BitPress

DOCKER

For PHP Developers

BY PAUL REDMOND

A Guide to Using Docker for PHP Development

Docker for PHP Developers

Paul Redmond

Thanks for buying my book! If you're interested in the accompanying screencasts and learning more about the full Docker for PHP Developers bundle, head on over to https://bitpress.io/docker-for-php-developers.

I create books, screencasts, and online content to help you and your team create fantastic software. Visit me at https://bitpress.io and follow along on Twitter @paulredmond.

Thanks for reading, it was my pleasure writing this book, and I would love to hear from you if you have praise, comments, or questions.

Paul Redmond
paul@bitpress.io

Contents

Chapter 1: Up and Running

When I first started using Docker, the ecosystem was daunting and all brand new. While I would argue that Docker is still daunting in many ways to developers, tooling and OS support are improving the learning curve.

It has never been a better time to work with Docker. When I first started using Docker, the only option on OS X was using Docker Machine with VirtualBox. The experience was slow and never felt as good as running Docker natively on Linux. Overall, Docker tools have improved drastically since the initial release of Docker, including native support for both Mac and Windows 10 Pro.

Docker provides an extensive user guide (https://docs.docker.com/engine/userguide/) which I highly recommend you go through later, but first, the goal in this chapter is to get you up and running with Docker on your local machine.

The guide does not string together everything PHP-specific that this book covers in a nice packaged way, but the documentation does provide a foundation for understanding Docker that will be helpful to you if you're just starting out. It might make more sense to you if you go through this book first. If you are anything like me, you learn better by doing and then later going back for deeper understanding once you see it in action.

In my opinion, local development is the most efficient way to work on applications, and I want a Docker development environment that is as close to developing locally as possible.

The goal of this chapter is to install Docker and then introduce you to a few key concepts that are different from more traditional environments you run PHP code within. Then we can start learning how to incorporate Docker into PHP projects from scratch!

Installing Docker on OS X

The first thing you need to do is grab the Docker Community Edition (https://store.docker.com/editions/community/docker-ce-desktop-mac) DMG and run the installer.

Once you are done installing Docker Community Edition for Mac, you should read through the getting started (https://docs.docker.com/docker-for-mac/) documentation to get familiar with Docker and managing docker through the provided menu bar (Figure 1.1).

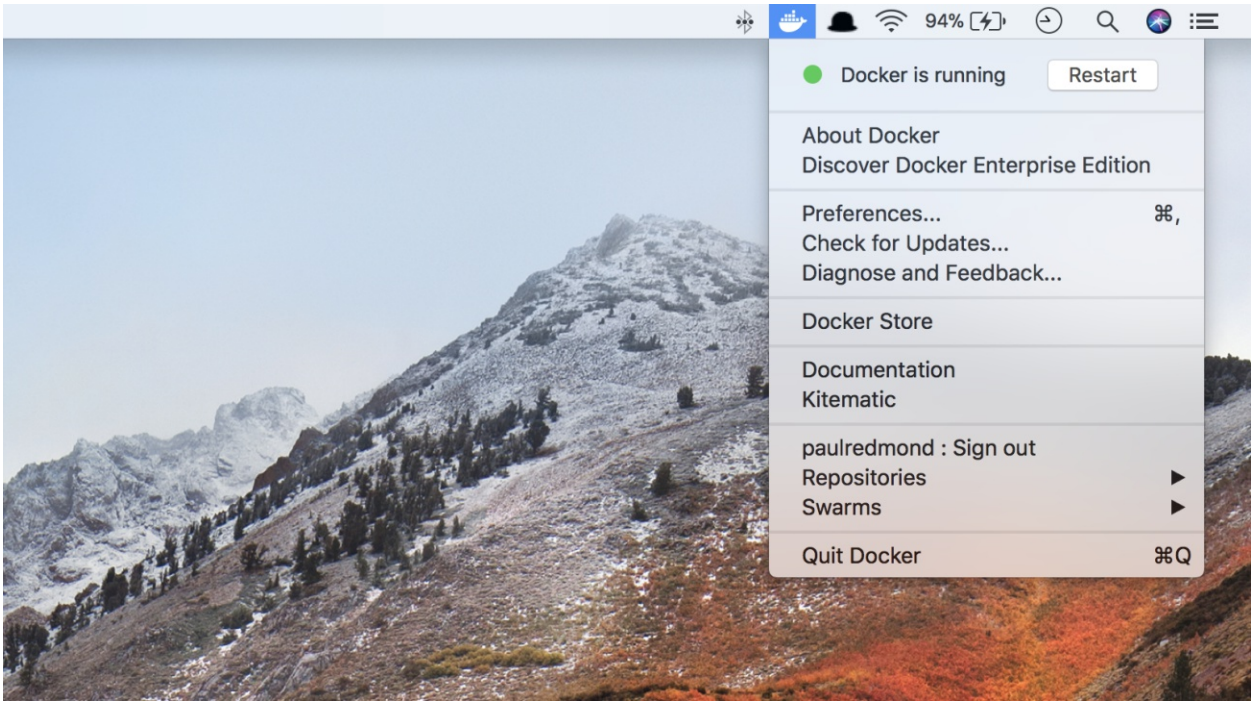


Figure 1.1: The Mac OS X Docker Menu Bar

Once you install everything, fire up your terminal of choice, and you should be able to run the following commands successfully (Listing 1.1).

Listing 1.1: Verifying a Mac Installation

```
$ docker --version
Docker version 17.09.0-ce, build afdb6d4

$ docker-compose --version
docker-compose version 1.16.1, build 6d1ac21

$ docker-machine --version
docker-machine version 0.12.2, build 9371605
```

You might want to read through the Linux installation regardless as that is the typical environment on which Docker runs in production. You can run Docker on any machine that supports it—which is one of its strongest benefits—but Linux tends to be the OS running production Docker servers.

You have successfully installed Docker on Mac OS X!

Installing Docker on Linux

Docker's Linux installation documentation provides instructions for the most common Linux distributions. You should be able to navigate to your distribution from the Ubuntu installation instructions (https://docs.docker.com/engine/installation/linux/ubuntu/), which is probably the most common Linux desktop platform. I do not list each link here, but you should easily find the instructions of your "distro" of choice.

Post-Install Steps for Linux

I strongly recommend following the Post-installation steps for Linux (https://docs.docker.com/engine/installation/linux/linux-postinstall/) after you install Docker. Steps include managing Docker as a non-root user (otherwise you would have to run *sudo docker* all the time) and starting Docker on system boot, which are both a must in my opinion.

Linux users have a few extra steps that Mac and Windows users do not have to worry about: installing *docker-compose* and *docker-machine*. To install Docker Compose and Docker Machine follow the following guides:

1. Install Docker Compose (https://docs.docker.com/compose/install/)
2. Install Docker Machine (https://docs.docker.com/machine/install-machine/)

Once you have installed docker, docker-compose, and docker-machine you should finally be able to test that everything worked (Listing 1.2).

Listing 1.2: Verifying a Linux Installation

```
$ docker --version
Docker version 17.03.1-ce, build c6d412e

$ docker-compose --version
docker-compose version 1.11.2, build dfed245

$ docker-machine --version
docker-machine version 0.10.0, build 76ed2a6
```

You have successfully installed Docker on Linux!

Installing Docker on Windows

Like the OS X offering, Docker for Windows (https://docs.docker.com/docker-for-windows/) has improved working with Docker on Windows drastically. Windows 10 Pro users have access to Hyper-V which allows Docker to run natively on Windows. Unfortunately, Windows 10 Home users must use a virtual machine.

If you have Windows 10 Pro, follow the Windows installation guide (https://docs.docker.com/docker-for-windows/install/). VirtualBox will no longer work after installing Docker (which installs Hyper-V). You can use the Windows 10 Home instructions below if you can't or don't want to lose VirtualBox support.

If you don't have Windows 10 Pro, but instead have Windows 10 Home, you need to install Docker Toolbox (https://docs.docker.com/toolbox/overview/) for Windows. The Docker Toolbox uses VirtualBox and other tools to help interact with Docker. It is not as fast as native Docker support, but at least it is now possible to run Docker on Windows 10 Home. File permissions might be a challenge because of how your machine mounts a volume with VirtualBox.

Once you have installed Docker, you should be able to confirm that Docker is working as expected (Listing 1.3).

Listing 1.3: Verifying a Windows 10 Installation

```
$ docker --version
Docker version 17.03.1-ce, build c6d412e

$ docker-compose --version
docker-compose version 1.11.2, build dfed245

$ docker-machine --version
docker-machine version 0.10.0, build 76ed2a6
```

Windows 10 Home users must use the shell provided by Docker Toolbox to run Docker commands.

You might want to read through the Linux installation regardless as that is the typical environment on which Docker runs in production. You can run Docker on any machine that supports it—which is one of its strongest benefits—but Linux tends to be the OS running production Docker servers.

You have successfully installed Docker on Windows!

Running Your First Docker Container

Now that you have docker running on your platform of choice let's use it to run the "Hello World" docker container and make sure docker is working properly (Listing 1.4):

Listing 1.4: Running Docker's Hello World Container

```
$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
78445dd45222: Pull complete
Digest: sha256:c5515758d4c5e1e838e9...
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working
correctly.

To generate this message, Docker took the following steps:

 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the
Docker Hub.
 3. The Docker daemon created a new container from the image
which is how Docker images run.
 4. The Docker daemon streamed that output to the Docker client,
which sent it to your terminal.
```

When you run *docker run hello-world*, Docker looks for the image locally. If the Docker image doesn't exist, Docker "pulls" the image and then runs it. After the image is downloaded, run *docker images* to see the *hello-world* image locally (Listing 1.5):

Listing 1.5: Output Local Docker Images

```
$ docker images | grep hello-world
hello-world   latest   48b5124b2768    3 months ago    1.84 kB
```

You can remove docker images with *docker rmi <hash>*, so running *docker rmi 48b5124b2768* in the example above would delete the local image as long as no containers depend on it. If you try to remove it, you get an error (Listing 1.6):

Listing 1.6: Attempt to Remove the Hello World Image

```
$ docker rmi 48b5124b2768
Error response from daemon: conflict: unable to delete
48b5124b2768 (must be forced) - image is being used by stopped
container fd0d5cd7d793
```

You cannot remove the image until you remove the stopped container; to remove the container, we need to know the container's ID. The error message conveniently listed the ID, but you should get used to getting a list of containers with the *docker ps* command (Listing 1.7):

Listing 1.7: Listing Docker Containers

```
$ docker ps
CONTAINER ID  IMAGE  COMMAND  CREATED  STATUS  PORTS  NAMES

# List all containers
docker ps -a
CONTAINER ID  IMAGE        COMMAND     CREATED        STATUS
fd0d5cd7d793  hello-world  "/hello"    8 minutes ago  Exited
...
```

Running *docker ps* without any flags only lists running containers. In the second example, *docker ps -a* lists **all** containers—both stopped and currently running. To remove the image we first need to remove the container with the *docker rm* command,

and then we can delete it (Listing 1.8):

Listing 1.8: Remove the Hello World Container and Image

```
$ docker rm fd0d5cd7d793
fd0d5cd7d793

$ docker rmi 48b5124b2768
Untagged: hello-world:latest
Untagged: hello-world@sha256:c5515758d4c5e1e838e9...
Deleted: sha256:48b5124b2768d2b917ed...
Deleted: sha256:98c944e98de8d3509710...
```

The *docker rmi* command is used to remove one or more images. You can list multiple image IDs separated by a space to remove multiple images with one command (Listing 1.9):

Listing 1.9: Removing multiple images

```
# Example removing multiple image ids
$ docker rmi 48b5124b2768 8672b4215b84

# Remove images by name:tag
$ docker rmi php:latest
```

Ready to Ship

Hopefully, you don't get sick of my dry, witty conclusion title puns.

We walked through installing Docker on OS X, Linux, and Windows. You downloaded your first Docker image and ran a container. Things might feel weird or confusing right now, but these patterns quickly become second nature as you work through this book.

Although we run images with *docker run* in various parts of the text, we use Docker compose to make it easier to repeat running your environments on other machines including your coworkers' development machines. Everyone has the same repeatable

environment. Using Docker does not mean you are free from all infrastructure or environment issues, but it does mean your environments are more repeatable, consistent, and easier to set up.

Chapter 2: PHP Container Basics

In this chapter we are going to cover the basics of running a PHP container with Docker. Before we get into the more exciting stuff, we need to learn how to build images, start containers, and copy files into them. Along the way, you'll work with basic Docker commands and start to get a feel for how to work with Docker on the command line.

Using the command line to build images, we'll extend our images from the official PHP Docker images (https://hub.docker.com/_/php/). I find the official image simplifies my setup and I can focus on configuring applications and not worrying about the low-level details of installing PHP.

Creating a New Project

When creating a new Docker project, the main file used to build images is the *Dockerfile*. This file is a set of instructions that define building images, each **step** creating a new layer on top of the previous. If this doesn't make much sense right now, don't worry, you don't need to be an expert to start being productive. I recommend that you keep the Dockerfile reference (https://docs.docker.com/engine/reference/builder/) handy as you work through this book.

The first task is creating the necessary files for our first Docker image. In the directory of your choice, create the following files (Listing 2.1):

Listing 2.1: Creating Docker files

```
$ mkdir -p ~/Code/docker-phpinfo
$ cd ~/Code/docker-phpinfo

# Create the project files
$ touch Dockerfile docker-compose.yml index.php
```

The *index.php* file will be the only source file in this chapter that we'll use to demonstrate changes to our builds, and in later chapters we will work with web frameworks.

The *docker-compose.yml* file is a configuration file that will help you run containers with the *docker-compose* CLI tool. If you are not familiar with Docker Compose, don't worry, we will use it throughout this book.

To start, we will define the Dockerfile to extend the PHP Apache image and copy the *index.php* file (Listing 2.2):

Listing 2.2: Defining the Dockerfile Instructions

```
FROM php:7.1.9-apache

LABEL maintainer="Paul Redmond"
COPY index.php /var/www/html
```

The *FROM* instruction means we are extending another image. Think of it like PHP class inheritance. You inherit the base image which takes care of things like installing Apache and building PHP from source. The official PHP image is doing most of the work for us!

