DOCKER FOR BEGINNERS

JASON SWETT

host of the

Code with Jason Podcast

Docker for Beginners

A gentle introduction to Docker concepts

Jason Swett

Contents

1 Introduction **1**
1.1 What you'll learn . . . 1

2 The two use cases for Docker **3**
2.1 Dockerizing for development . . . 3
2.2 Dockerizing for production . . . 3

3 Docker vs. Docker Compose **5**

4 Docker concepts and terminology **7**
4.1 Host machine . . . 7
4.2 Images and containers . . . 7
4.3 Dockerfile . . . 8
4.4 Volume . . . 8
4.5 Docker Compose . . . 9
4.6 Docker Hub . . . 9

5 A Docker "hello world" app **11**
5.1 What we're going to do . . . 11
5.2 Adding a Dockerfile . . . 12
5.2.1 FROM . . . 12
5.2.2 RUN . . . 13
5.2.3 WORKDIR . . . 13
5.3 Listing our existing images . . . 14
5.4 Listing our existing containers . . . 14

5.5 Building our image 14
5.6 Confirming that our image was created 15
5.7 Creating and shelling into a container 15
5.8 Viewing our new container 15
5.9 Poking around in our container 16
5.10 Running the Lisp REPL 16
5.11 Exiting the container 16
5.12 Deleting the image 17
5.13 Recap 17

6 Dockerizing a Sinatra application 19
6.1 What we're going to do and why 19
6.2 Creating the Sinatra application 19
6.3 Dockerizing the Sinatra application 21
6.3.1 Creating the Dockerfile 21
6.3.2 Building the Docker image 22
6.3.3 Running the Docker image 23

7 Adding Docker Compose to our Sinatra app 25
7.1 What we're going to do and why 25
7.1.1 Docker Compose recap 25
7.1.2 Our objective 26
7.2 Adding Docker Compose 26
7.2.1 Creating the project directory 26
7.2.2 Adding our docker-compose.yml file 26
7.2.3 The docker-compose.yml file, explained 27
7.2.4 Building our environment 28
7.2.5 Running our environment 28

8 Adding a PostgreSQL database to our application 29
8.1 What we're going to do and why 29
8.2 Adding PostgreSQL 29
8.2.1 Creating the project directory 29

8.2.2 Adding PostgreSQL to our Docker Compose configuration 30
8.2.3 The postgresql service, explained 30
8.2.4 Running the PostgreSQL service 32
8.2.5 Getting into the container in a more direct way 33
8.2.6 Adding some data to work with 33
8.2.7 Showing some database data in our Sinatra app 33

9 Conclusion **35**

Chapter 1

Introduction

1.1 What you'll learn

First we're going to cover the two use cases for Docker: Dockerizing for development and Dockerizing for production. This book will focus exclusively on Dockerizing for development, although some of the things you learn will be applicable to Dockerizing for production if you should later choose to learn about that area.

Then we'll see how to create our own Dockerized application. We're going to create a Dockerized application using Sinatra, a very lightweight Ruby web development framework. Then we'll add a Dockerized PostgreSQL database to our Sinatra application and connect our application with our database.

Let's get started.

Chapter 2

The two use cases for Docker

2.1 Dockerizing for development

The main reason you would want to Dockerize an app for development is to make it easier for a new developer to get a development environment set up on their machine.

When you have your app Dockerized for development, Docker can install and run services for you. For example, if your app uses PostgreSQL, Redis, webpack-dev-server and Sidekiq (for example), Docker will run all those processes for you. And before running these services, Docker will install them for you if you don't already have them. (For example, if you don't have PostgreSQL installed, Docker will install it for you.)

This means that each new developer who starts the project doesn't have to go through a potentially long and painful process of installing all the dependencies for your app before getting to work. The developer can just clone the repo, run a single command, and be up and running.

2.2 Dockerizing for production

The benefits of Dockerizing for production overlap with the benefits of Dockerizing for development, but the two use cases aren't exactly the same.

In my development example I mentioned that you might have certain services running locally like PostgreSQL, Redis, webpack-dev-server and Sidekiq. In a production environment, you of course don't have all these processes running on a single machine or VM the way you would for local development. Rather, you might have (if you're using AWS for example), your database hosted on RDS, a Redis instance on ElasticCache, Sidekiq running on a worker EC2 instance, and no webpack-dev-server because that doesn't apply in production. So the need is very different.

