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.

How Containers Can Talk to Each Other

If two containers are isolated, independent processes, how come, as we just saw, that they are able to talk to one another? While it's true that the *code and processes* running in a container are sandboxed, that does not mean the container has no way to communicate with the outside world. If containers could not communicate, we would not be able to connect them together to create a powerful, connected system of services that together make up our application.

If you remember back to *Launching Our App,* on page ?, we said that docker-compose up creates a new network for the app. By default, all containers for our app are connected to the app's network and can communicate with each other. This means that our containers, just like a physical or virtual server, can communicate outside themselves using TCP/IP networking.

Let's list our currently defined networks using the command:

```
$ docker network ls
```

You should see some output similar to the following:

```
NETWORK ID          NAME                DRIVER       SCOPE
128925dfad81        bridge              bridge       local
5bd7167263e8        host                host         local
e2af02026928        myapp_default       bridge       local
d1145155d62a        none                null         local
```

The first network called bridge is a legacy network to provide backwards compatibility with some older Docker features—we won't be using it now that we've switched to Compose. Similarly, the host and none networks are special networks that Docker sets up that we don't need to care about.

The network we do care about is called myapp_default—this is our app's dedicated network that Compose created for us (Compose uses the <appname>_default naming convention). The reason Compose creates this network for us is simple: it knows that the services we're defining are all related to the same application, so inevitably they are going to need to talk to one another.

But how do containers on this network find each other?

All Docker networks (except for the legacy bridge network) have built-in Domain Name System (DNS) name resolution. That means that we can communicate with other containers running on the same network by name. Compose uses the service name (as defined in our docker-compose.yml) as the DNS entry. So if we wanted to reach our web service, that's accessible via the hostname web.

This provides a basic form of *service discovery*—a consistent way of finding container-based services, even across container restarts.

This explains how we were able to connect from the ad-hoc container running the redis-cli to our Redis server running as the redis service. Here's the command we used:

```
$ docker-compose run --rm redis redis-cli -h redis
```

The option -h redis says, "Connect to the host named redis." This only worked because Compose had already created our app's network and set up DNS entries for each service. In particular, our redis service can be referred to by the hostname redis.

Our Rails App Talking to Redis

Although it's great that we've started up a Redis server using Compose, it's not much use to us by itself. The whole point of running the Redis server is so our Rails app can talk to it and use it as a key-value store. So let's connect our Rails app to Redis and actually use it for something. Sound like fun?

Now, there are a million ways an app might want to use Redis. For our purposes, though, we don't really care *what* we use Redis for; we care more about *how* to use it. We're going to use an intentionally basic example: our Rails app will simply store and retrieve a value. However, keep the larger point in mind—once you know how to set up the Rails app to talk to the Redis server in a container, you can use it however you like.

Ready? Let's begin.

Installing the Redis Gem

The first thing we need to do to get our Rails app talking to Redis is to install the redis gem. You may remember that to update our gems, we need to update our image as we saw on page ?.

So first, in our Gemfile, uncomment the Redis gem in the Gemfile like so:

```
gem 'redis', '~> 4.0'
```

Next, stop our Rails server:

```
$ docker-compose stop web
```

and rebuild our custom Rails image:

```
$ docker-compose build web
```

Among other things, this runs bundle install, which installs the Redis gem:

```
Building web
Step 1/8 : FROM ruby:2.6
«...»
Step 6/8 : RUN bundle install
«...»
Installing redis 4.1.0
«...»
Bundle complete! 16 Gemfile dependencies, 69 gems now installed.
Bundled gems are installed into `/usr/local/bundle`
«...»
Removing intermediate container 3831c10d2cb5
 ---> 1ca01125bd35
Step 7/8 : COPY . /usr/src/app/
 ---> 852dc1f2b419
Step 8/8 : CMD ["bin/rails", "s", "-b", "0.0.0.0"]
 ---> Running in 280c7e2eb556
Removing intermediate container 280c7e2eb556
 ---> d9b3e5325308
Successfully built d9b3e5325308
Successfully tagged myapp_web:latest
```

It's good to get into the habit of rebuilding our image to perform bundle install for us, having updated our Gemfile. That said, we'll learn about a more advanced approach to gem management on page ? that, as well as being much faster, allows us to stick with our familiar bundle install workflow.

Let's start up our newly built Rails server again:

```
$ docker-compose up -d web
```

Updating Our Rails App to Use Redis

Next, we're going to actually use Redis from our Rails app. As we said before, we just want a basic demonstration that we can connect to the Redis server and store and retrieve values. So let's start by generating a welcome controller in our Rails app with a single index action:

Linux Users: File Ownership

Make sure you have chowned the files by running:

```
$ sudo chown <your_user>:<your_group> -R .
```

See *File Ownership and Permissions*, on page ?, for more details.

```
$ docker-compose exec web bin/rails g controller welcome index
      create  app/controllers/welcome_controller.rb
       route  get 'welcome/index'
      invoke  erb
      create    app/views/welcome
      create    app/views/welcome/index.html.erb
```

```
invoke  helper
create    app/helpers/welcome_helper.rb
invoke  assets
invoke    coffee
create      app/assets/javascripts/welcome.coffee
invoke    scss
create      app/assets/stylesheets/welcome.scss
```

Let's modify our welcome#index action (in app/controllers/welcome_controller.rb) to be as follows:

```
Line 1  class WelcomeController < ApplicationController
     2    def index
     3      redis = Redis.new(host: "redis", port: 6379)
     4      redis.incr "page hits"
     5  
     6      @page_hits = redis.get "page hits"
     7    end
     8  end
```

In our index action, on line 3, we use the Redis client gem to connect to the Redis server by name and by the port number we expect it to be running on. Then, on line 4, we increment a Redis key-value pair, called "page hits." If you're wondering what happens the very first time this code is run, don't fret: if the key is not found, Redis will initialize it to zero, so our code will work as expected. Finally, on line 6, we fetch the current number of page hits from Redis, storing it in an instance variable, ready to display it in our view.

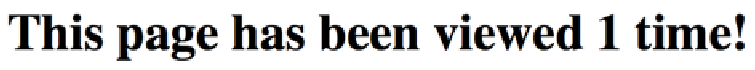
Now let's edit our view file (app/views/welcome/index.html.erb) to display the number of page hits:

```
<h1>This page has been viewed <%= pluralize(@page_hits, 'time') %>!</h1>
```

Finally, in config/routes.rb, let's change the autogenerated route so we can access our new WelcomeController's index action from /welcome (rather than /welcome/index):

```
Rails.application.routes.draw do
  get 'welcome', to: 'welcome#index'
end
```

Now let's visit our Rails app in the browser at http://localhost:3000/welcome. You should see a page with our rendered welcome index.html.erb file, as shown in the following figure:



This page has been viewed 1 time!

The page loads without errors—a good sign. Now try reloading the page. Every time you do, you should see the number of page hits increasing.

What does this mean? It means that our Rails app connected to the Redis server, incremented the value of "page hits" from default of 0 to 1, and finally displayed our welcome message with the number of page hits. More generally, we successfully got two containers to talk to each other. This is possible thanks to Compose creating the network for the app and automatically attaching containers to it.