As outlined in the README found on https://hub.docker.com/_/php/, you copy the source files of your project to */var/www/html* using COPY. In our case we'll copy the *index.php* file into the image at */var/www/html/index.php*. Note that the COPY instruction

can take an individual file or a directory.

The LABEL instruction is how you add metadata to an image. In this case, we are following the recommended guideline for setting a maintainer, which helps others know who is maintaining the Dockerfile. You can see the metadata for an image by running *docker inspect*:

```
# docker inspect <image_name>
$ docker inspect php:7.1.9-apache
```

Next, let's output PHP's configuration to the browser so we can verify our PHP setup (Listing 2.3):

</> Listing 2.3: Update the index.php File

```
<?php phpinfo(); ?>
```

Running the PHP Container

It's time to run our first image and inspect the PHP environment. In order to run it, we need to build it using the *docker build* command (Listing 2.4):

>_ Listing 2.4: Build the Docker Image

```
$ docker build -t phpinfo .
$ docker run -p 8080:80 -d --name=my-phpinfo phpinfo
```

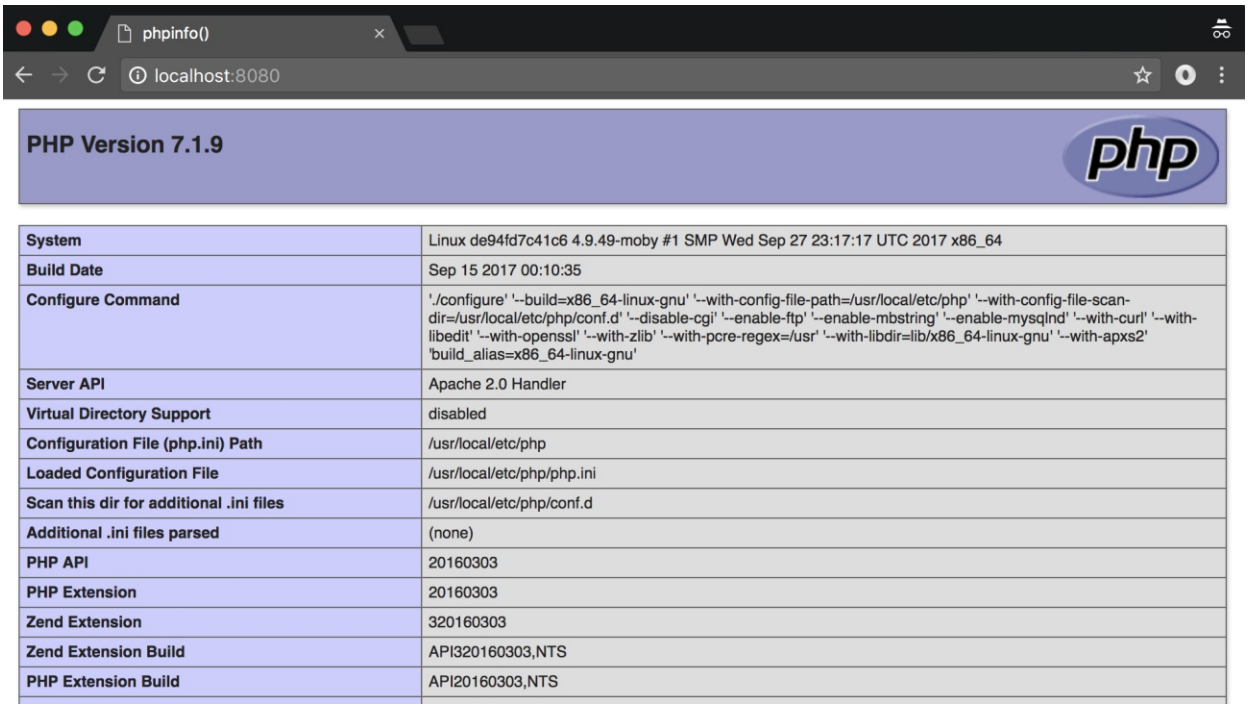
The build command has a *-t* flag, which tags the image as *phpinfo*, and the last argument (.) is the path where Docker will look for our files.

The run command runs a container with the tagged *phpinfo* image, using the *-p* flag to map port *8080* on your machine to port *80* in the container, which means that we'll use port *8080* locally to access our application.

The *--name* flag assigns a name to the running container that you can use to issue further commands, like *docker stop my-phpinfo*. If you don't provide a name, Docker creates a random auto-generated name for you.

The *-d* flag (detach) is used to run the container in the background. Without the *-d* flag Docker runs in the foreground.

Next, point your browser to http://localhost:8080, and you should see the output from *phpinfo()* (Figure 2.1):



System	Linux de94fd7c41c6 4.9.49-moby #1 SMP Wed Sep 27 23:17:17 UTC 2017 x86_64
Build Date	Sep 15 2017 00:10:35
Configure Command	'./configure' '--build=x86_64-linux-gnu' '--with-config-file-path=/usr/local/etc/php' '--with-config-file-scan-dir=/usr/local/etc/php/conf.d' '--disable-cgi' '--enable-ftp' '--enable-mbstring' '--enable-mysqlnd' '--with-curl' '--with-libedit' '--with-openssl' '--with-zlib' '--with-pcre-regex=/usr' '--with-libdir=lib/x86_64-linux-gnu' '--with-apxs2' 'build_alias=x86_64-linux-gnu'
Server API	Apache 2.0 Handler
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/usr/local/etc/php
Loaded Configuration File	/usr/local/etc/php/php.ini
Scan this dir for additional .ini files	/usr/local/etc/php/conf.d
Additional .ini files parsed	(none)
PHP API	20160303
PHP Extension	20160303
Zend Extension	320160303
Zend Extension Build	API320160303,NTS
PHP Extension Build	API20160303,NTS

Figure 2.1: phpinfo

Our container is running, which means that we can inspect it from the command line by issuing the *docker ps* command. Unless you are already running something with Docker, you should see just one container (Listing 2.5):

Listing 2.5: The docker ps Command (partial output)

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
88c6424ebb5b        phpinfo             "apache2-foreground"
```

The *ps* command outputs the names of the containers, which you can use to issue commands like *docker stop my-phpinfo* and *docker start my-phpinfo*. The *my-phpinfo* argument is the name we provided the container in the *docker run* command.

If a container is stopped, running *docker ps* does not show the container in the list; however, you can still see all containers by running *docker ps -a*. To remove a container, you could run *docker rm my-phpinfo*.

In practice, running containers with *docker run* isn't going to help your productivity. In fact, it will be kind of clunky when you share the application with others. That's exactly what Docker Compose will help us automate, so let's dive in!

Before we start working with Docker Compose, shut down the container you are running:

```
# Stop the container
$ docker stop my-phpinfo

# Remove the container
$ docker rm my-phpinfo
```

Running Containers with Docker Compose

What is Docker Compose? From the Docker Compose overview page (https://docs.docker.com/compose/overview/):

> *Compose is a tool for defining and running multi-container Docker applications.*

One my biggest breakthroughs when I was learning about Docker was running containers with Docker Compose (https://docs.docker.com/compose/), because it simplifies running your stack over individually running containers with *docker run*.

Your applications will need dependencies like MySQL, Redis, etc., and with Docker

Compose, we can automate the orchestration of these services. Can you imagine multiple *docker run* commands and networking everything together by hand? Me neither.

In more traditional environments, all dependencies run on the same operating system (or virtual machine). However, with Docker, you can break up your application into multiple containers. This separation can simplify your setup and lets you scale parts of your application independently.

Before we start adding services like MySQL in future chapters, let's just replicate what we were doing with *docker run* inside the *docker-compose.yml* file to get started (Listing 2.6):

Listing 2.6: Your First docker-compose.yml File

```
version: "3"
services:
  phpinfo:
    build: .
    ports:
      - "8080:80"
```

The *services* key defines one service called *phpinfo*.

Inside the *phpinfo* service, the *build* key references a dot (.), which means we expect the *Dockerfile* in the current path. Lastly, the *ports* key contains an array of port maps from the host server, just like our previous *docker run -p 8080:80* flag. The port format is: *<host_port>:<container_port>*, which in our case means that port *8080* on the local machine will map to port *80* inside the container.

We are using version three (https://docs.docker.com/compose/compose-file/), which is the recommended version at the time of writing. I use the documentation frequently, and I recommend that you bookmark it and use it as a reference.

With our service defined, now anyone that comes along and needs to run this project can simply run *docker-compose up* (Listing 2.7):

Listing 2.7: Using Docker Compose

```
$ docker-compose up --build

# Or, if you want to run it in the background
$ docker-compose up -d --build

# Now list all the containers running
$ docker-compose ps
```

After running the command, you should see the output from the *phpinfo()* function when you visit http://localhost:8080.

If you ran your containers in the background (*-d*), you can use the *stop* command to stop everything (Listing 2.8):

Listing 2.8: Stopping Containers with Docker Compose

```
# From the root of the project
$ docker-compose stop
```

Here are commonly used commands that you should become familiar with (Listing 2.9):

Listing 2.9: Additional Docker Compose Commands

```
# List running containers that Docker Compose is managing
$ docker-compose ps

# Restart the containers
$ docker-compose restart
```

```
# Restart a specific container
# matches the service key in docker-compose.yml
$ docker-compose restart phpinfo

# Remove stopped containers
$ docker-compose stop && docker-compose rm

# Stop containers and remove containers, networks,
# volumes, and images created
$ docker-compose down

# Remove named volumes
$ docker-compose down --volumes
```

Don't worry about memorizing these commands. You can always run *docker-compose --help* to get a list of commands, and run, for example *docker-compose up --help* to get help on subcommands. You'll also get plenty of practice setting up Docker Compose and running containers throughout this book.

Basic PHP INI Changes

We have the *phpinfo()* settings handy, so let's make a few small tweaks to the *php.ini* file and validate our changes. We'll also jump into a running container and peek around, which feels very much like SSH to me (but it's nothing like that).

According to the PHP image documentation, the *php.ini* file is located at */usr/local/etc/php/php.ini*, however, I want to show you how to find the location on your own. We will then make a few adjustments, rebuild the image, and verify our INI changes.

First we need to find out the PHP container's ID, so we can use it to run bash inside the container (Listing 2.10):

Listing 2.10: Find the Running Container ID

```
# Run the image if you are not already doing so
$ docker-compose up -d

# Get the image ID
$ docker ps
CONTAINER ID
c0ee14f0c047
```

The container ID that you see will be different. Copy the container ID for your output and use it to run the following commands (Listing 2.11):

Listing 2.11: Run bash in the container

```
$ docker exec -it c0ee14f0c047 bash

# Inside the container, run php --ini
root@c0ee14f0c047:/var/www/html# php --ini

Configuration File (php.ini) Path: /usr/local/etc/php
Loaded Configuration File:         (none)
Scan for additional .ini files in: /usr/local/etc/php/conf.d
Additional .ini files parsed:      (none)
root@c0ee14f0c047:/var/www/html#
```

You can exit the container by hitting "Ctrl + D" or typing "exit."

Although the image has no INI configuration file defined, we can create our own in the project, and then copy it into the image (Listing 2.12):

Listing 2.12: Create a php.ini File and Set the Timezone

```
# Create a config folder
$ mkdir config/
```

```
# I am partial to Phoenix, I live here after all...
# and we don't observe daylight savings time, win!
$ echo "date.timezone = America/Phoenix" >> config/php.ini
```

Our php.ini file has one *date.timezone* setting, which configures the timezone to America/Phoenix. I prefer *UTC*, but I want to show you a non-default for demonstration purposes.

We can now copy our php.ini file into the image at the correct path listed in the *php --ini* command by adding a COPY instruction in the DO (Listing 2.13):

</> Listing 2.13: Copy the php.ini File Into the Container

```
FROM php:7.1.9-apache

LABEL maintainer="Paul Redmond"
COPY config/php.ini /usr/local/etc/php/
COPY index.php /var/www/html
```

In order to get our *php.ini* file into the container, we need to build the image again (Listing 2.14):

>_ Listing 2.14: Rebuild the phpinfo image

```
$ docker-compose stop
$ docker-compose up -d --build
```

The image should now contain a *php.ini* config file and you should see the following "datetime" change (Figure 2.2):

date

date/time support	enabled
"Olson" Timezone Database Version	2017.2
Timezone Database	internal
Default timezone	America/Phoenix

Directive	Local Value	Master Value
date.default_latitude	31.7667	31.7667
date.default_longitude	35.2333	35.2333
date.sunrise_zenith	90.583333	90.583333
date.sunset_zenith	90.583333	90.583333
date.timezone	America/Phoenix	America/Phoenix

Figure 2.2: phpinfo datetime changes

Composed and Ready for Adventure

We covered a bunch of ground quickly. In a nutshell, you learned the following:

- Extending an existing Docker image
- Building a custom Docker image
- Running custom docker images
- Using Docker Compose to automate running containers
- Executing a bash shell in a running container
- Debugging and adding PHP INI files

Using Docker requires a new way of thinking, and can be quite a transition. If you feel overwhelmed or confused right now, don't worry. I've been there too. You'll get plenty more wrench time running commands and making changes as you start going over more practical uses of Docker by running real-world applications in this book!

Chapter 3: LAMP Baby!

We've learned how to build images and run containers, and we are ready to work on a complete LAMP stack with Docker. Instead of just using an *index.php* file, we will install an entire application framework and a database. We also need to configure the web server to handle the application requests and copy the source code into the image.

Along the way, you will learn how to run multiple containers with Docker Compose and expose ports locally in order connect to a MySQL server running in a Docker container. We will also work on configuring the application to connect to a database, and learning how Compose provides networking between them out-of-the-box.

Setting up the LAMP Project

A nice benefit of showing you examples with a complete application is that you can start to figure out how you prefer to organize your projects using Docker. We will create the core files we need for our LAMP Docker automation alongside our project and start to get a feel for Docker file organization.

The framework we are going to use for this chapter is Lumen (https://github.com/laravel/lumen), an API framework by Laravel. You will see other frameworks later in the text, such as Laravel and Slim, but the core focus is around showing you how to use Docker.

With all that explaining out of the way, let's create the initial project files. I am also assuming that you already have PHP Composer (https://getcomposer.org/) installed on your machine and know how to use it (Listing 3.1):

Listing 3.1: Creating the core files for our LAMP project