So unlike a development use case, a production Docker use case doesn't necessarily include installing and running various services for you, because running all kinds of services on a single host machine is not how a "modern" production infrastructure configuration works.

Chapter 3

Docker vs. Docker Compose

As mentioned in the previous chapter, an app can either be Dockerized for development or for production (or both).

Docker Compose is a tool that's typically useful when Dockerizing for development. The idea is that you tell Docker Compose what services you need to run and Docker Compose will install and run them for you.

Here's a very stripped-down pseudo-example of what a Docker Compose config file might look like.

```
services:
  postgres:
    image: postgres:13.1-alpine

  redis:
    image: redis:4.0.14-alpine

  elasticsearch:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.11.1
```

In this example, we're saying that our services include PostgreSQL, Redis and Elasticsearch. The `image` directive tells Docker Compose which Docker image to use for that service.

We can run `docker compose up` to run all of our specified services. If any of these services don't have the necessary software installed (e.g. our computer doesn't have PostgreSQL installed), Docker Compose will use the Docker image we specified in order to install the necessary software.

Chapter 4

Docker concepts and terminology

4.1 Host machine

The *host machine* is the machine that Docker is running on. When Dockerizing for development, the host machine is typically the laptop that you're working on. In a production environment, the host machine could be (for example) an AWS EC2 instance.

4.2 Images and containers

Images and containers are sufficiently interrelated that it makes sense to explain them together.

A Docker image is like a blueprint. A container is like the house that you build with that blueprint. You can build any number of houses off of the same blueprint. If you want to, you can even build a house today, demolish it tomorrow, change the blueprint, and build a new version of the house the next day.

Let's take that analogy and make it concrete. A Docker image (which again is the blueprint) might specify something like "use the Ubuntu operating system, install PostgreSQL, and get PostreSQL running." Using that image, you can then create a container which, according to the image, will run on Ubuntu, have PostgreSQL installed and have PostgreSQL running.

A container is like a computer inside your computer. If you have a container running, you can do things like shell into it or use a web browser to visit a website hosted on the container, for example.

4.3 Dockerfile

A `Dockerfile` is a specification for building an image.

For example, here's a `Dockerfile` that says "Use Ubuntu as the operating system, and install PHP".

```
FROM ubuntu:20.04

RUN apt update && apt install -y php

WORKDIR /usr/src
```

4.4 Volume

One of the defining characteristics of a Docker container is that it's ephemeral. Containers can be started, stopped, created and deleted, and they often are. This creates a problem if you want to save any files or make permanent changes to existing files. Once your container gets deleted, anything you've done on the container's filesystem is gone.

The solution to this problem is volumes. The idea is that instead of storing things on the *container's* filesystem, you store things somewhere on the *host machine's* filesystem.

To use volumes, you can tell Docker that a certain service should store its files in a certain location on the host machine. For example, you can say, "Hey Docker, whenever you need to store PostgreSQL data, store it in `/var/lib/postgresql/d` on the host machine".

This way your data can survive containers being stopped or deleted.

4.5 Docker Compose

Docker Compose is a tool for composing an environment from one or more services.

For example, if my development environment needs PostgreSQL, Redis and Elasticsearch, I could tell Docker via a Docker Compose config file that it should install all three of these services and get them running for me. I can also tell Docker that my environment includes e.g. a custom Rails application.

In addition to specifying that all these services exist, I can use a Docker Compose file to specify how these services are supposed to be configured and how they are to talk to one another.

4.6 Docker Hub

Docker Hub is a place where Docker images are stored. You can think of it analogously to open source libraries being hosted on GitHub.

If I want to use Redis, for example, I can tell Docker to grab me the `redis:4.0.14-alp` image and the default place that Docker will look for this image is on Docker Hub.

Chapter 5

A Docker "hello world" app

5.1 What we're going to do

In this tutorial we're going to illustrate what I consider to be Docker's central and most magical ability: to let you run software on your computer that you don't actually have installed on your computer.

The specific software we're going to run is the Lisp REPL (read-evaluate-print loop). The reason I chose Lisp is because you're unlikely to happen to have Lisp already installed. If I had chosen to use a language that you might already have installed on your computer, like Ruby or Python, the illustration would lose much of its sizzle.

Here are the steps we're going to carry out. Don't worry if you don't understand each step right now because we're going to be looking at each step in detail as we go.

