Extracted from:

Docker for Rails Developers

Build, Ship, and Run Your Applications Everywhere

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Docker for Rails Developers

Build, Ship, and Run Your Applications Everywhere

Rob Isenberg

edited by Adaobi Obi Tulton

Docker for Rails Developers

Build, Ship, and Run Your Applications Everywhere

Rob Isenberg

The Pragmatic Bookshelf

Raleigh, North Carolina

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Susan Conant
Development Editor: Adaobi Obi Tulton
Copy Editor: Nicole Abramowitz
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-273-2
Book version: P1.0—February 2019

Ruth. In hindsight, writing a book whilst having a baby and renovating a house probably wasn't the best idea—who knew? Thank you for your patience, love, and support. None of this would have been possible without you.

Sammy. I couldn't have imagined the joy and love you'd bring into our life. Be kind, be brave, and be willing to take risks in pursuit of your happiness and passions. I love you so much.

Mum and Dad. Thank you for everything.

Defining Our First Custom Image

In real life, a factory doesn't come out of nowhere. It has to be constructed from blueprints: detailed plans and instructions that describe exactly what it's supposed to look like. Docker's container factories—in other words, images—are no different. They require a special blueprint file aptly named a Dockerfile. A Dockerfile uses special syntax to describe exactly how the image should be constructed. If you've heard the expression *infrastructure as code*, this is an example of it: a Dockerfile describes how a machine image is configured as shown in the figure on page 8.

A Dockerfile is made up of various *instructions*—such as FROM, RUN, COPY, and WORKDIR—each capitalized by convention. Rather than talk about them in the abstract, though, let's look at a specific example.

Here's a basic Dockerfile for running our Rails app. It's not perfect—we'll make several improvements in Chapter 3, *Fine-Tuning Our Rails Image*, on page ?—but it's good enough for now. There's no need to create this file; for now we'll just discuss it:

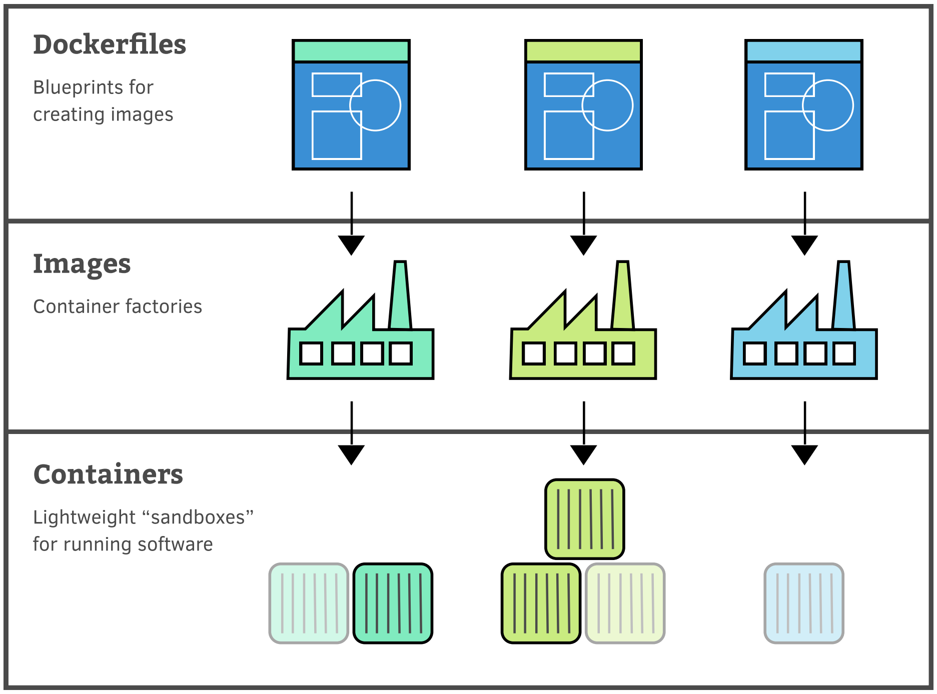
```
FROM ruby:2.6

RUN apt-get update -yqq
RUN apt-get install -yqq --no-install-recommends nodejs

COPY . /usr/src/app/

WORKDIR /usr/src/app
RUN bundle install
```

Every image has to start from something: another, preexisting image. For that reason, every Dockerfile begins with a FROM instruction, which specifies the image to use as its starting point. Typically, we'll look for a starting image that's close to what we need but more general. That way we can extend and customize it to our needs.

The first line of our Dockerfile is

```
FROM ruby:2.6
```

This is saying that our image will be based off the ruby:2.6 image, which, as you've probably guessed, has Ruby 2.6 preinstalled. We've chosen to start from this image because having Ruby installed is our biggest requirement, so this image gets us most of the way there.


Dockerfiles
Blueprints for creating images
Images
Container factories
Containers
Lightweight “sandboxes” for running software

Right at the Top: Base Images

There are several special images with no parent image—known as a *base image*—that ultimately all images depend on. They contain the minimal user filesystem for an operating system.

If you want to build your own stripped-down image, you could build your image `FROM scratch` (reads well, doesn't it?) where `scratch` is a minimal base image.

It's even possible to create your own base images,[1] although this is an advanced topic. We're not going to cover it since chances are you'll never need to do it.