```
$ cd ~/Code
$ composer create-project --prefer-dist laravel/lumen docker-lamp
$ cd ~/Code/docker-lamp
$ touch Dockerfile docker-compose.yml

# Create a php and apache directory
$ mkdir -p .docker/{php,apache}
$ touch .docker/php/php.ini
$ touch .docker/apache/vhost.conf
```

You can organize your Docker builds in various ways, and I will demonstrate a few throughout this book. I prefer to keep my Docker files with my application code, so I can efficiently work with the application and Docker configuration together.

In Listing 3.1, the *mkdir* command created a *.docker* folder, with two subfolders (*php* and *apache*), which is where we'll put configuration files. The *.docker* folder is my **personal convention** to organize Docker-specific files. I am not 100% sold on it, but it works well most of the time. One downside is that your files are hidden, so you could just use *docker/* as the folder name instead.

Feel free to adapt file organization to your preferences, but you might consider sticking with mine until you get more comfortable with what's going on.

Now that we've created our project files, our first goal is getting the default Lumen page loaded in a browser. The first step is building a Docker image for our Lumen code, much like we did in Chapter 2 (Listing 3.2):

Listing 3.2: The Initial LAMP Dockerfile

```
FROM php:7.1.9-apache

LABEL maintainer="Paul Redmond"
```

```
COPY .docker/php/php.ini /usr/local/etc/php/
COPY . /srv/app
COPY .docker/apache/vhost.conf /etc/apache2/sites-available/000-
default.conf
```

Not much is new here, except for the vhost.conf file and *COPY* instructions. The last *COPY* instruction will replace the *000-default.conf* file with the contents of *vhost.conf*. The default.conf is the default Apache Vhost file, so we effectively make our configuration the default. In the image, the name will still be *000-default.conf*, but locally the project file is *vhost.conf*. Think of the last line just like the *cp* command: *cp foo.txt bar.txt*.

The application source files get copied into the */srv/app* folder inside the Docker image, which is a convention I use as the path for my web application files. You are free to use any convention you like, for example, you could use the default Apache path of */var/www/html*.

Like in the last chapter, the Dockerfile also expects to copy in a *php.ini* file, which we will use to define the *date.timezone* setting (Listing 3.3):

</> Listing 3.3: The php.ini file

```
date.timezone = UTC
```

We will add more configuration to this file later on, but for now, we'll just define the *date.timezone* setting. There's something cool about having the PHP INI configuration in a project right at developers' fingertips.

Now, we are going to override the default Apache Vhost file, creating a much more interesting *.docker/apache/vhost.conf* file that will replace the default (Listing 3.4):

Listing 3.4: The default Apache Virtual Host file

```
<VirtualHost *:80>
    # ServerName www.example.com

    ServerAdmin webmaster@localhost
    DocumentRoot /srv/app/public

    <Directory "/srv/app/public">
        AllowOverride all
        Require all granted
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

The VirtualHost definition is a slightly modified version of the default file that ships with the official image. If you recall, we copied our source code to */srv/app*, and we defined our document root accordingly as *DocumentRoot /srv/app/public*.

We enable a directory override for development and use Apache 2.4's *Require all granted* which you might know as *Allow from all* in previous versions. You can provide a *ServerName* if you want, but for our purposes, we'll use *localhost*.

Next, we need to add our application service to the project's *docker-compose.yml* file. We will map ports and mount a volume (https://docs.docker.com/engine/tutorials/dockervolumes/) so that changes made locally reflect immediately in the running container (Listing 3.5):

</> Listing 3.5: The docker-compose.yml File

```
version: "3"
services:
  app:
    build: .
    ports:
      - "8080:80"
    volumes:
      - .:/srv/app
```

We define an *app* service with the same port mapping we've already seen in the last chapter. More interestingly, the *volume* key accepts an array of folders separated by a colon (:). The first path is the local path to the folder or file, and the second is the desired path inside the container. Volumes allow you to avoid running *docker build* (which copies the files into the container) every time you make a source code change while developing.

With our PHP and Vhost configurations updated, we're ready to run Docker Compose and verify that our configuration is working (Listing 3.6):

>_ Listing 3.6: Running the application with docker-compose

```
$ docker-compose up --build
```

Adding the *--build* flag will build the images before running the containers defined under the services key. After the containers are running, you can visit http://localhost:8080 to verify everything is working as expected, and you should see something like "Lumen (5.5.1) (Laravel Components 5.5.*)." which means that our application is working.

Installing PHP Modules

Our next step in getting a LAMP environment going is installing PHP modules.

Specifically, we need to install the *pdo_mysql* extension to connect to a MySQL database container.

So how do you install modules with the official PHP image?

The PHP Docker hub page (https://hub.docker.com/_/php/) mentions three helper scripts to work with PHP extensions: *docker-php-ext-configure*, *docker-php-ext-install*, and *docker-php-ext-enable*. To run these commands during a Docker build, you use the RUN instruction.

Here's how the RUN instruction looks (Listing 3.7):

Listing 3.7: Installing the PDO MySQL Module

```
FROM php:7.1.9-apache

LABEL maintainer="Paul Redmond"
COPY .docker/php/php.ini /usr/local/etc/php/
COPY . /srv/app
COPY .docker/apache/vhost.conf /etc/apache2/sites-available/000-
default.conf
RUN docker-php-ext-install pdo_mysql
```

The RUN instruction executes commands in a new layer, and in our case runs the *docker-php-ext-install* command. You will see more examples of the RUN command going forward.

Docker creates a new layer for each RUN command. You can optimize the number of layers by combining multiple commands within a single RUN instruction. This is best explained with an example:

```
# Valid but creates separate layers
RUN cp /srv/.env.example /srv/.env
RUN touch /tmp/foo
```

```
# Combines commands into one RUN instruction
RUN cp /srv/.env.example /srv/.env \
    && touch /tmp/foo
```

As you might have guessed, we need to run a new build to install the module (Listing 3.8):

>_ Listing 3.8: Build the Image after the latest Dockerfile changes

```
$ docker-compose stop
$ docker-compose build
```

When you run the build command, you should see some build output fly by about the installation of the *pdo_mysql* PHP module from source. While the build finishes, you can start configuring the application database container.

The Database Container

The database service will use the official MariaDB 10.1 image (https://hub.docker.com/_/mariadb/), but feel free to use any MySQL variant you want. Most MySQL variants (if not all of them) have official Docker Hub images.

The database will be another service in the Docker Compose file, which means that when we run *docker-compose up*, both containers will start.

Defining The MariaDB Service

This is what the database service in *docker-compose.yml* looks like (Listing 3.9):

</> Listing 3.9: Defining the Database Container

```
version: "3"
services:
  app:
    build: .
```

```
    depends_on:
      - mariadb
    ports:
      - "8080:80"
    volumes:
      - .:/srv/app
    links:
      - mariadb:mariadb
  mariadb:
    image: mariadb:10.1.21
    ports:
      - "13306:3306"
    environment:
      - MYSQL_DATABASE=dockerphp
      - MYSQL_ROOT_PASSWORD=password
```

The MariaDB service introduces the *image* key, which points to version *10.1.21* of the MariaDB Docker Hub image. The image format is just like the Dockerfile format we've been using for the PHP image: *<name>:<tag>*. If you provide *image: mariadb* with no tag, Docker Compose will use the "latest" tag.

Using Tagged Image Versions

Providing a tag version is a good practice to avoid unexpected changes to your application's environment without your explicit control. Using *latest* should be used with caution on a real project.

Another interesting line in this file is the database service *ports* key. We mapped port 13306 locally to 3306 inside the container. Using this non-standard local port avoids collision with any local MySQL instances running, which is my convention. As you will see in a second, exposing this port allows you to connect to the database container from your computer.

Next, the *environment* key defines environment variables for the container that we are

using to set the root password and the name of the database that we want to use. The MariaDB documentation outlines which environment variables are available that you can use to configure the database.

Last, we added *links* to the app definition, which links to the database container. The *links* configuration is the default, which means it's redundant, but I've left it so you can understand how it works. The *links* format specifies the name and a link alias (*<service>:<alias>*), which enables you to use the alias to communicate with the container.

While links are not required for containers to communicate—they can reach each other using the service name—it's important to understand that defining the link as "*- mariadb*" is shorthand for "*- mariadb:mariadb*". If you want to define an alias for the service, use the *links* key with something like "*- mariadb:db*", and then you would use *db* as the hostname to communicate with MariaDB from the app container.

Offical Docker Image Readme File

Official Docker images usually provide thorough README files, which outlines relevant details like configuration, using the image, and getting technical support. It's advisable to at least skim through the documentation on how to use the image.

Connecting to the Database Locally

Connecting to the database with a GUI tool is a **must** for my development workflow. I use migrations with Laravel, but I use Sequel Pro (https://www.sequelpro.com/) all the time to inspect the database.

We've exposed port 13306 to allow us to connect to the container database locally. If you have MySQL installed locally, you can test the connection using the CLI or GUI (listing 3.10):

Listing: 3.10: Connect to the MariaDB Database from the Host Machine

```
# Let's delete and restart the containers
$ docker-compose stop
$ docker-compose rm -v # Remove anonymous volumes attached
$ docker-compose up --build

# Once the database container finishes starting,
# open another tab and connect via mysql-client or a GUI
$ mysql -u root -h 127.0.0.1 -P13306 -ppassword
$ mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| dockerphp          |
| information_schema |
| mysql              |
| performance_schema |
+--------------------+
4 rows in set (0.00 sec)
```

You should see a *dockerphp* database listed, which matches the *MYSQL_DATABASE* environment variable we defined in docker-compose.yml.

Here's an example from Sequel Pro (Figure 3.1):

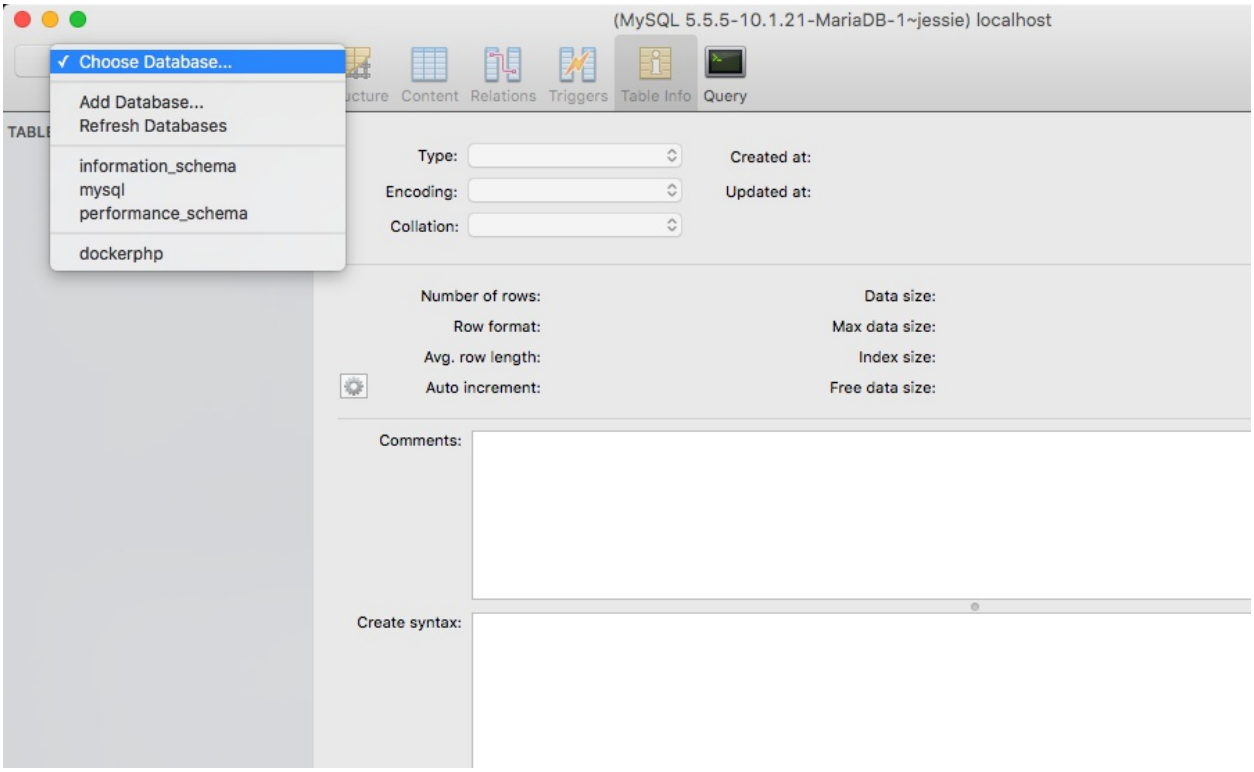


Figure 3.1: Connecting to the MariaDB Container with a GUI

Application Database Connection

We are ready to configure the application to use MariaDB and run the built-in database migrations provided in Lumen. To configure the database connection, Lumen uses a *.env* file in the root of the project (Listing 3.11):

Listing 3.11: Update the Database Configuration

```
APP_ENV=local
APP_DEBUG=true
APP_KEY=
APP_TIMEZONE=UTC

DB_CONNECTION=mysql
DB_HOST=mariadb
DB_PORT=3306
DB_DATABASE=dockerphp
DB_USERNAME=root
DB_PASSWORD=password
```

One thing that confused me when I first started using Docker was the peculiar hostnames used to communicate between containers. If you look closely, we are using the hostname *mariadb* in the container to connect to the database. The equivalent console connection would look like this for a visualization:

```
# Example using localhost
$ mysql -u root -h 127.0.0.1 -ppassword

# Using the Docker network alias
$ mysql -u root -h mariadb -ppassword
```

For now, just understand that the *links* key on the right side will be the name of the MariaDB host you use to connect inside your app container. If you don't define a *links* key, you can use the name of the service key (i.e., app or mariadb) to reach services. On the other hand, if you configure an alias like the following:

```
services:
  app:
    links:
      - mariadb:db
```

Then connecting to the container would look like this:

```
$ mysql -u root -h db -ppassword
```

If you want to learn more about how Docker Compose networking works, check out the networking documentation (https://docs.docker.com/compose/networking/).