1. Add a `Dockerfile`
2. Build an image from our `Dockerfile`
3. Run our image, which will create a container
4. Shell into that container
5. Start a Lisp REPL inside the container

6. Run a Lisp "hello world" inside the REPL
7. Exit the container
8. Delete the image

5.2 Adding a Dockerfile

A **`Dockerfile`** is a specification for building a Docker image. We're going to write a **`Dockerfile`**, use the **`Dockerfile`** to build a custom image, create a container using the image, and finally shell into the container and start the Lisp REPL.

First I'm going to show you the **`Dockerfile`** and then I'll explain each individual part of it.

```
# Dockerfile

FROM ubuntu:20.04

RUN apt update && apt install -y sbcl

WORKDIR /usr/src
```

5.2.1 FROM

```
FROM ubuntu:20.04
```

The **`FROM`** directive tells Docker what image we want to use as our *base image*. A base image, as the name implies, is the image that we want to use as a starting point for our custom image. In this case our base image is just providing us with an operating system to work with.

The **`FROM`** directive takes the form of **`<image>:<tag>`**. In this case our image is **`ubuntu`** and our tag is **`20.04`**. When Docker sees **`ubuntu:20.04`**, it will look on Docker Hub for an image called **`ubuntu`** that's tagged with **`20.04`**.

5.2.2 RUN

```
RUN apt update && apt install -y sbcl
```

The `RUN` command in a `Dockerfile` simply takes whatever it's given and runs it on the command line.

In this case the command we're running is `apt update && apt install -y sbcl`. The `&&` in between the two commands means "execute the first command, then execute the second command if and only if the first command was successful". Let's deal with each of these commands individually.

The `apt update` command is a command that downloads package information. If we were to skip the `apt update` command, we would get an error that says `Unable to locate package sbcl` when we try to install `sbcl`. So in other words, running `apt update` makes our package manager aware of what packages are available to be installed.

The `apt install -y sbcl` command installs a package called `sbcl`. SBCL stands for Steel Bank Common Lisp which is a Common Lisp compiler. (Common Lisp itself is a popular dialect of the Lisp language.)

The `-y` part of `apt install -y sbcl` means "don't give me a yes/no prompt". If we were to leave off the `-y` we'd get an "are you sure?" prompt which would be no good because the `Dockerfile` isn't executed in an interactive way that would actually allow us to respond to the prompt.

5.2.3 WORKDIR

```
WORKDIR /usr/src
```

The `WORKDIR /usr/src` directive specifies which directory to use as the working directory inside the container. Imagine being logged into a Linux machine and running `cd /usr/src`. After running that command, you're "in" `/usr/src` and `/usr/src` is your working directory. Similar idea here.

5.3 Listing our existing images

Before we use our **`Dockerfile`** to build our image, let's list our existing Docker images. If this is your first time doing anything with Docker then the list of existing images will of course be empty.

In any case, let's run the **`docker image ls`** command:

```
$ docker image ls
```

5.4 Listing our existing containers

In addition to the **`docker image ls`** command which lets us list any images we have, there's an analogous command that lets us list our containers.

```
$ docker container ls
```

5.5 Building our image

We can use the **`docker build`** command to build our image. The **`--tag lisp`** part says "give the resulting image a tag of **`lisp`**". The **`.`** part says "when you look for the **`Dockerfile`** to build the image with, look in the current directory".

```
$ docker build --tag lisp .
```

After you run this command you'll see some entertaining output fly across the screen while Docker is building your image for you.

5.6 Confirming that our image was created

Assuming that the build was successful, we can now use the `docker image ls` command once again to list all of our existing images, which should now include the image we just built.

```
$ docker image ls
```

You should see something like the following:

```
REPOSITORY    TAG       IMAGE ID       CREATED          SIZE
lisp          latest    91f4fa2a754a   11 minutes ago   140MB
```

5.7 Creating and shelling into a container

Run the following command, which will place you on the command line inside a container based on your image. You should of course replace `<image id>` with the image id that you see when you run `docker image ls`.

```
$ docker run --interactive --tty <image id> /bin/bash
```

The `docker run` command is what creates a container from the image. The `/bin/bash` argument says “the thing you should run on this container is the `/bin/bash` program”.

