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

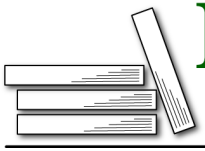
Docker for Rails Developers

Build, Ship, and Run Your Applications Everywhere

Rob Isenberg

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

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.

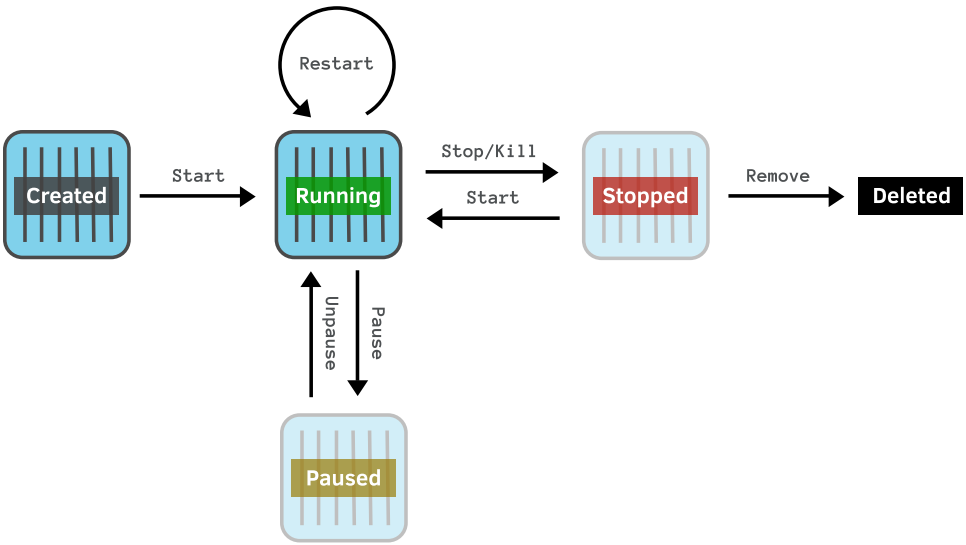
Sammy. I couldn't have imagined the joy and love you'd bring into our life. Be kind, be brave, and be willing to take risks in pursuit of your happiness and passions. I love you so much.

Mum and Dad. Thank you for everything.

Starting and Stopping Services

A common thing we'll need to do while developing our application is to stop or start the various services that make it up. In a moment, we'll dive into the fine-grained control Compose gives us to do this. Before we do, though, it's helpful to have in mind the journey that containers go through, from creation until they are no longer needed.

The following figure shows a simplified version of a container's life cycle:



A container comes into existence in the *created* state. It doesn't execute any code; it merely sits around waiting until it's called for. When the container is started, it moves into the *running* state, where it actively executes code. The `docker run` command we've seen creates a new container, then starts it running.

In the running state, a container can be restarted, stopped, killed, or paused. Pausing a container suspends the running processes so that they can be resumed some time later. Stopping a container attempts to shut down gracefully by sending a terminate signal (`SIGTERM`) to the main process inside the container—falling back to a forceful shutdown with a kill signal (`SIGKILL`) if this fails. Killing a container jumps straight to the forceful termination.

A container moves into the *stopped* state if it is stopped, or killed, or if the main process running inside it terminates. The stopped state is similar to the created state: the container sits there doing nothing until it is called upon. From there, the container can either be restarted or, if you have no more use

for it, removed from the system. With that in mind, let's see how this works in practice using Compose.

First of all, let's check what containers are currently running. To do this, we use the ps command:

```
$ docker-compose ps
   Name              Command           State           Ports
------------------------------------------------------------------------
myapp_web_1   bin/rails s -b 0.0.0.0   Up      0.0.0.0:3000->3000/tcp
```

The listing includes the container name, the command used to start it, its current state, and its port mappings. Here you can see the container for our Rails server; it's still running from when we previously ran docker-compose up -d (Up means it's running).

If we now wanted to stop the Rails server, we'd do so with the stop command. By default, the command would apply to *all* services listed in our docker-compose.yml file. For example, to stop all containers in the entire application, we would run:

```
$ docker-compose stop
```

To target a *particular* service, we'd specify the service name after the action like so:

```
$ docker-compose stop <service_name>
```

This may seem like a moot point since, currently, web is the only service we have defined. However, we'll soon be adding more services, starting in Chapter 5, *Beyond the App: Adding Redis*, on page ?. It's common to want to target commands at a specific service, so it's very useful to remember this pattern—particularly as all the docker-compose commands follow it.

Let's go ahead and stop the web service:

```
$ docker-compose stop web
Stopping myapp_web_1 ... done
```

Loading localhost:3000 in the browser will now fail, and listing the containers will report that the Rails server has terminated:

```
$ docker-compose ps
   Name              Command           State    Ports
-------------------------------------------------------
myapp_web_1   bin/rails s -b 0.0.0.0   Exit 1
```

Having stopped a container, we can start it again with the start command:

```
$ docker-compose start web
```

```
Starting web ... done
```

There's also a restart command that's handy if, for example, you've made some config changes and want the Rails server to pick them up.

```
$ docker-compose restart web
Restarting myapp_web_1 ... done
```

The various Compose commands we've seen all rely on underlying docker commands.[3] However, we won't cover those in detail since we'll be using Compose from now on. Compose gives us all the power of the lower-level docker commands, but with simpler, app-centric commands.

3. https://docs.docker.com/engine/reference/commandline/start/