Migrating the Database

We are ready to run some database queries to test our database connection using Lumen's built-in database migration command. We'll need to execute the commands inside the *app* container for this to work properly because we've configured our *.env* file to use the *mariadb* hostname, which is only available in the provided Docker

Compose network.

Let's run bash inside of the container and execute our migrations to test the database connection (Listing 3.12):

Listing 3.12: Running Lumen's migrate Command

```
# Run the containers and jump into the app container
$ docker-compose up
$ docker ps # note the app container id
$ docker exec -it 6a50b2398826 bash

# Inside the app container, navigate to the project
# and run the migrate command
$ cd /srv/app
$ php artisan migrate
Migration table created successfully.
Nothing to migrate.
```

After running the migration, you should see a new table called *migrations* in the database. Check with your local GUI by refreshing the tables or using the MySQL CLI command.

Feel free to explore creating database migrations, models, and writing routes for Lumen at this point. Our setup is capable enough to run a LAMP application, and you can quickly add other container services like Redis or Memcache to practice running different services together.

PHP Module Configuration

One of my favorite parts of using Docker is how close my server configuration is to my codebase. I can quickly update PHP module configurations and rebuild the image to get my new changes in place. With confidence, I know that when I release my new changes, production will get the same configuration.

I install the OPcache (http://php.net/manual/en/book.opcache.php) module on every project, so it's an excellent candidate to walk you through configuring PHP modules with Docker.

You are probably familiar with APC and OPcache, but for those who are not familiar, the OPcache module description is as follows:

> *OPcache improves PHP performance by storing precompiled script bytecode in shared memory, thereby removing the need for PHP to load and parse scripts on each request.*

That sounds great for production, but not so much for development environments. To deal with this issue, we will cover how to make INI configuration more flexible in a later chapter with **environment variables**. Right now we will just focus on enabling and configuring PHP modules so you can get a feel for working with them.

If you run *php -m | grep opcache* inside the app container, the *opcache* module is not installed yet, so let's go ahead and add it (Listing 3.13):

</> Listing 3.13: Installing the OPcache module

```
FROM php:7.1.9-apache

LABEL maintainer="Paul Redmond"
COPY .docker/php/php.ini /usr/local/etc/php/
COPY . /srv/app
COPY .docker/apache/vhost.conf /etc/apache2/sites-available/000-
default.conf
RUN docker-php-ext-install pdo_mysql \
    && docker-php-ext-install opcache
```

Next, we will provide some OPcache configuration settings in the *php.ini* file, which is located at *.docker/php/php.ini* (Listing 3.14):

Listing 3.14: OPcache configuration

```
date.timezone = UTC

[opcache]
opcache.enable=1
opcache.revalidate_freq=0
opcache.fast_shutdown=1

; 0 or 1. 0 is recommended in production
; and will require a restart when files change.
opcache.validate_timestamps=1

; Keep this above the number of files in project
; You can check how many files you have with
; `find . -type f -print | grep php | wc -l`
opcache.max_accelerated_files=6000

; Caches duplicate strings into one shared immutable value
opcache.interned_strings_buffer=16
```

The INI values and comments are from an OPcache write-up on scalingphpbook.com (https://www.scalingphpbook.com/blog/2014/02/14/best-zend-opcache-settings.html), which is an excellent resource for scaling PHP applications.

Alternatively, we could have provided a custom *opcache.ini* file and copied it into the directory configured to scan additional INI files (*/usr/local/etc/php/conf.d*). I prefer to organize each extension's settings into separate files, but for this example, I kept it simple. There's nothing wrong with keeping the settings in one *php.ini* file either, if you prefer.

With our new Dockerfile and INI changes in place, we can now rebuild the image and verify that the OPcache module is configured with our overrides.

You can build all images with *docker-compose build*. However, we can specify which image to build based on the service name if we only want to build one (Listing 3.15):

>_ Listing 3.15: Build the Latest App

```
$ docker-compose build app
$ docker-compose up
```

At this point, if you add "*phpinfo(); exit;*" at the top of your project's *public/index.php* file, you can verify that OPcache is installed and configured (Figure 3.2):

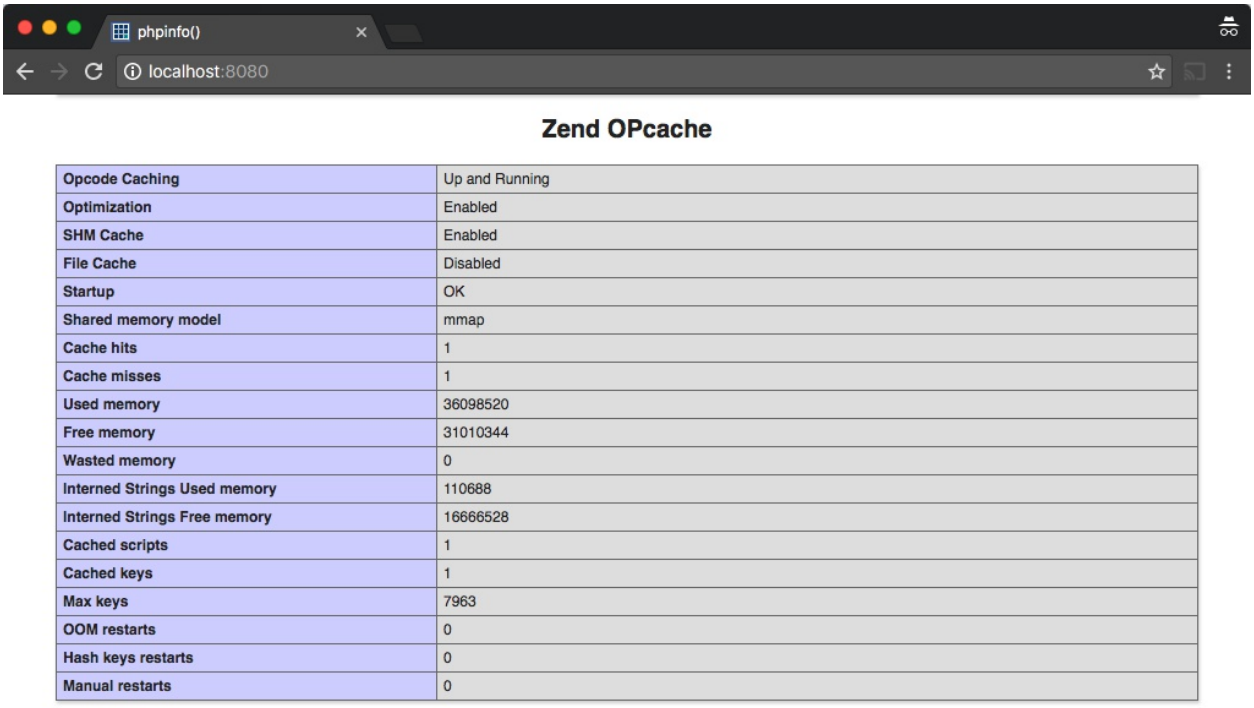


Figure 3.2: Zend OPcache is Enabled

Enabling Apache Modules

To wrap up this chapter, we are going to get "pretty URLs" working in Apache. If you look at the *public/.htaccess* file in our project, you can see that our application needs mod_rewrite (Listing 3.16):

</> Listing 3.16: The Application .htaccess File

```
<IfModule mod_rewrite.c>
    <IfModule mod_negotiation.c>
        Options -MultiViews
    </IfModule>

    RewriteEngine On

    # Redirect Trailing Slashes If Not A Folder...
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule ^(.*)/$ /$1 [L,R=301]

    # Handle Front Controller...
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteRule ^ index.php [L]

    # Handle Authorization Header
    RewriteCond %{HTTP:Authorization} .
    RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
</IfModule>
```

The official Docker image doesn't enable mod_rewrite by default, but you almost always want it with Apache. To enable the module, we can use *a2enmod* in the *Dockerfile* (Listing 3.17):

</> Listing 3.17: Enable mod_rewrite

```
FROM php:7.1.9-apache

LABEL maintainer="Paul Redmond"
COPY .docker/php/php.ini /usr/local/etc/php/
COPY . /srv/app
COPY .docker/apache/vhost.conf /etc/apache2/sites-available/000-
default.conf
```

```
RUN docker-php-ext-install pdo_mysql \
    && docker-php-ext-install opcache \
    && a2enmod rewrite negotiation
```

When you rerun the build, you will see "Enabling module rewrite" on the last step. The negotiation module is already enabled, but it doesn't hurt to add it to the Dockerfile just in case.

Now, if you define a route in *routes/web.php* it should work correctly (Listing 3.18):

</> Listing 3.18: Define a Route to Test mod_rewrite in routes/web.php

```
$router->get('/test', function () {
    return ['test' => 'OK'];
});
```

Make a request to */test* and you should get a similar response (Listing 3.19):

>_ Listing 3.19: Verify mod_rewrite is working

```
$ curl -i http://localhost:8080/test
HTTP/1.0 200 OK
Date: Mon, 09 Oct 2017 14:04:09 GMT
Server: Apache/2.4.10 (Debian)
X-Powered-By: PHP/7.1.9
Cache-Control: no-cache, private
Content-Length: 13
Connection: close
Content-Type: application/json

{"test":"OK"}
```

The mod_rewrite module is working as expected, allowing us to define routes that will be rewritten by the .htaccess file. You can also disable .htaccess files and set the rewrite rules in your Vhost file for more performance.

LAMPed Up

We've covered the basics of running a Docker LAMP environment from scratch, running multiple containers, communicating between containers, and customizing PHP modules.

Using a full PHP framework to work through this chapter, we were able to see how the Dockerfile builds our application code. When I first started trying to add Docker to my applications, I felt perplexed about how to work with Docker and my application together. An essential goal of this book is to ease that burden for you.

At this point, we have a working LAMP environment, but we need to improve our development workflow. In the next chapter, we are going to continue building on our LAMP application by making our configuration more flexible, installing Xdebug, and working with profiling tools.

Chapter 4: Development Tools

In this chapter, we are going focus on our development environment. We will slowly ramp up by learning more about environment variables, and by the end of the chapter, you will have a full development environment with debugging capabilities using XDebug. Along the way, we'll learn how to use environment variables in PHP INI files to make module configuration easy to change in any environment.

Developing applications in containers might be a transition for you if you like to work locally, but if you are using homestead or a virtual machine, you should catch on pretty quick. We will run plenty of commands that will help you get more comfortable using Docker to develop features, as well as debug and profile your applications.

One of the benefits of Docker I've touted is that you will have the same environment for development and production. There's one caveat though: XDebug shouldn't be used in production. Removing XDebug in production presents a bit of a problem in Docker, so we will also look at how to make it disappear in production.

If you are following along, we will continue with the code from Chapter 3; just continuing to work on the same project will do.

Environment Configuration

Environment variables help you create flexible Docker images by allowing you to configure your applications for different environments. For example, in development, you might be connecting to a database within a container, and in production, use something like Amazon Relational Database Service (RDS). You can also use environment variables to keep sensitive data out of your codebase.

You've already seen an example of **how** to use environment variables with Docker: the MariaDB container from Chapter 3 used them to configure the database and root password (Listing 4.1):

</> Listing 4.1: Example Environment configuration in docker-compose.yml

```
services:
  mariadb:
    # ...
    environment:
      - MYSQL_DATABASE=dockerphp
      - MYSQL_ROOT_PASSWORD=password
```

Listing 4.1 is how you define them with Docker Compose, and the equivalent with *docker run* would look like the following (Listing 4.2):

>_ Listing 4.2: Setting Environment Variables with *docker run*

```
# Passing environment variables with the -e|--env flag
$ docker run -e MYSQL_ROOT_PASSWORD=password --name my-db \
  -d mariadb

# Using an external file in the same path
$ docker run --env-file .docker.env --name my-db -d mariadb
```

In the last line, I demonstrated the *--env-file* flag. In Docker Compose, you can also pass environment variables from an external file with the env_file (https://docs.docker.com/compose/compose-file/#/envfile) configuration option:

```
# An env_file configuration example
services:
  mariadb:
    env_file: .docker.env
```

Using an external file provides a starting point that works and yet allows developers to maintain their custom settings.

Setting up Environment Variables

Let's take what we just learned about external environment files and apply it to our project. We will use an external file that is ignored by version control, and provide sensible defaults that each developer can copy as a starting point (Listing 4.3):

>_ Listing 4.3: Create the Docker Environment Files

```
$ touch .docker.env.example .docker.env

# Set an example value for demonstration
$ echo "HELLO=WORLD" >> .docker.env.example
$ cat .docker.env.example > .docker.env

# Ignore the .docker.env file when using VCS
$ echo ".docker.env" >> .gitignore
```

In Listing 4.3, we created an example environment file and a local environment file. Later on, when a new developer first checks out your repository, they will copy the versioned *.docker.env.example* file to the ignored *.docker.env* with *cp .docker.env.example .docker.env*.

Let's add this environment file to *docker-compose.yml* so we can test it out our test environment variable (Listing 4.4)

</> Listing 4.4: Adding the .docker.env file

```
version: "3"
services:
  app:
    build: .
    depends_on:
      - mariadb
```

```
    ports:
      - "8080:80"
    volumes:
      - .:/srv/app
    links:
      - mariadb:mariadb
    env_file: .docker.env
  mariadb:
    image: mariadb:10.1.21
    ports:
      - "13306:3306"
    environment:
      - MYSQL_DATABASE=dockerphp
      - MYSQL_ROOT_PASSWORD=password
```

We added the *env_file* configuration pointing to the unversioned *.docker.env* file we created earlier. Now you can manage your environment for Docker through this file, and when you run *docker-compose*, the values will take effect in the container.

Trying out Environment Variables

Let's run the container with the new configuration option and verify that the environment variable has been set (Listing 4.5):

>_ Listing 4.5: Verify the Docker Env File is Working

```
$ docker-compose up -d
$ docker ps # note the app container id
$ docker exec -it 36b079b1aa04 bash
# Inside the running container
root@36b079b1aa04:/var/www/html# echo $HELLO
WORLD

# Shut down the docker containers
$ docker-compose stop
```

You should see "WORLD" printed when you echo the *$HELLO* environment variable inside of the running container. If you change the value of the variable, you should see the value reflected when you restart the container.