5.8 Viewing our new container

Now that we’ve invoked the `docker run` command, we have a new container. Open a separate terminal window/tab and run `docker container ls`.

```
$ docker container ls
```

You should see a new container there with an image ID that matches the ID of the image you saw when you ran `docker image ls`.

```
CONTAINER ID    IMAGE           COMMAND         CREATED           STATUS
5cee4af0cfa9    91f4fa2a754a    "/bin/bash"     4 seconds ago     Up 3 seconds
```

5.9 Poking around in our container

Just for fun, run a few commands in the container and see what's what.

```
$ pwd    # show the current directory
$ ls -la # show the contents of the current directory
$ whoami # show the current user
```

5.10 Running the Lisp REPL

Finally, let's run the Lisp REPL by running the `sbcl` command inside our container.

```
$ sbcl
```

Once you're inside the REPL, run this piece of “hello, world!” Lisp code.

```
(format t "hello, world!")
```

5.11 Exiting the container

Press CTRL+D to exit the Lisp REPL and then CTRL+D again to exit the container. The container will delete itself once exited.

5.12 Deleting the image

Run the following command to delete the image. The `rmi` part stands for "remove image" and the `-f` flag stands for "force".

```
$ docker rmi -f <image id>
```

5.13 Recap

You've completed this exercise. You're now capable of the following things:

- Writing a `Dockerfile` that specifies an operating system to use and some software to install
- Listing Docker images and containers
- Building Docker images
- Shelling into a container
- Using the software that was installed on the container

If you possess these capabilities and have at least a little bit of a grasp of the underlying concepts, then you're well on your way to being able to use Docker to accomplish real work.

Chapter 6

Dockerizing a Sinatra application

6.1 What we're going to do and why

If our Lisp application was a "Docker hello world", then this application will be a "Docker web app hello world". We'll see not just how to containerize an application but how to containerize a web application that we can actually visit in the browser.

6.2 Creating the Sinatra application

First of all, let's create a directory in which to keep our Sinatra application.

```
$ mkdir sinatra_app
$ cd sinatra_app
```

Our Sinatra "application" will have just one file. The application will have just one endpoint. Create a file called `hello.rb` with the following content.

```
# hello.rb

require 'sinatra'

get '/' do
  'It works!'
end
```

We'll also need to create a `Gemfile` that says Sinatra is a dependency.

```
# Gemfile

source 'https://rubygems.org'

gem 'sinatra'
```

Lastly for the Sinatra application, we'll need to add the rackup file, `config.ru`.

```
# config.ru

require './hello'

run Sinatra::Application
```

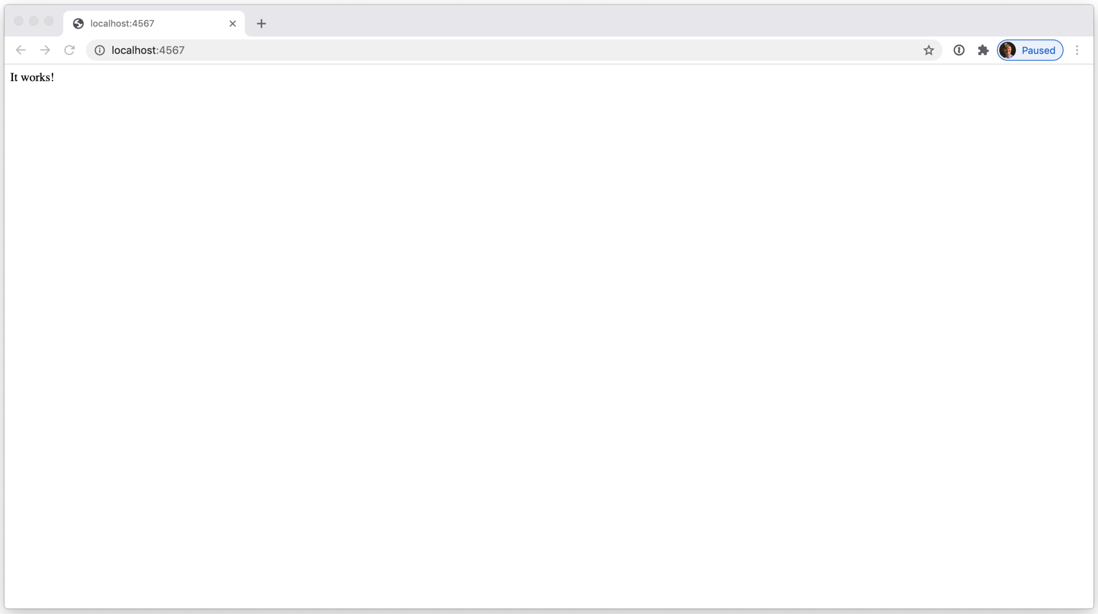
After we run `bundle install` to install the Sinatra gem, we can run the Sinatra application by running `ruby hello.rb`.

```
$ bundle install
$ ruby hello.rb
```

Sinatra apps run on port `4567` by default, so let's open up `http://localhost:4567` in a browser.

```
$ open http://localhost:4567
```

If everything works properly, you should see the following.