The next two lines of our `Dockerfile` are `RUN` instructions, which tell Docker to execute a command:

```
RUN apt-get update -yqq
RUN apt-get install -yqq --no-install-recommends nodejs
```

1. docs.docker.com/engine/userguide/eng-image/baseimages/

Here, we tell Docker to run apt-get update -yqq, followed by apt-get install -yqq --no-install-recommends nodejs—but what do these two commands achieve for us?

As you may already know, apt-get is a command used to install software on Debian (and some other) Linux distributions.[2] We're using it in our Dockerfile because the official Ruby image that our image builds on top of is based on Debian—specifically, a version called Stretch.[3]

The apt-get update command tells the package manager to download the latest package information. Many Dockerfiles will have a similar line, because without it, apt has no package information at all, and therefore won't be able to install anything. The -yqq option is a combination of the -y option, which says to answer "yes" to any prompts, and the -qq option, which enables "quiet" mode to reduce the printed output.

Next, the apt-get install command installs Node.js, a prerequisite for running Rails. The --no-install-recommends says not to install other recommended but nonessential packages—we don't need them, and we want to keep our image size as small as possible by not installing unnecessary files.

If you're familiar with apt-get in Linux, you may be wondering why we're not running the commands as root with sudo. That's because, by default, commands inside a container are run by the root user, so sudo is unnecessary (although, as we mention on page ?, this has security implications for production apps).

Let's shift gears briefly before we look at the next line of our Dockerfile.

Remember that images, and the containers they spawn, are separate from our local machine—they are isolated, sandboxed environments. Therefore, we need a way to include some of our local files inside the containers we run.

We've already seen in *Generating a New Rails App Without Ruby Installed*, on page ?, that we can mount a local directory into a running container. A mounted volume acts like a shared directory between the container and the host, and is one way we can make local files accessible inside the container.

However, mounting a volume has a serious downside if it's the only way you get files into a container. Files in a volume aren't part of the image itself; they are overlaid onto the image at *runtime* (when you start a container.) If the mounted files were essential, the image wouldn't function without them, but

2. https://en.wikipedia.org/wiki/Advanced_Packaging_Tool
3. https://github.com/docker-library/ruby/blob/a04dd5259eaef8d682dae2bb709f03219a6e5905/2.5/stretch/Dockerfile#L1

the whole point of images is to package up everything they need in order to run. Therefore, it's good practice to bake any needed files into the image itself.

The next line in our Dockerfile serves exactly this purpose:

```
COPY . /usr/src/app/
```

This tells Docker to copy all the files from our local, current directory (.) into /usr/src/app on the filesystem of the new image. Since our local, current directory is our Rails root, effectively we're saying, "Copy our Rails app into the container at /usr/src/app." The source path on our local machine is always *relative* to where the Dockerfile is located.

Having added our Rails files into the image at /usr/src/app, we're going to want to run various commands that need to operate in this directory where the files are. For example, soon we'll want to run our app with a Rails server in a container with a command like this:

```
$ docker run [OPTIONS] <our custom image> bin/rails server
```

Unfortunately, this command would fail because, by default, a container's working directory is /, which doesn't contain our Rails app files—we copied those into /usr/src/app.

However, the WORKDIR instruction can help us fix the situation. Effectively, it performs a *change directory* (cd) command, changing what the image considers its current directory. The next line in our Dockerfile uses it to set /usr/src/app as the working directory:

```
WORKDIR /usr/src/app
```

Now running bin/rails server (and similar) commands will work because they will be executed from the correct directory.

You can use multiple WORKDIR instructions in your Dockerfile, each one remaining in effect until another one is issued. The final WORKDIR will be the initial working directory for containers created from the image.

Finally, we come to the last line of our Dockerfile:

```
RUN bundle install
```

The command is executed from the container's current working directory, which in the previous command was set to be /usr/src/app. So this will install the gems defined in our Rails project's Gemfile, which are needed in order to start the application.

Putting It All Together

Armed with all this knowledge, our Dockerfile should now be much more understandable. Let’s review it one more time:

```
Line 1  FROM ruby:2.6
     2
     3  RUN apt-get update -yqq
     4  RUN apt-get install -yqq --no-install-recommends nodejs
     5
     6  COPY . /usr/src/app/
     7
     8  WORKDIR /usr/src/app
     9  RUN bundle install
```

First, on line 1, we say that our custom image will use the ruby:2.6 image as its starting point. Next we update the apt package manager’s package information (line 3), so it knows where to install things from. Then we use it to install nodejs (line 4), which we need for Rails’ asset pipeline.

With the prerequisites for Rails taken care of, we then copy our Rails app files from our local directory into the container at /usr/src/app (line 6) so they are baked into the image. We make this the current working directory for the image (line 8) so that we can execute Rails commands against the image from the correct directory.

Finally, we bundle install (line 9) to install the gems we need for our Rails project.

Now that it makes more sense, let’s go ahead and actually create this Dockerfile. First let’s make sure we’re in the top-level (root) directory of our Rails app:

```
$ ls
Gemfile    Rakefile  bin     config.ru  lib  package.json  storage  vendor
README.md  app       config  db         log  public        tmp
```

Then crack open your editor of choice and create a file called Dockerfile with the contents as shown. I’d highly recommend typing it in by hand rather than copying and pasting—when learning a new skill, physically typing things out helps to cement it in your mind and build up your muscle memory.

With our swanky Dockerfile in hand, let’s turn our attention to how we use it to create an actual image.