Environment variables are an excellent way to make your Docker setup more flexible for applications like Laravel. For example, Laravel uses phpdotenv (https://github.com/vlucas/phpdotenv) to read environment variables through a *.env* file. Using this approach, you can override the values in the *.env* file with system environment variables.

Xdebug Setup

I reach for Xdebug at least once a day, but when I moved my development to Docker, it was a bit tricky getting debugging working consistently.

We will also learn how to configure Xdebug with environment variables, which is compelling for developers wanting to customize the way Xdebug works during runtime. Without changing the Docker image, we can tweak Xdebug's configuration at runtime.

After this section, setting up Xdebug with Docker should be a breeze. Along the way, we'll also ensure that the Docker builds don't have a trace of the Xdebug module when you ship your code to non-development environments.

Installing Xdebug

The first thing we are going to tackle is installing the Xdebug module in our container. Xdebug is a PECL extension, and the official PHP Docker image provides the *pecl* command that we can use to install Xdebug, and then we'll enable it (Listing 4.6):

</> Listing 4.6: Installing and Enabling Xdebug

```
FROM php:7.1-apache

LABEL maintainer="Paul Redmond"
COPY .docker/php/php.ini /usr/local/etc/php/
COPY . /srv/app
COPY .docker/apache/vhost.conf /etc/apache2/sites-available/000-
default.conf
RUN docker-php-ext-install pdo_mysql opcache \
    && pecl install xdebug-2.5.1 \
    && docker-php-ext-enable xdebug \
    && a2enmod rewrite

RUN chown -R www-data:www-data /srv/app
```

After updating the Dockerfile, stop any running containers with *docker-compose stop* and then rebuild the app service with *docker-compose build app*.

After completing the build, run the container to make sure the installation worked (Listing 4.7):

>_ Listing 4.7: Verify that Xdebug was installed and enabled

```
$ docker-compose build app
$ docker-compose up -d
# Note the container id from `docker ps`
$ docker exec -it 00fded44032c bash

# Inside the container...
root@00fded44032c:/var/www/html# php -i | grep ^extension_dir
extension_dir => /usr/local/lib/php/extensions/no-debug-non-zts-
20160303
root@00fded44032c:/var/www/html# ls -la
/usr/local/lib/php/extensions/no-debug-non-zts-20160303 | grep
xdebug
```

```
-rw-r--r-- 1 root staff 1086808 Feb 28 06:03 xdebug.so

# Lists all the lines mentioning xdebug in the ini settings
root@00fded44032c:/var/www/html# php -i | grep xdebug
root@00fded44032c:/var/www/html# php -m | grep xdebug
xdebug
```

You should see the xdebug module listed when you run *php -m* in the container, which means it's working and we are now ready to configure it!

Configuring Xdebug

Installing Xdebug in the container provides an xdebug.ini file containing configuration for Xdebug. To find out where this INI file is located, run *php --ini* inside the container (Listing 4.8):

Listing 4.8: View the xdebug ini file

```
# Still inside the running container
root@00fded44032c:/var/www/html# php --ini
Configuration File (php.ini) Path: /usr/local/etc/php
Loaded Configuration File:         /usr/local/etc/php/php.ini
Scan for additional .ini files in: /usr/local/etc/php/conf.d
Additional .ini files parsed:
/usr/local/etc/php/conf.d/docker-php-ext-opcache.ini,
/usr/local/etc/php/conf.d/docker-php-ext-pdo_mysql.ini,
/usr/local/etc/php/conf.d/docker-php-ext-xdebug.ini

# Output the contents of the xdebug file
root@00fded44032c:/var/www/html# cat
/usr/local/etc/php/conf.d/docker-php-ext-xdebug.ini
zend_extension=/usr/local/lib/php/extensions/no-debug-non-zts-
20160303/xdebug.so
```

The Xdebug file contains one line to enable the extension, and the rest of the Xdebug values are defaults.

We are going to override few default settings to provide some convenience around working with Xdebug, so create a new *.docker/php/xdebug-dev.ini* file for our customizations.

We will start out by adding hard-coded values to the INI file to verify everything works, and then we'll move it to an environment-driven configuration (Listing 4.9):

Listing 4.9: Adding an xdebug ini file

```
[xdebug]
xdebug.default_enable=1
xdebug.remote_autostart=1
; remote_connect_back is not safe in production!
xdebug.remote_connect_back=1
xdebug.remote_port=9001
xdebug.remote_enable=1
xdebug.idekey=DOCKER_XDEBUG
```

I like enabling *remote_autostart* so that Xdebug automatically tries to connect without a GET, POST, or COOKIE variable.

The remote_connect_back setting will try to connect to the client that made the HTTP request. The *remote_connect_back* setting, while not safe in production, is convenient because you don't have to worry about the *remote_host* setting.

For *xdebug.remote_port* I select a non-default remote port (9001) because I have PHP-FPM running locally and I need different port mapping.

Let's drop the created INI settings into the image by copying them in the Dockerfile (Listing 4.10):

Listing 4:10: Copy the Xdebug INI file into the container

```
FROM php:7.1-apache

LABEL maintainer="Paul Redmond"
COPY .docker/php/php.ini /usr/local/etc/php/
COPY . /srv/app
COPY .docker/apache/vhost.conf /etc/apache2/sites-available/000-
default.conf
RUN docker-php-ext-install pdo_mysql opcache \
    && pecl install xdebug-2.5.1 \
    && docker-php-ext-enable xdebug \
    && a2enmod rewrite

COPY .docker/php/xdebug-dev.ini /usr/local/etc/php/conf.d/xdebug-
dev.ini

RUN chown -R www-data:www-data /srv/app
```

With our current configuration file saved, rebuild the container with *docker-compose build app*. Once you finish building the container, you should see your Xdebug settings by adding *phpinfo()* to the top of your *public/index.php* file. You should see the specific settings we changed in our *xdebug-dev.ini* file reflected in the output.

Setting up PhpStorm

With the Xdebug configuration updated, it's time to verify that we can connect to Xdebug with our editor. We are going to use PhpStorm to communicate with Xdebug in this chapter, an excellent commercial IDE for PHP. I prefer PhpStorm's debugging UI and capabilities, but adapting these instructions to any Xdebug client should be relatively straightforward.

The first thing you'll do is open PhpStorm's preferences and navigate to "Languages & Frameworks > PHP > Debug". Because we specified 9001 in our Xdebug configuration, we need to change PhpStorm to use port 9001 as well (Figure 4.1).

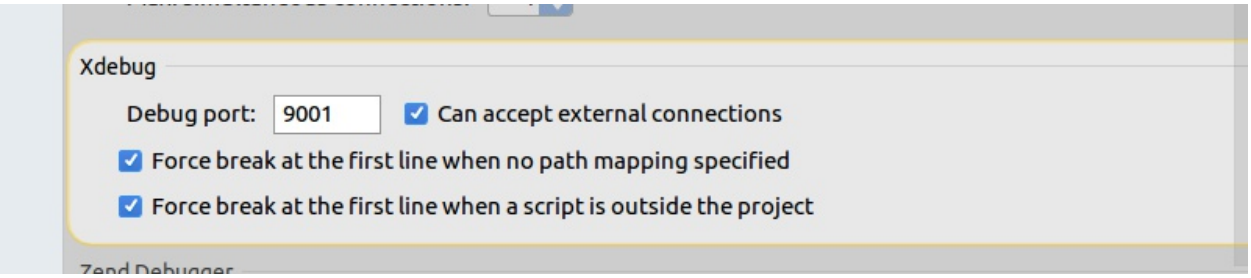


Figure 4.1: Tweaking PhpStorm Xdebug Port

The next step is setting up a run configuration and server. In PhpStorm, click the Xdebug bar (usually in the top right corner) drop-down and click "edit configurations..." (Figure 4.2).

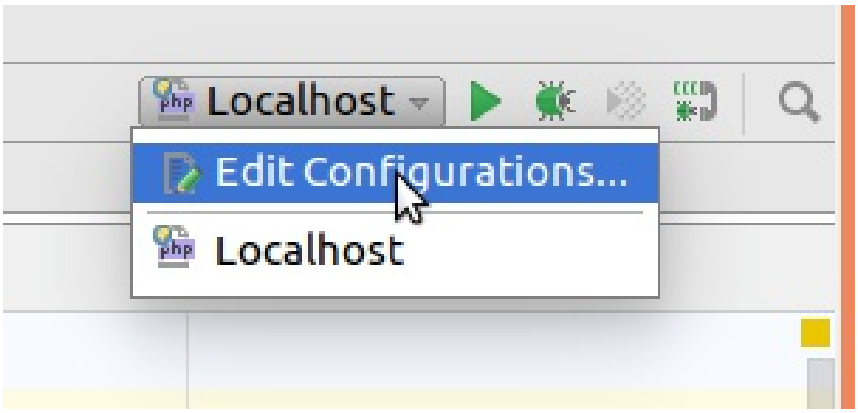


Figure 4.2: Edit run configurations

On the following screen, click the plus (+) button in the top left and select "PHP Web Application" to create a run configuration. Give the application a name (i.e., Localhost), and then click the "..." button to add a server.

On the server screen, make sure and enter port "8080", select Xdebug as the debugger, and map your local project path to the path on the server (*/srv/app*) so Xdebug knows how to map files correctly (Figure 4.3).

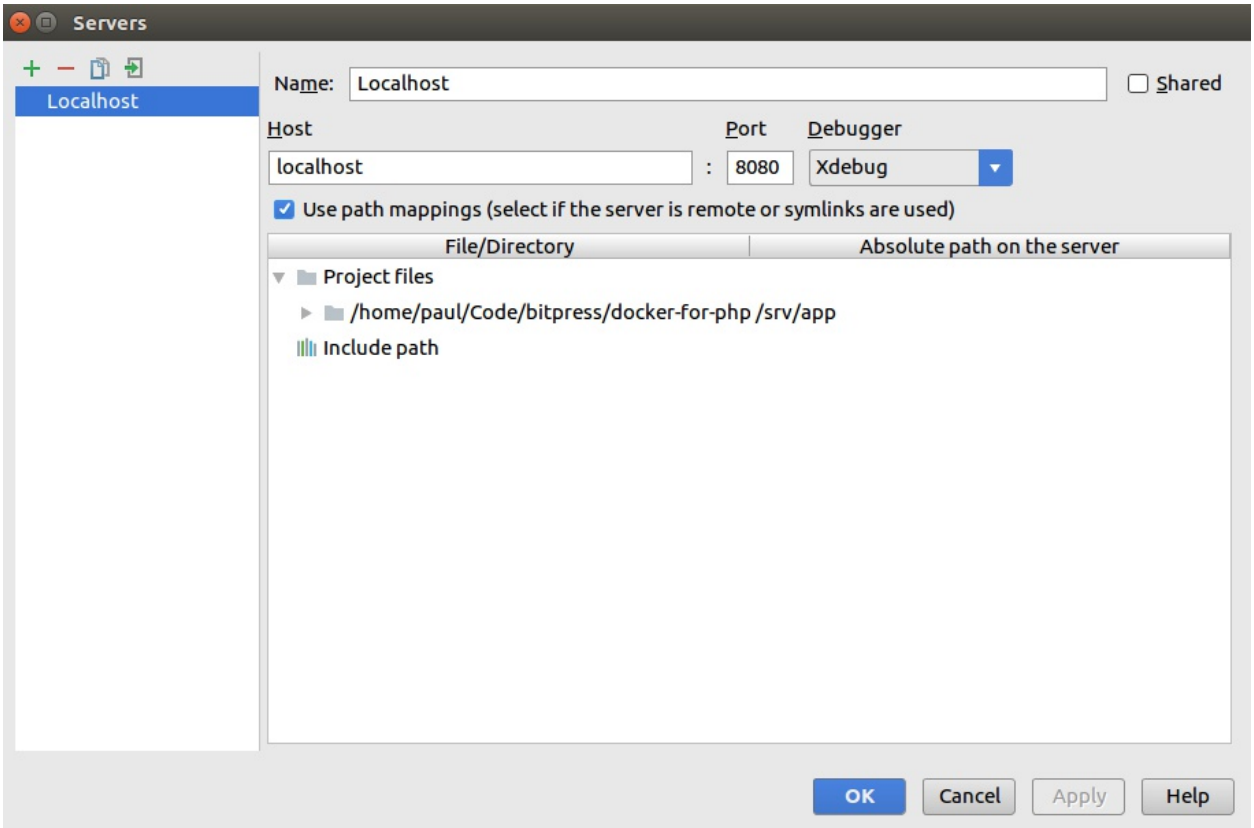


Figure 4.3: Add a server to the run configuration

Once you have configured the web server and run configuration, toggle the "start listening for PHP Debug Connections" icon (Figure 4.4):

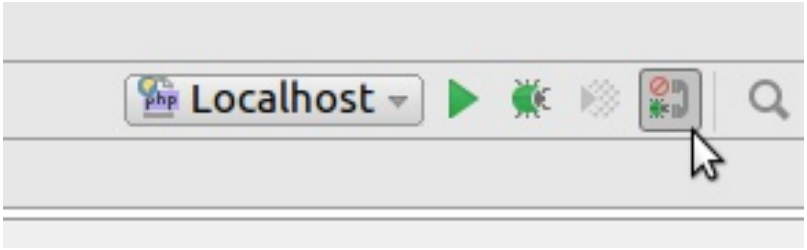


Figure 4.4: Toggle start listening for connections

You should be able to connect to Xdebug by setting a breakpoint or breaking on the first line.

Build The Latest Image

Remember to rebuild the latest container with *docker-compose build*, so your INI settings are copied into the Docker image.

Using Environment to Configure Xdebug

The last part of this section is one of my favorite Docker tricks with PHP. Using environment variables, we can make our INI values dynamic, allowing us to update configuration without rebuilding an image each time you want to change something.

Open the *.docker.env.example* file and replace the contents of the file with the following variables (Listing 4.11).

Listing 4.11: Add Xdebug Environment Variables to *.docker.env.example*

```
# Xdebug
PHP_XDEBUG_DEFAULT_ENABLE=1
PHP_XDEBUG_REMOTE_AUTOSTART=1
PHP_XDEBUG_REMOTE_CONNECT_BACK=1
PHP_XDEBUG_REMOTE_PORT=9001
PHP_XDEBUG_REMOTE_ENABLE=1
PHP_XDEBUG_IDEKEY=DOCKER_XDEBUG
```

Be sure to update your *.docker.env* file with the same values or this won't work!