It works!

6.3 Dockerizing the Sinatra application

Dockerizing the Sinatra application will involve two steps. First, we'll create a `Dockerfile` will tells Docker how to package up the application. Next we'll use our `Dockerfile` to build a Docker *image* of our Sinatra application.

6.3.1 Creating the Dockerfile

Here's what our `Dockerfile` looks like. You can put this file right at the root of the project alongside the Sinatra application files.

```
# Dockerfile

FROM ruby:2.7.4

WORKDIR /code
COPY . /code
RUN bundle install

EXPOSE 4567
```

```
CMD ["bundle", "exec", "rackup", "--host", "0.0.0.0", "-p", "4567"]
```

Since it might not be clear what each part of this file does, here's an annotated version.

```
# Dockerfile

# Include the Ruby base image (https://hub.docker.com/_/ruby)
# in the image for this application, version 2.7.4.
FROM ruby:2.7.4

# Put all this application's files in a directory called /code.
# This directory name is arbitrary and could be anything.
WORKDIR /code
COPY . /code

# Run this command. RUN can be used to run anything. In our
# case we're using it to install our dependencies.
RUN bundle install

# Tell Docker to listen on port 4567.
EXPOSE 4567

# Tell Docker that when we run "docker run", we want it to
# run the following command:
# $ bundle exec rackup --host 0.0.0.0 -p 4567.
CMD ["bundle", "exec", "rackup", "--host", "0.0.0.0", "-p", "4567"]
```

6.3.2 Building the Docker image

All we need to do to build the Docker image is to run the following command. I'm choosing to *tag* this image as **`hello`**, although that's an arbitrary choice that doesn't connect with anything inside our Sinatra application. We could have tagged it with anything. The **`.`** part of the command tells **`docker build`** that we're targeting the current directory. In order to work, this command needs to be run at the project root.

```
$ docker build --tag hello .
```

Once the `docker build` command successfully completes, you should be able to run `docker image ls` and see the `hello` image listed.

6.3.3 Running the Docker image

To run the Docker image, we'll run `docker run`. The `-p 4567:4567` portion says "take whatever's on port 4567 on the container and expose it on port 4567 on the host machine".

```
$ docker run -p 4567:4567 hello
```

If we again visit `http://localhost:4567`, we should see the Sinatra application being served.

```
$ open http://localhost:4567
```

Congratulations. You now have a working Dockerized Sinatra application.

Chapter 7

Adding Docker Compose to our Sinatra app

7.1 What we're going to do and why

7.1.1 Docker Compose recap

You'll recall that Docker Compose is a tool that lets you *compose* an environment out of multiple services.

As a reminder from Chapter 3 (Docker vs. Docker Compose), you could tell Docker Compose that your environment has three dependencies: PostgreSQL, Redis and Elasticsearch. Docker Compose will then do the work of downloading and running those three services for you.

In addition to off-the-shelf services like PostgreSQL etc., it's common for a Docker Compose configuration to include one "custom" service which is the application itself. So in this scenario you would say "Hey Docker Compose, I want PostgreSQL, Redis, and Elasticsearch, and I have my own custom web app too." The custom app is at sibling level to the other services, although the custom app is built in a different way, as we'll see.

7.1.2 Our objective

Our ultimate objective with this Sinatra app is to get a PostgreSQL database connected to the app using Docker Compose. This will involve two steps:

1. Applying Docker Compose to our application
2. Adding PostgreSQL to our Docker Compose configuration

Each of these steps is non-trivial, so we're going to take the steps one at a time. This chapter deals only with step 1. The next chapter will deal with step 2.

