
```

We've seen the volumes definition before; in this case it maps the log directory in the container to our local log directory. We previously covered how secrets are managed.

ports is new. This maps port 8001 on our development machine to port 80 in the container. This means that we'll be able to access our application as http://localhost:8001 once it's up and running.

depends_on controls the startup order of the containers. In this case, we want the database to be up and running when we start our appliction.

This leaves one last option: build. While we were able to make use of a pre-built docker image for our database, this won't be the case for the application we just wrote. The value of . here means that the Dockerfile used to build this image can be found in the current directory.

rails7/depot_td/Dockerfile

```
FROM phusion/passenger-full:2.2.0

RUN rm /etc/nginx/sites-enabled/default
RUN rm -f /etc/service/nginx/down
RUN rm -f /etc/service/redis/down
ADD config/nginx.conf /etc/nginx/sites-enabled/depot.conf
```

```
USER app
RUN mkdir /home/app/depot
WORKDIR /home/app/depot

ENV RAILS_ENV=production
ENV BUNDLE_WITHOUT="development test"
COPY --chown=app:app Gemfile Gemfile.lock .
RUN bundle install
COPY --chown=app:app . .

RUN SECRET_KEY_BASE=`bin/rails secret` \
  bin/rails assets:precompile

USER root
CMD ["/sbin/my_init"]
```

This is remarkably short, and that's because we can start with a pre-build image as a starting point. Phusion provides a number of images that you can build upon.[8] We chose full because it includes everything we need. Other images provide different versions of Ruby, and those are "some assembly required" starters.

Before proceeding, feel free to look around the image by running the command provided:[9]

```
$ docker run --rm -t -i phusion/passenger-full bash -l
```

This particular image provides an initialization script (/sbin/my_init) and is designed to be configured by removing and adding files. We proceed to remove the default site, enable nginx and redis, and provide our own site definition:

rails7/depot_td/config/nginx.conf

```
server {
    listen 80;
    server_name www.depot.com;
    root /home/app/depot/public;

    passenger_enabled on;
    passenger_user app;
    passenger_ruby /usr/bin/ruby;

    location /cable {
        passenger_app_group_name /home/app/depot/cable;
        passenger_force_max_concurrent_requests_per_process 0;
    }
}
```

This file starts by giving the web port and server name and identifying where static files that can be served by the web server itself can be found. Then it

8. https://github.com/phusion/passenger-docker#about-the-image
9. https://github.com/phusion/passenger-docker#inspecting-the-image

enables passenger and tells it what Unix user to use to run this app and where the ruby executable can be found. Finally, it configures Action Cable, which passenger runs in a separate process and allows an unlimited number of simultaneous requests. More information on this can be found in the documentation.[10]

We return to the Dockerfile, specifically looking at the portion creating the application:

rails7/depot_td/Dockerfile

```
USER app
RUN mkdir /home/app/depot
WORKDIR /home/app/depot

ENV RAILS_ENV=production
ENV BUNDLE_WITHOUT="development test"
COPY --chown=app:app Gemfile Gemfile.lock .
RUN bundle install
COPY --chown=app:app . .

RUN SECRET_KEY_BASE=`bin/rails secret` \
  bin/rails assets:precompile
```

Most of this is very straightfoward: the user is set, and a directory is created and set as the current working directory. The Rails environment is set to production, the code is copied to the image, and bundle install is run. Note that the Gemfile is copied separately—the reason for this will become clear shortly.

The final command runs assets:precompile.[11] During development, the assets directory is monitored for changes and served dynamically as needed. This is unnecessary overhead once deployed, so Rails provides a command to do this only when necessary. Run this command every time you deploy your appplication if it's possible that one or more assets have changed. The SECRET_KEY_BASE is a work-around to Rails requiring a master key even on commands that won't make use of it.[12]

Before proceeding, we have one last thing we may need to clean up: the passenger-full:2.2.0 image contains Ruby version 3.1.1p18. If you're running with a version of Ruby *other* than 3.1.1, you have three choices:

- Find another image that matches the version of Ruby you're running locally. The Changelog will prove helpful in finding that image.[13]

10. https://www.phusionpassenger.com/library/config/nginx/action_cable_integration/#running-the-action-cable-server-on-the-same-host-and-port-under-a-sub-uri
11. https://guides.rubyonrails.org/asset_pipeline.html#precompiling-assets
12. https://github.com/rails/rails/issues/32947#issuecomment-401886372
13. https://github.com/phusion/passenger-docker/blob/master/CHANGELOG.md

- Install a different version of Ruby on your development machine, perhaps using rbenv, rvm, or chruby. Chapter 1, Installing Rails, on page 3, contains some helpful information on how to do this.
- Remove the .ruby-version file from your project and comment out the ruby line in your Gemfile. This is *not* recommended for production but may be the most expedient way for you to get experience with Docker images.

Getting Up and Running

Three small steps and we're done with our planning. Now it's time to get things up and running. The first thing we need to do is install Docker itself. You can get it at the Docker website.[14]

Next we use docker to build our image for the web service with a single command:

```
$ docker compose build
```

This command will take a while. It will download an image. And most of the remaining time will be spent installing gems. If you run the same command again, it'll run quickly as nothing needs to be redone. If you change any file *other* than your Gemfile, the image will be updated quickly with the change. If you change the Gemfile, run bundle update, and then rerun docker compose build, it'll take longer as it will rerun the bundle install step on a fresh image.

Next we start both the db and web containers with a single command:

```
$ docker compose up
```

Normally this command will be run with the --detach or -d option which will run the containers in the background, but for now it's helpful to see the output.

Once the database is started, create the database, run the migrations, and load the seed data:

```
$ docker compose exec web bin/rails db:create db:migrate db:seed
```

This command only needs to be run once as long as the volume exists. You can list and remove volumes using the docker volume command. To recreate the volume, run docker compose down followed by docker compose up.

At this point, your application is up and running! It can be accessed at http://localhost:8001/.

14. https://docs.docker.com/get-docker/

Checking Up on a Deployed Application

Once we have our application deployed, we'll no doubt need to check up from time to time on how it's running. We can do this in two primary ways. The first is to monitor the various log files output by both our front-end web server and the nginx server running our application. The second is to connect to our application using rails console.

Looking at Log Files

To get a quick look at what's happening in our application, we can use the tail command to examine log files as requests are made against our application. The most interesting data will usually be in the log files from the application itself. Even if nginx is running multiple applications, the logged output for each application is placed in the production.log file for that application.

Assuming that our application is deployed into the location we showed earlier, here's how we look at our running log file:

```
# On your server
$ tail -f log/production.log
```

Sometimes, we need lower-level information—what's going on with the data in our application? When this is the case, it's time to break out the most useful live server debugging tool.

Using Console to Look at a Live Application

We've already created a large amount of functionality in our application's model classes. Of course, we created these to be used by our application's controllers—but we can also interact with them directly. The gateway to this world is the rails console script. We can launch it on our server with this:

```
# On your server
$ docker compose exec web bin/rails console
Loading production environment (Rails 7.0.4)
irb(main):001:0> p = Product.last
=>
#<Product:0x0000004013a47ad8
...
irb(main):002:0> p.title
=> "Modern CSS with Tailwind"
irb(main):003:0> p.price = 29.00
=> 29.0
irb(main):003:0> p.save
=> true
```

Once we have a console session open, we can poke and prod all the various methods on our models. We can create, inspect, and delete records. In a way, it's like having a root console to your application.

Once we put an application into production, we need to take care of a few chores to keep the application running smoothly. These chores aren't automatically taken care of for us, but luckily we can automate them.

Dealing with Log Files

As an application runs, it constantly adds data to its log file. Eventually, the log files can grow extremely large. To overcome this, most logging solutions can *roll over* log files to create a progressive set of log files of increasing age. This breaks up our log files into manageable chunks that can be archived or even deleted after a certain amount of time has passed.

The Logger class supports rollover. We need to specify how many (or how often) log files we want and the size of each, using a line like one of the following in the file config/environments/production.rb:

```
config.logger = Logger.new(config.paths['log'].first, 'daily')
```

Or this is a possibility:

```
require 'active_support/core_ext/numeric/bytes'
config.logger = Logger.new(config.paths['log'].first, 10, 10.megabytes)
```

Note that in this case an explicit require of active_support is needed, because this statement is processed early in the initialization of your application—before the Active Support libraries have been included. In fact, one of the configuration options that Rails provides is to not include Active Support libraries at all:

```
config.active_support.bare = true
```

Alternatively, we can direct our logs to the system logs for our machine:

```
config.logger = SyslogLogger.new
```

Find more options at http://guides.rubyonrails.org/configuring.html.

Moving On to Launch and Beyond

Once we've set up our initial deployment, we're ready to finish the development of our application and launch it into production. We'll likely set up additional deployment servers, and the lessons we learn from our first deployment will tell us a lot about how we should structure later deployments. For example, we'll likely find that Rails is one of the slower components of our system: more of the request time will be spent in Rails than in waiting on the database or

filesystem. This indicates that the way to scale up is to add machines to split up the Rails load.

However, we might find that the bulk of the time a request takes is in the database. If this is the case, we'll want to look at how to optimize our database activity. Maybe we'll want to change how we access data. Or maybe we'll need to custom-craft some SQL to replace the default Active Record behaviors.

One thing is for sure: every application will require a different set of tweaks over its lifetime. The most important activity is to listen to it over time and discover what needs to be done. Our job isn't done when we launch our application. It's actually just starting.

Although our job is just starting when we first deploy our application to production, we've completed our tour of the Depot application. After we recap what we did in this chapter, let's look back at what we've accomplished in remarkably few lines of code.

What We Just Did

We covered a lot of ground in this chapter. We took our code that ran locally on our development machine for a single user and placed it on a different machine, running a different web server, accessing a different database, and possibly even running a different operating system.

To accomplish this, we used a number of products:

- We made use of a stock PostGreSQL container and configured our application to use this as our database server.
- We encrypted and securely deployed application secrets consisting initially of the password of our initial administrator.
- We added our application to a base container that included Phusion Passenger and nginx as a starting point.

Playtime

Here's some stuff to try on your own:

- Instead of using the `passenger-full` docker image, try one of the `passenger-ruby` images. This will enable you to pin down the version of Ruby that you're using. To make this work, you'll need to run redis in a separate container.

- Use Docker secrets instead of having the PostGreSQL password directly in the compose file.[15] Update your Rails application to use this same password.
- Select a hosting provider and deploy your application in the cloud. Most hosting options support containers and provide a free tier for initial experimentation, which generally is more than sufficient for you to get started. You'll likely be able to directly use the app container that you built in this chapter but may find that a control panel replaces your `docker-compose.yml` file.

15. https://docs.docker.com/engine/swarm/secrets/