Variable Naming Convention

You can name the environment variables anything you'd like. I like to prefix my PHP environment variables with *PHP_* and match the INI configuration name by replacing dots (.) with underscores (_). This convention gives me an idea of configuration values at a glance. For example, the environment

variable *PHP_XDEBUG_REMOTE_AUTOSTART* matches the Xdebug INI setting *xdebug.remote_autostart*.

With the variables in place, let's update the *.docker/php/xdebug-dev.ini* file to use them (Listing 4.12):

</> Listing 4.12: Use Environment Variables in the Xdebug INI File

```
[xdebug]
xdebug.default_enable = ${PHP_XDEBUG_DEFAULT_ENABLE}
xdebug.remote_autostart = ${PHP_XDEBUG_REMOTE_AUTOSTART}
; remote_connect_back is not safe in production!
xdebug.remote_connect_back = ${PHP_XDEBUG_REMOTE_CONNECT_BACK}
xdebug.remote_port = ${PHP_XDEBUG_REMOTE_PORT}
xdebug.remote_enable = ${PHP_XDEBUG_REMOTE_ENABLE}
xdebug.idekey = ${PHP_XDEBUG_IDEKEY}
```

PHP INI configuration files can read from the environment by wrapping environment variables in curly brackets (*${}*).

Before we can test out our changes, we need to rebuild the Docker image so that our latest *xdebug-dev.ini* file gets built into the image (Listing 4.13):

>_ Listing 4.13: Rebuild the Application Image

```
$ docker-compose build app
$ docker-compose up -d
```

With the container running, add "*phpinfo(); exit;*" to the top of your project's *public/index.php* file so you can verify your settings in the browser; you should see something like the following (Figure 4.5):

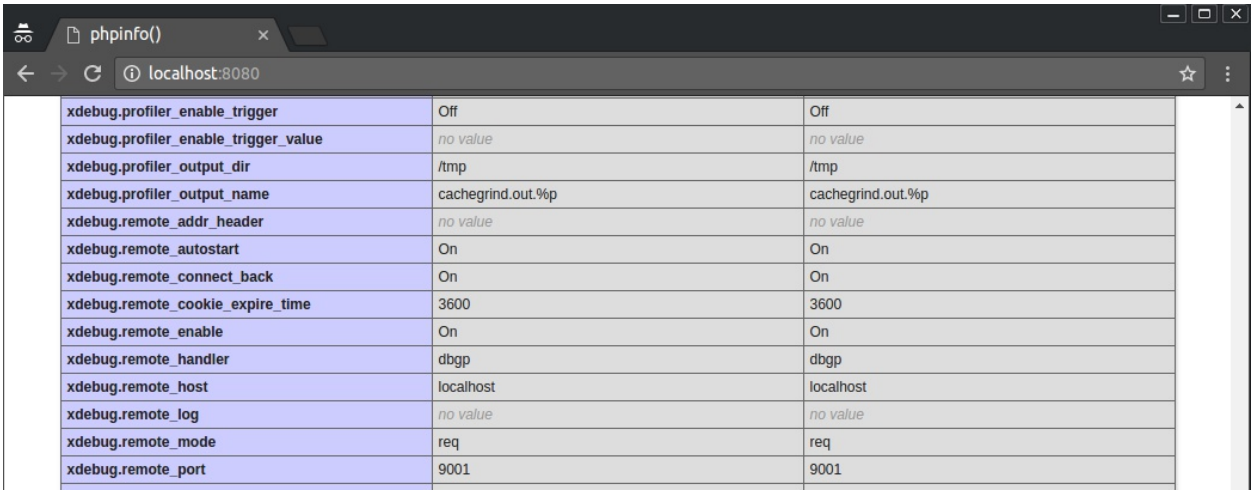


Figure 4.5: Verify Xdebug Settings via Environment

At this point, you should have a working environment configuration, but let's verify that it's working as expected by changing a value and confirming that the value is updated (Listing 4.15):

Listing 4.15: Change Remote Autostart to off in .docker.env

```
PHP_XDEBUG_REMOTE_AUTOSTART=0
```

For environment changes to take effect in a container, you need to restart it with *docker-compose restart*. Once the container is finished rebooting (which should be very quick) you should see that remote autostart is disabled (Figure 4.6).

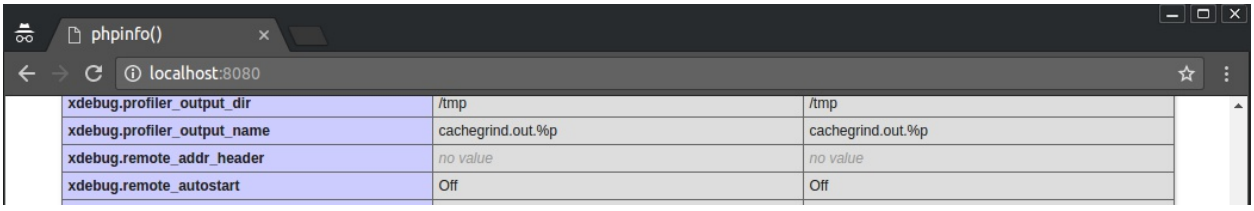


Figure 4.6: Verify Xdebug Remote Autostart Changes

Now you have a flexible Xdebug configuration that each developer can change without needing to rebuild the Docker image. Our setup is pretty sweet if you ask me!

Xdebug Profiling

The Xdebug profiler (https://xdebug.org/docs/profiler) is an excellent way to find performance bottlenecks, analyze your code, and find the most frequently called methods. However, using the profiler with Docker poses a bit of a problem, because profiler results (Cachegrind files) reside in the container.

Before we worry about getting the profiler results locally, let's first see how profiling works and then add a volume to access profiling data locally.

First, let's see the what the configuration looks like by default (Listing 4.16):

Listing 4.16: The Default Xdebug Profiling Settings

```
# Jump into the running app container
$ docker exec -it 902039ff5c41 bash

# Inside the app container
$ php -i | grep xdebug.profiler
xdebug.profiler_aggregate => Off => Off
xdebug.profiler_append => Off => Off
xdebug.profiler_enable => Off => Off
xdebug.profiler_enable_trigger => Off => Off
xdebug.profiler_enable_trigger_value => no value => no value
xdebug.profiler_output_dir => /tmp => /tmp
xdebug.profiler_output_name => cachegrind.out.%p =>
cachegrind.out.%p
```

If you look carefully, the *profiler_output_dir* is */tmp*, and the profiler is not enabled by default.

Let's first work on Getting the Xdebug profiler running by adding profiling configuration to the *.docker/php/xdebug-dev.ini* file that allows us to tweak the directory and enable profiling (Listing 4.17):

</> Listing 4.17: Add Profiler Configuration

```
[xdebug]
xdebug.default_enable = ${PHP_XDEBUG_DEFAULT_ENABLE}
xdebug.remote_autostart = ${PHP_XDEBUG_REMOTE_AUTOSTART}
; remote_connect_back is not safe in production!
xdebug.remote_connect_back = ${PHP_XDEBUG_REMOTE_CONNECT_BACK}
xdebug.remote_port = ${PHP_XDEBUG_REMOTE_PORT}
xdebug.remote_enable = ${PHP_XDEBUG_REMOTE_ENABLE}
xdebug.idekey = ${PHP_XDEBUG_IDEKEY}

; profiling
xdebug.profiler_enable = ${PHP_XDEBUG_PROFILER_ENABLE}
xdebug.profiler_output_dir = ${PHP_XDEBUG_PROFILER_OUTPUT_DIR}
```

You added two profiler values that will match up with two new environment variables you need to add to *.docker.env* and *.docker.env.example* (Listing 4.18):

</> Listing 4.18: Add Profiler Environment Values

```
PHP_XDEBUG_DEFAULT_ENABLE=1
PHP_XDEBUG_REMOTE_AUTOSTART=1
PHP_XDEBUG_REMOTE_CONNECT_BACK=1
PHP_XDEBUG_REMOTE_PORT=9001
PHP_XDEBUG_REMOTE_ENABLE=1
PHP_XDEBUG_IDEKEY=DOCKER_XDEBUG
PHP_XDEBUG_PROFILER_ENABLE=1
PHP_XDEBUG_PROFILER_OUTPUT_DIR=/tmp
```

We've enabled the profiler and configured the output directory. For now, we leave the output directory set as the default /tmp path while we verify that profiling is working in the container.

Save your INI and *.docker.env* changes, and then rebuild the image, so your *xdebug-dev.ini* changes are part of the build (Listing 4.19):

Listing 4.19: Rebuild the Image

```
$ docker-compose down
$ docker-compose rm -v
$ docker-compose build app
$ docker-compose up -d

# Find the container id
$ docker ps

# Jump into the container
$ docker exec -it 8c2d771eacd8 bash

# Now refresh http://localhost:8080
# After refreshing the container, inspect the /tmp folder
$ ls /tmp/
cachegrind.out.10  cachegrind.out.8  pear
```

After refreshing the browser, you will see Cachegrind files in the "/tmp" path.

If you destroy the container, start a new one, and then inspect the */tmp* path your files will be gone. Docker container images are immutable, which means that when containers are destroyed, the modifications are lost.

If you examine the owner of the Cachegrind files, you will notice that the Apache user (www-data) owns the Cachegrind report. File ownership will be critical in a moment when we move the profiler output path to our volume for local consumption.

Our next step is making the profiler data available locally by updating the output directory (Listing 4.20):

Listing 4.20: Configure the Profiler Output Path

```
PHP_XDEBUG_DEFAULT_ENABLE=1
PHP_XDEBUG_REMOTE_AUTOSTART=1
```

```
PHP_XDEBUG_REMOTE_CONNECT_BACK=1
PHP_XDEBUG_REMOTE_PORT=9001
PHP_XDEBUG_REMOTE_ENABLE=1
PHP_XDEBUG_IDEKEY=DOCKER_XDEBUG
PHP_XDEBUG_PROFILER_ENABLE=1
PHP_XDEBUG_PROFILER_OUTPUT_DIR=/srv/app/storage/logs
```

On the last line, we point the output directory to our application's storage path which has an attached volume. Since Cachegrind files are within the mounted volume path, this means we can access cachegrind files locally right?

Well, maybe.

Depending on your setup, you might have a permissions issue. I am running the examples on Ubuntu 16.04 LTS, and at this point, I cannot see Cachegrind files after starting a new container and refreshing the browser.

With the new profile path when I refresh my browser and use *ls -la /srv/app/storage/logs* I don't see any cachegrind files (Listing 4.21):

>_ Listing 4.21: Directory Permissions

```
# Inside a running container
$ php -i | grep xdebug.profiler_output_dir
$ ls -la /srv/app/storage/logs/
total 12
drwxrwxr-x 2 1000 1000 4096 Mar  9 05:47 .
drwxrwxr-x 5 1000 1000 4096 Mar  9 05:47 ..
-rwxrwxr-x 1 1000 1000   14 Mar  9 05:47 .gitignore
-rwxrwxr-x 1 1000 1000    0 Mar  9 05:47 lumen.log

# The www-data user cannot write in the logs folder
$ id -u www-data
33
```

The volume might prevent the Apache user from saving Cachegrind files on your setup depending on permissions. If you do not see cachegrind files, run the following command for any folder that needs to be writable by the web server (Listing 4.22):

>_ Listing 4.22: Fix Volume Permissions Issues

```
# Allow "others" read and write access
$ chmod -R o+rw bootstrap/ storage/

# Refresh the browser, you should see Cachegrind files locally
$ ls storage/logs
cachegrind.out.8  lumen.log
```

In the case of Lumen, we need to recursively make sure that "others" can write to the defined *storage/* path to sort out permission issues caused by volumes.

Allowing this level or read/write access **is only needed for local development**! When running the container without a volume (i.e., production), the *www-data* user owns the application files and directories are not world-writable.

The permission changes in Listing 4.22 should allow the container *www-data* user to write to the *storage/* paths on your local machine as needed. If you are running Docker locally on Linux, the Cachegrind files will not be owned by your $USER but should be readable. You can also remove them with *rm -f storage/logs/cachegrind**.

Equipped with local Cachegrind files, you can now use PhpStorm (or the tool of your choice) to visualize profiler data (Figure 4.7):

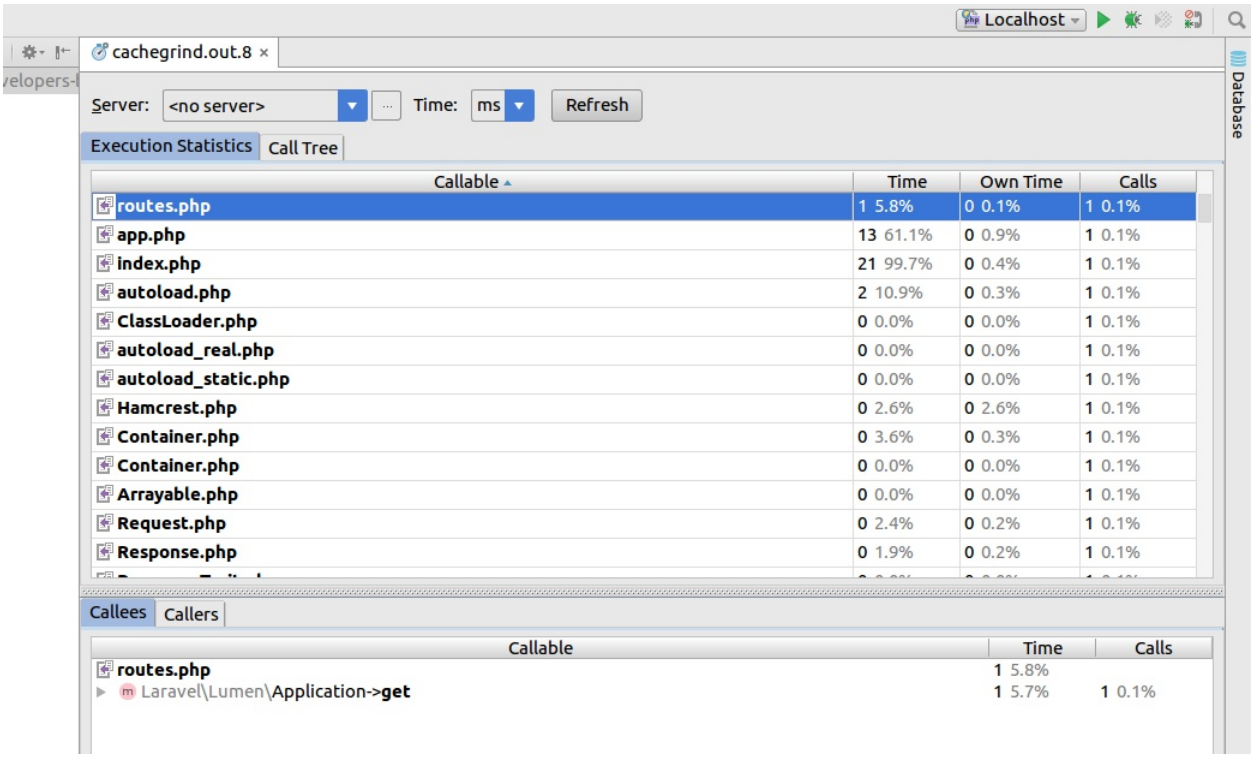


Figure 4.7: Cachegrind Output Example

Making Xdebug Disappear

When I first learned about Docker, one of the neatest selling points to me was the claim that "Docker runs the same everywhere" and provides "more repeatable environments." These claims are probably the closest thing to true as can be expected, but in the case of Xdebug, you don't want any trace of Xdebug installed on production. Even if Xdebug is a registered module and disabled, it has proven to cause overhead.