7.2 Adding Docker Compose

7.2.1 Creating the project directory

First, let's create a new directory to house this new version of our Sinatra app. We can copy the existing `sinatra_app` directory and use it as a starting point.

```
$ cp -R sinatra_app sinatra_app_with_docker_compose
```

7.2.2 Adding our docker-compose.yml file

Paste the following into a new file called `docker-compose.yml`.

```
# docker-compose.yml
---
version: '3.8'

services:
  web:
    build: .
    volumes:
      - .:/usr/src/app:cached
    ports:
      - "127.0.0.1:4567:4567"
```

7.2.3 The docker-compose.yml file, explained

Here's an annotated version of our `docker-compose.yml`.

```
# docker-compose.yml
---
# This line specifies the version of Docker Compose that we're using.
version: '3.8'

# Under the "services" directive we'll list all the services that
# our environment needs. Right now we're only specifying one service,
# called "web".
services:

  # The name we're giving this service, "web", is totally arbitrary.
  # We could have called it "zaphodbeeblebrox" if we wanted to.
  web:

    # The "build" directive tells Docker Compose where to look for
    # a Dockerfile. In this case we just want Docker Compose to
    # look for our Dockerfile in the current directory.
    build: .

    # As you'll recall from Chapter 4, Docker volumes are a way for
    # files to be shared between container filesystems and host machine
    # filesystems.
    #
    # When we say .:/usr/src/app, we're saying "take the current
    # directory on the host machine and sync it with the /usr/src/app
    # directory on the container".
    #
    # The "cached" part specifies the syncing strategy to use. There
    # are three options:
    # - cached: files change on host machine, container is read-only
    # - delegated: files change on container, host machine is read-only
    # - default: host machine and container sync with each other
    volumes:
      - .:/usr/src/app:cached

    # The "ports" directive tells Docker Compose what ports to expose
    # for this service. Here we're saying "take whatever this service
    # puts on port 4567 on the container, and forward it to port 4567
    # on the host machine".
    ports:
      - "127.0.0.1:4567:4567"
```

7.2.4 Building our environment

Before we build our environment, let's run `docker image ls` to see all our images. Then, after we build the environment, we'll run `docker image ls` again to see what's new.

```
$ docker image ls
```

Now run `docker compose build` to build the environment. When we run this command, Docker Compose will look at our `docker-compose.yml` and do what's necessary to get images in place for any services we've specified. Since we've only specified one service in this case (`web`), all that will happen is that Docker Compose will use our `Dockerfile` to build the image for the `web` service.

```
$ docker compose build
```

Now that the image has been built, let's run `docker image ls` again and see what we see.

```
$ docker image ls
```

There should be a new image present called `sinatra_app_with_docker_compose_w`

7.2.5 Running our environment

Now that we've built our environment, let's run it. We can do this by running `docker compose up`.

```
$ docker compose up
```

If you visit `localhost:4567`, you should see the "It works!" page.

Now let's add PostgreSQL to our Docker Compose environment.

Chapter 8

Adding a PostgreSQL database to our application

8.1 What we’re going to do and why

As explained in the previous chapter, our ultimate objective is to connect our Sinatra application to a PostgreSQL database. In the previous chapter we added Docker Compose to our Sinatra application. In this chapter we’re going to add a PostgreSQL service to our Docker Compose configuration.

8.2 Adding PostgreSQL

8.2.1 Creating the project directory

Let’s use our Docker-Compose-enabled Sinatra app as a starting point.

```
$ cp -R sinatra_app_with_docker_compose sinatra_app_with_postgresql
```

8.2.2 Adding PostgreSQL to our Docker Compose configuration

Let's add a service called **`postgres`** to our services.

```yaml
# docker-compose.yml
---
version: '3.8'

services:
  web:
    build: .
    volumes:
      - .:/usr/src/app:cached
    ports:
      - "127.0.0.1:4567:4567"

  postgres:
    image: postgres:13.1-alpine
    volumes:
      - postgresql:/var/lib/postgresql/data:delegated
    ports:
      - "127.0.0.1:5444:5432"
    environment:
      POSTGRES_USER: my_app
      POSTGRES_HOST_AUTH_METHOD: trust

volumes:
  postgresql:
```