So far, the Dockerfile installs the Xdebug module, and we have no way of controlling the build to only include Xdebug in development. We need a way to manage the configuration of Xdebug at runtime.

How might we go about stopping that from happening in production-like environments but not development?

While we can't conditionally control the installation of Xdebug during a build, there are a few ways we can ensure that only development environments have the Xdebug module enabled in PHP.

One way of disabling Xdebug is using a custom bash script to start the container and remove the Xdebug INI configuration before starting the web server. We are not ready to talk about custom commands yet (we cover them in Chapter 8), so we will look at a different way using the skills we've learned so far.

Later in the book, you can figure out how to remove Xdebug with a custom bash command, which is probably the best way to remove Xdebug at runtime.

Enable Xdebug with Environment

The first way we will solve this is a slight trick in the *Dockerfile* coupled with customizing the *PHP_INI_SCAN_DIR* environment variable. We will have a set of INI files for development and another set for non-development.

First, we are going to modify our *Dockerfile* to provide a development-specific configuration path for development that overrides the default location PHP looks for INI files (Listing 4.23):

Listing 4.23: Provide a Development-Specific Scan Dir Path

```
FROM php:7.1-apache

LABEL maintainer="Paul Redmond"

COPY .docker/php/php.ini /usr/local/etc/php/
COPY . /srv/app
COPY .docker/apache/vhost.conf /etc/apache2/sites-available/000-
default.conf
RUN docker-php-ext-install pdo_mysql opcache \
    && pecl install xdebug-2.5.1 \
    && docker-php-ext-enable xdebug \
    && a2enmod rewrite

COPY .docker/php/xdebug-dev.ini /usr/local/etc/php/conf.d/xdebug-
dev.ini
```

```
RUN cp -R /usr/local/etc/php/conf.d \
    /usr/local/etc/php/conf.d-dev \
    && rm -f /usr/local/etc/php/conf.d/*-dev.ini \
    && rm -f /usr/local/etc/php/conf.d/*xdebug.ini

RUN chown -R www-data:www-data /srv/app
```

We've copied the contents of the scan directory (/usr/local/etc/php/conf.d) into a separate development path. The next line removes any INI files that end in *-dev.ini* and also specifically the *xdebug.ini* file when we enable Xdebug with *docker-php-ext-enable xdebug*.

You could consider this approach a hack (it wouldn't hurt my feelings), and you **should be careful** about copying *.ini* files in a later part of the Dockerfile.

I wouldn't say this is the cleanest approach to solving this issue, but it works okay in my opinion. The more appropriate way to solve this would be through a custom CMD script that checks for which environment Docker is running and enables or disables Xdebug, but we're not ready to go over the CMD instruction yet.

After you rebuild the image and jump into a running container, you can see that the Xdebug module is now gone (Listing 4.24):

Listing 4.24: The Xdebug Module is Gone

```
$ php -m | grep xdebug
# Nothing...

$ export PHP_INI_SCAN_DIR=/usr/local/etc/php/conf.d-dev
$ php -m | grep xdebug
xdebug
```

The next step is changing your *.docker.env* (and *.docker.env.example*) file to use the development scan dir (Listing 4.25):

</> Listing 4.25: Configure the Development INI Scan Directory

```
PHP_XDEBUG_DEFAULT_ENABLE=1
PHP_XDEBUG_REMOTE_AUTOSTART=1
PHP_XDEBUG_REMOTE_CONNECT_BACK=1
PHP_XDEBUG_REMOTE_PORT=9001
PHP_XDEBUG_REMOTE_ENABLE=1
PHP_XDEBUG_IDEKEY=DOCKER_XDEBUG
PHP_XDEBUG_PROFILER_ENABLE=1
PHP_XDEBUG_PROFILER_OUTPUT_DIR=/srv/app/storage/logs

PHP_INI_SCAN_DIR=/usr/local/etc/php/conf.d-dev
```

The configuration file will now take care of setting the *PHP_INI_SCAN_DIR* environment variable. In production environments, the *PHP_INI_SCAN_DIR* will be the default.

After you restart the container, you should have the Xdebug module enabled (Listing 4.26):

>_ Listing 4.26: Checking our Scan Dir Environment Setting

```
$ docker-compose down
$ docker-compose build
$ docker-compose up -d
$ docker ps
# Replace 8552cd067c83 with your container id
$ docker exec -it 8552cd067c83 bash

# In the container
$ echo $PHP_INI_SCAN_DIR
/usr/local/etc/php/conf.d-dev
$ php -m | grep xdebug
xdebug
```

Our last step in this section is to create a "production" docker-compose file which will help demonstrate how our container will run without volumes.

The *docker-compose.prod.yml* file lives in the root of the project is just like the *docker-compose.yml* file, minus the volume (Listing 4.27):

</> Listing 4.27: A Production-Like Compose File

```
version: "3"
services:
  app:
    build: .
    depends_on:
      - mariadb
    ports:
      - "8080:80"
    links:
      - mariadb:mariadb
    environment:
      DB_CONNECTION: mysql
      DB_HOST: mariadb
      DB_PORT: 3306
      DB_DATABASE: dockerphp
      DB_USERNAME: root
      DB_PASSWORD: password
  mariadb:
    image: mariadb:10.1.21
    ports:
      - "13306:3306"
    environment:
      - MYSQL_DATABASE=dockerphp
      - MYSQL_ROOT_PASSWORD=password
```

The production version doesn't have the *env_file* setting, which means we are no longer using the *.docker.env* file to configure the environment.

We provide some hard-coded environment variables to override the values set in the *.env* file, but this file is just to simulate production. Make sure that you are not hard-coding your environment variables in the docker-compose file, but defining

development-specific values isn't a big deal.

Before we build and run containers with this file, update your *public/index.php* file with *"phpinfo(); exit;*" at the top so you can verify that Xdebug is not present.

Skipping the volume in the Docker compose file will demonstrate that you have to rebuild the image to pick up code changes.

Let's try out our new Compose file and verify that Xdebug registered (Listing 4.28):

Listing 4.28: Run Containers with docker-compose.prod.yml

```
$ docker-compose down

# Build so your phpinfo() call gets copied into the container
$ docker-compose --file=docker-compose.prod.yml build
$ docker-compose --file=docker-compose.prod.yml up -d
```

Now that your container is running you will not find Xdebug enabled. Also, the PHP scan directory is set to the default */usr/local/etc/php/conf.d* path, and that the *xdebug.ini* files are gone.

By default, Docker Compose looks for a *docker-compose.yml* file, but by using the *docker-compose --file* flag, you can use an alternate Docker Compose file.

While using the *docker-compose.prod.yml* file, you can no longer modify your project files and have them instantaneously update in the container via volumes. Try it. Your changes will not take effect until you stop the container, rebuild the image, and then start a new container while you are using the prod file.

It's imperative that you understand the immutable nature of containers. While we need the convenience of mounted volumes in development, the build shouldn't rely on a mounted volume. Because you cannot rely on local files (think saving file uploads and

serving them) inside the container, I also use an external storage service like S3 for uploads.

When a host machine (your development machine) mounts a host directory (volume) on an existing path in the container (*/srv/app* in this case), it overrides the existing files in that path.

According to the Docker Documentation tutorial, Manage Data in Containers (https://docs.docker.com/engine/tutorials/dockervolumes/#mount-a-host-directory-as-a-data-volume), "the mount overlays but does not remove the pre-existing content. Once the mount is removed, the content is accessible again."

De-Debugged

This chapter got into the nitty-gritty of using and configuring Xdebug in containers. Along the way, you learned a bunch about environment configuration, volumes, permissions, and running Docker compose with alternate file names. You should be getting more comfortable building, starting, stopping, and running bash inside your containers at this point.

The next chapter will shift focus to a somewhat tricky topic of using PHP Composer with Docker and installing private composer packages.

Chapter 5: Using Composer with Docker

In this chapter we explore installing Composer (https://getcomposer.org/) within a Docker image, working with private repositories, and strategies to build your projects with Composer and Docker.

It may or may not be apparent to you if you should install your Composer dependencies before or during a build. Up to this point, we've been installing composer dependencies locally and then copying them into the image during a build.

When you start installing composer dependencies as part of your Dockerfile builds, it might become an annoyance to rebuild your docker images with each new composer change. Composer is a relatively quick operation, but as you'll shortly see, building the image over again can become time-consuming.

So should you manage your composer dependencies outside of Composer? The short answer is that both approaches have tradeoffs.

One could make the argument that the *composer.json* file provides constraints for the application code, PHP modules, and even PHP version; therefore, the dependencies should be installed in the Dockerfile while running *docker build* every time.

One can also make a case for installing Composer dependencies outside of Docker to make development quicker and more convenient, treating composer as a pre-build step that happens before the docker build process on a CI server, which copies the results into the image during *docker build*.

Project Setup

We are going to create a new project using the Laravel Framework (https://laravel.com/) as we work through this chapter. The principles are the same for any Composer project we use with Docker, not just Laravel.

Let's start by creating the project files (Listing 5.1):

>_ Listing 5.1: Create the Project Files

```
$ mkdir -p ~/Code/composer-in-docker
$ cd $_
$ touch Dockerfile composer-installer.sh

# Create a Composer project in an "app" subfolder
$ composer create-project --prefer-dist laravel/laravel:5.5 app/
```

This project is organized a little differently than the last chapter so you can see different project organization styles. In this chapter, the Dockerfile is at the root of the project, and our application is in an *app/* folder.

There's no "wrong" way to organize Docker in your projects, but showing you a few different styles gives you some perspective. If you recall in Chapter 4, we embedded the Docker-specific files in a *.docker/* folder and we return to that format in future chapters. I prefer to organize Docker around my code, not the other way around.

Installing Composer in Docker

The first thing we work on is installing the Composer executable in a Docker image and making it executable. We can then run composer commands during a *docker build* and work with composer inside of a running container. Your local project has a *vendor/* folder when you install the project, and the COPY command copies this folder during a build, so I'll introduce you to ignoring this folder in Docker to ensure a pristine build. There are a few caveats around COPY that you learn about along the way too.

Adding Composer to the Dockerfile

Let's work on the first part: installing Composer inside of a Docker image (Listing 5.2):

</> Listing 5.2: Install the Composer Executable

```
FROM php:7.1-apache

RUN curl -sS https://getcomposer.org/installer \
    | php -- --install-dir=/usr/local/bin --filename=composer \
    && chmod +x /usr/local/bin/composer
```

The RUN command downloads the Composer installer and installs the executable at */usr/local/bin/composer*. However, this technique doesn't actually verify the Composer installation. According to Composer's documentation, here's how we can install Composer programatically (https://getcomposer.org/doc/faqs/how-to-install-composer-programmatically.md) as recommended. Add the following to *composer-installer.sh* (Listing 5.3):

</> Listing 5.3: The programatic composer installer script

```
#!/bin/sh

EXPECTED_SIGNATURE=$(curl -s
https://composer.github.io/installer.sig)
php -r "copy('https://getcomposer.org/installer', 'composer-
setup.php');"
ACTUAL_SIGNATURE=$(php -r "echo hash_file('SHA384', 'composer-
setup.php');")

if [ "$EXPECTED_SIGNATURE" != "$ACTUAL_SIGNATURE" ]
then
    >&2 echo 'ERROR: Invalid installer signature'
    rm composer-setup.php
    exit 1
fi
```

```
php composer-setup.php --quiet
RESULT=$?
rm composer-setup.php
exit $RESULT
```

We slightly modified the script from the documentation, using *cURL* instead of *wget* because cURL is already available. Using the *-s* flag we make the curl command silent with no output. Other than this line, the rest of the script is identical to the official documentation.

Here's how we can use the installer script to install composer and put it in the path (Listing 5.4):

>_ Listing 5.4: Verify and Install Composer

```
FROM php:7.1-apache

COPY composer-installer.sh /usr/local/bin/composer-installer

# Install composer
RUN chmod +x /usr/local/bin/composer-installer \
    && composer-installer \
    && mv composer.phar /usr/local/bin/composer \
    && chmod +x /usr/local/bin/composer \
    && composer --version
```

This technique copies an installer script we version locally and uses it to verify and install composer. Once the script creates the *composer.phar* file, we move it to */usr/local/bin* and make it executable. Last, we output the version which verifies that composer works.

For good measure, here's another technique that doesn't use an external script to install composer, but still verifies the signature:

```
FROM php:7.1-apache

RUN curl -o /tmp/composer-setup.php
https://getcomposer.org/installer \
    && curl -o /tmp/composer-setup.sig
https://composer.github.io/installer.sig \
    && php -r "if (hash('SHA384',
file_get_contents('/tmp/composer-setup.php')) !==
trim(file_get_contents('/tmp/composer-setup.sig'))) {
unlink('/tmp/composer-setup.php'); echo 'Invalid installer' .
PHP_EOL; exit(1); }" \
    && php /tmp/composer-setup.php \
        --no-ansi \
        --install-dir=/usr/local/bin \
        --filename=composer \
        --snapshot \
    && rm -f /tmp/composer-setup.*
```

The last technique comes from a Stack Overflow answer (https://stackoverflow.com/a/42147748) using cURL and PHP without an external script. I prefer the script provided from Composer, but you have plenty of options.