8.2.3 The postgresql service, explained

Here's an annotated version of our new **`docker-compose.yml`**. No annotations are needed for the **`web`** service because we of course covered that in the previous chapter.

```yaml
---
version: '3.8'

services:
  web:
    build: .
    volumes:
      - .:/usr/src/app:cached
    ports:
      - "127.0.0.1:4567:4567"

  # We're assigning this service the name of "postgres", although just
  # like "web" this name is arbitrary and could have been anything.
  postgres:

    # Unlike our "web" service for which we specified a build directory,
    # here we're specifying an image to use.
    #
    # Docker will see this image we specified and go looking on Docker
    # Hub for an image called "postgres" with a tag of "13.1-alpine".
    # 13.1 is the PostgreSQL version and "alpine" means Alpine Linux,
    # a lightweight Linux distribution often used in Docker setups.
    #
    # Rather than building a custom image like our "web" service did,
    # Docker Compose will just download the postgres image from Docker
    # Hub.
    image: postgres:13.1-alpine

    # We need to specify a volume for our PostgreSQL service or else
    # we'd lose all our database data each time we restarted our
    # PostgreSQL container.
    #
    # Recall that "delegated" means that the files change on the
    # container and the host machine is read-only. If you think about
    # it, this syncing strategy makes sense for a database.
    volumes:
      - postgresql:/var/lib/postgresql/data:delegated

    # For our web service, we said "take what's running on port 4567
    # on the container and forward it to port 4567 on the host machine".
    # No reason not to use the same port in that case. In this case,
    # however, there's a good chance that the user of this environment
    # might already have PostgreSQL running on their machine, and so
    # might have a conflict on port 5432. Therefore we aren't
    # forwarding port 5432 to port 5432 but rather to port 5444.
    ports:
      - "127.0.0.1:5444:5432"

    # This is where we specify certain environment variables for our
    # environment. We have to specify something for the PostgreSQL
    # user, so we're just calling it "my_app". This is another
    # arbitrary choice that could have been anything. We're also
    # specifying a POSTGRES_HOST_AUTH_METHOD of "trust", meaning that
    # we don't have to specify a password when connecting to the
    # database.
    environment:
      POSTGRES_USER: my_app
      POSTGRES_HOST_AUTH_METHOD: trust
```

```
volumes:
  postgresql:
```

8.2.4 Running the PostgreSQL service

If you run `docker compose up`, Docker Compose will download a PostgreSQL image and get a PostgreSQL service running.

```
$ docker compose up
```

Now let's do the following:

1. Shell into the container that's running PostgreSQL
2. Log into the PostgreSQL command-line interface (CLI) using `psql`

First let's run `docker container ls` to get the id of the container that's running PostgreSQL.

```
$ docker container ls
```

You should see a container called `sinatra_app_with_postgresql_postgres_1`. Copy the id for that container.

The following command will allow us to shell into the container. It's kind of like using SSH to shell into a remote server. The command says "get into the container `<container id>` and run the program `/bin/bash`".

```
$ docker exec -it <container id> /bin/bash
```

Now we're in the container that's running Postgresql. We can get into the PostgreSQL CLI by running `psql -U my_app`.

```
$ psql -U my_app
```

8.2.5 Getting into the container in a more direct way

Rather than running `/bin/bash` and then invoking `psql` ourselves, we can cut out the middleman and just invoke `psql` directly from the beginning. Exit the container and then run this command from your host machine:

```
$ docker exec -it <container id> psql -U my_app
```

This will put us straight into the PostgreSQL CLI.

8.2.6 Adding some data to work with

Databases of course aren't much fun unless we have some data to work with. Let's run the following commands to create a table called `cities` and then insert a few rows into it.

```
create table cities (name varchar(100));
insert into cities (name) values ('Denver'), ('Salt Lake City'), ('Boise');
```

Let's verify our work by querying the `cities` table.

```
select * from cities;
```

8.2.7 Showing some database data in our Sinatra app

This is the step where it all comes together. Let's modify our `hello.rb` file so that it connects to and queries our database.

```
# hello.rb

require 'sinatra'
require 'pg'

get '/' do
  connection = PG.connect(
    dbname: 'my_app',
    user: 'my_app',
    host: 'postgres',
    port: '5432'
  )

  connection.exec_params('SELECT * FROM cities').to_a.to_s
end
```

Annoyingly, we do have to restart our environment by doing CTRL+C and then `docker compose up` again in order for our change to take effect. This is not because of Docker but because of Sinatra. Sinatra requires a server restart in order for code changes to take effect.

If we visit `http://localhost:4567`, we should now see the names of our cities displayed in the browser.

```
$ open localhost:4567
```

Chapter 9
Conclusion

Now you know the use cases for Docker. You've also Dockerized a simple app's development environment.

If you'd like to learn more about what you can do with Docker, you can visit https://www.codewithjason.com/articles/ where I have articles on Docker as well as other topics.

If you have any feedback on this book or suggestions on what I should write about next, you can contact me at jason@codewithjason.com.