Regardless of which method you use, let's build the image and check that Composer is installed correctly (Listing 5.5):

Listing 5.5: Verify the Composer executable

```
$ docker build -t ch05-composer .
$ docker run --rm ch05-composer /usr/local/bin/composer --version
Composer version 1.5.2 2017-09-11 16:59:25
```

Although we are using a container that runs Apache by default, we passed the *docker run* command an argument—the path to the Composer executable—which runs the command in a new container. The *--rm* flag automatically removes the container on exit, which means it does not show up when you run *docker ps -a*.

Installing Composer Dependencies

Now that we have the Composer executable in the image, let's copy our application and install Composer dependencies during a build (Listing 5.6):

Listing 5.6: Install Composer Dependencies

```
FROM php:7.1-apache

COPY composer-installer.sh /usr/local/bin/composer-installer

# Install composer
RUN apt-get -yqq update \
    && apt-get -yqq install --no-install-recommends unzip \
    && chmod +x /usr/local/bin/composer-installer \
    && composer-installer \
    && mv composer.phar /usr/local/bin/composer \
    && chmod +x /usr/local/bin/composer \
    && composer --version

# Add the project
ADD app /var/www/html

WORKDIR /var/www/html

RUN composer install \
    --no-interaction \
    --no-plugins \
    --no-scripts \
    --prefer-dist
```

We installed the unzip package so that Composer can download and unzip package dependencies. Next, we add the *app/* folder and set the *WORKDIR* to our application root path.

The next new instruction is the *ADD app /var/www/html*, which adds files inside the root

app/ folder (the location of our application locally) into the image at */var/www/html*.

The *WORKDIR* instruction sets the working directory for subsequent Docker commands like *RUN*, *CMD*, *ENTRYPOINT*, *COPY*, and *ADD* Docker instructions. You can use *WORKDIR* multiple times, in our case we are using it and then the next *RUN* command relatively runs *composer install* in the */var/www/html* directory.

At this point, any small changes to your source code require the Docker build to run the *composer install* step without cache, which is very slow. We revisit this later in the chapter and learn how to cache Composer dependencies in the image to speed up builds.

Ignoring Local Vendor Files

If you run a Docker build, you might be surprised how quickly "composer install" runs during a build. It should run much slower from scratch, but you instead see "Nothing to install or update" when you run *docker build*. The mystery behind the quick builds is the fact that we are copying all the local vendor files in from the initial *composer create-project* command into the image before running *composer install*.

To get a clean Composer installation, we need to ignore the local *vendor/* folder during a build using a .dockerignore file. The *.dockerignore* file resides in the same directory as the *Dockerfile* file and works similarly to a .gitignore file, ensuring that unintended files are left out during a *COPY* or *ADD* instruction (Listing 5.7):

Listing 5.7: Ignore the vendor folder

```
$ echo "app/vendor/" >> .dockerignore
# Re-run the build with no cache
$ docker build --no-cache -t ch05-composer .

# After the build, verify the files inside a container
$ docker run -it --rm ch05-composer bash
root@9cc40128b59f:/var/www/html# ls -la vendor/
```

We added the *app/vendor/* path to the *.dockerignore* file, which means Docker ignores this path when *ADD app /var/www/html* runs. Without the vendor folder copied into the image, Composer installs everything from scratch. This nuance of copying files from your local machine (or a build server) is hard to sometimes catch with Docker; pay attention to which files get copied into an image.

Installing Private Composer Packages in Docker

When you try to move to Docker, you eventually run into the problem of providing proper credentials while installing private Git repositories in your projects. There are a couple of ways you could go about solving permissions issues, but all have security concerns:

1. Copy an SSH key into the Docker image from a build machine
2. Install Composer dependencies on a credentialed machine and then copy the vendor/ folder into the image during a build
3. Use an OAuth token with a Composer config file during a *docker build*

I hesitate to share the first one, because of the substantial security issues of embedding SSH access into an image. However, I have seen this technique used in the wild, so I bring it up for awareness.

For argument's sake, if you put aside the significant security risks, the first option does work in the technical sense. However, copying an SSH key into your image at build time is cumbersome at best. Each developer would require access to this SSH key for local development builds. If you ever need to update or revoke the SSH key, everyone would need to get a new copy of the private key.

I have found the best way to deal with this situation, due to the security risks and lack of built-time secrets, is the second option: install composer dependencies on a credentialed build server and then copy the *vendor/* folder into the Docker image.

The landscape for using build-time secrets is still a work-in-progress (see https://github.com/moby/moby/issues/13490) at the time of writing. I imagine that soon build-time secrets make it possible to pass in sensitive credentials that you can use to produce an *auth.json* file in the image, install composer, and remove the *auth.json* file in the same layer, all without a trace of your secret key.

In the meantime, if you are adamant about installing private Composer dependencies during a *docker build* I think the third option is the best, so we cover it in this text, and you should be able to adapt these techniques as Docker improves the ability to handle secrets at build time.

Here are the steps we take:

1. Create a private git project on Bitbucket.org
2. Create a read-only OAuth consumer on Bitbucket.org
3. Generate a unversioned *auth.json* config in our project
4. Define the private Composer dependency as a required package
5. Run our existing *docker build* command

Bitbucket offers free private repositories at the time of this writing, so we use Bitbucket in our example, but you can also use Github and others. You need to sign up (https://bitbucket.org/account/signup/) for a Bitbucket.org account if you want to follow along.

After logging in, go to "Bitbucket Settings > OAuth" and then click "Add a Consumer." Fill out a name and description, and lastly **you must enter a callback URL for Composer to work with this key**! I fill out something like *http://example.com*, but it can be anything. Composer won't use this callback, but it's mandatory for things to work right. The particular grant flow that Composer utilizes does not use the callback.

Bitbucket OAuth Help

Bitbucket provides documentation for setting up OAuth on Bitbucket Cloud if you need assistance (https://goo.gl/jZqgkq).

Once you set up BitBucket, you need to create a private repository and then clone a local copy on your machine so we can set up the private package for consumption by our Docker builds.

Setting up the Private Package

Once you create the project in version control, add a *composer.json* file to the root of the project (Listing 5.8):

Listing 5.8: Private composer.json File

```
{
    "name": "bitpressio/docker-private-package",
    "description": "An example private composer package",
    "license": "MIT",
    "authors": [
        {
            "name": "Paul Redmond"
        }
    ],
    "require": {},
    "autoload": {
        "psr-4": {
            "Bitpress\\DockerPrivatePackage\\": "src/"
        }
    },
    "extra": {
        "branch-alias": {
            "dev-master": "1.0.x-dev"
```

```
        }
    }
}
```

Change the Composer *name* property to match your Bitbucket repository and modify the namespace to match whatever you want to use. The *autoload* section autoloads the package's namespace, and finally the *extra* key contains a branch alias so we can install out dependency as *1.0.x-dev* from the *master* branch.

Next, create the file in *src/Example.php* but **update the namespace** to match the namespace you configured in the *composer.json* file (Listing 5.9):

</> Listing 5.9: The Example Class in our Private Package

```
<?php

namespace Bitpress\DockerPrivatePackage;

/**
 * An example class demonstrating installing a private
 * Composer package in Docker
 */
class Example {}
```

If you are following along, commit your files and push them to your private Bitbucket repository.

Authenticating the Private Package in Composer

We need to provide a way to authenticate our private Bitbucket repository using the OAuth consumer we created at the beginning of this section.

The Composer documentation references how a bitbucket-oauth configuration key (https://getcomposer.org/doc/06-config.md#bitbucket-oauth) should be defined.

This file lives in the root of your Composer project and Composer looks in that location (among others) for an *auth.json* file.

We don't want to store private credentials in the repository, so we build a template version of the *auth.json* file that can be distributed to all environments easily. Create a file at *app/auth.dist.json* with the following contents (Listing 5.10):

Listing 5.10: Composer auth.dist.json Template

```
{
    "bitbucket-oauth": {
        "bitbucket.org": {
            "consumer-key": "%consumer-key%",
            "consumer-secret": "%consumer-secret%"
        }
    }
}
```

The file has two tokenized values that we replace with a bash script so that developers and build servers can keep secrets out of the repository.

To replace the tokenized values, let's create *auth-setup.sh* in the root of the project alongside the Dockerfile and add the following (Listing 5.11):

</> Listing 5.11: Bash Script to Automate Generating auth.json

```
#!/usr/bin/env bash

[ -e app/auth.json ] && echo "auth.json already exists.
Skipping." && exit 0;

if [ ! -z "$BITBUCKET_CONSUMER_KEY" ] && [ ! -z
"$BITBUCKET_CONSUMER_SECRET" ]; then
    cp app/auth.dist.json app/auth.json
    sed -i '' -e "s/%consumer-key%/$BITBUCKET_CONSUMER_KEY/"
app/auth.json
    sed -i '' -e "s/%consumer-
secret%/$BITBUCKET_CONSUMER_SECRET/" app/auth.json
    echo "Created the auth.json file"
    exit 0
fi

echo "You need to set '\$BITBUCKET_CONSUMER_KEY' and
'\$BITBUCKET_CONSUMER_SECRET' environment variables!";
exit 1;
```

The *auth-setup.sh* file checks for two environment variables and exit with a warning if they don't exist. The script copies the *auth.dist.json* file to *auth.json* and replaces the tokens with the Bitbucket key and secret.

Remember to make the file executable so you can run it (Listing 5.12):

>_ Listing 5.12: Create the auth.json file

```
$ chmod u+x auth-setup.sh

# Replace key and secret with actual values
$ export BITBUCKET_CONSUMER_KEY=key
$ export BITBUCKET_CONSUMER_SECRET=secret
```

```
# Apply the key/secret to app/auth.json
$ ./auth-setup.sh
Created the auth.json file
```

Be careful not to commit *auth.json* into version control–versioning it would defeat the whole purpose (Listing 5.13):

>_ Listing 5.13: Ignore auth.json

```
$ echo "auth.json" >> app/.gitignore
```

The next step is adding your private repository to composer, including additional configuration for the location of the VCS repository in the *repositories* key (Listing 5.14):

</> Listing 5.14: Add Your Private Repository to composer.json (partial file)

```
{
    "require": {
        "php": ">=7.0.0",
        "fideloper/proxy": "~3.3",
        "laravel/framework": "5.5.*",
        "laravel/tinker": "~1.0",
        "bitpressio/docker-private-package": "1.0.x-dev"
    },
    "repositories": [
        {
            "type": "vcs",
            "url":  "https://bitbucket.org/bitpressio/docker-
private-package.git"
        }
    ]
}
```

If you recall earlier, you used *branch-alias* in your private repository's *composer.json* to alias the master branch to version *1.0.x-dev*.

Even though you are running Composer while building a Docker image, you need to update your Composer dependencies locally, and version control the *composer.lock* file. Composer uses the lock file to install the desired set of packages during the *docker build* command (Listing 5.15):

Listing 5.15: Adding the Private Repository to the Composer lock file

```
# Locally
$ cd app/
$ composer update bitpressio/docker-private-package
Package operations: 1 install, 0 updates, 0 removals
  - Installing bitpressio/docker-private-package (dev-master
4d9485b)
# ...
```

Building with a Private Package

We are ready to build a private package defined in our Composer dependencies. Make sure the *auth.json* file we created earlier exists with valid credentials, and run a build from the root of the project (Listing 5.16):

Listing 5.15: Adding the Private Repository to the Composer lock file

```
$ docker build -t ch05-composer .

$ docker run --rm -it ch05-composer /bin/bash

# Inside the docker image
$ ls -la vendor/bitpressio/docker-private-package
drwxr-xr-x 3 root root 4096 Apr 15 07:48 .
```

```
drwxr-xr-x 3 root root 4096 Apr 15 07:48 ..
-rw-r--r-- 1 root root    8 Apr 11 07:04 .gitignore
-rw-r--r-- 1 root root  435 Apr 11 07:04 composer.json
drwxr-xr-x 2 root root 4096 Apr 11 07:04 src
```

You can also run the *ls* command locally instead:

```
$ docker run --rm -it ch05-composer \
    ls -la /var/www/html/vendor/bitpressio/docker-private-package
```

Installing the private package worked, because the *auth.json* file gets copied into the image during builds, and Composer checks for the existence of the file to validate credentials.

We could remove the auth file from the image with a subsequent RUN instruction, but the file is still present in the ADD layer where we added the source code and technically leaked. The file appears removed in the resulting image, but it is still part of the fabric of the image.

Until build-time secrets progress, I prefer to build my composer dependencies outside of the *Dockerfile*, copy them in during a build, and then to verify the image.

Caching Composer Dependencies for Faster Builds

In the last section, we installed Composer dependencies as part of our *Dockerfile* build process. One problem so far is that if you make changes to your code and then run *docker build*, the *composer install* step needs to install dependencies from scratch.

If your project hasn't changed since your last build, each layer is cached and builds quickly. However, if you modify a project file, the ADD *app /var/www/html* step needs to be updated. As a result, all the proceeding layers need to be updated, including "composer install."

Composer dependencies don't change much day-to-day in stable projects, so we can adjust a few things to gain some caching benefits.

The basic idea is that we copy the *composer.json* and *composer.lock* files into the image to cache dependencies in a layer above the ADD *app /var/www/html* step that needs to run more often (Listing 5.21):

Listing 5.21: Caching Composer Dependencies

```
FROM php:7.1-apache

COPY composer-installer.sh /usr/local/bin/composer-installer

# Install composer
RUN apt-get -yqq update \
    && apt-get -yqq install --no-install-recommends unzip \
    && chmod +x /usr/local/bin/composer-installer \
    && composer-installer \
    && mv composer.phar /usr/local/bin/composer \
    && chmod +x /usr/local/bin/composer \
    && composer --version

# Cache Composer dependencies
WORKDIR /tmp
ADD app/composer.json app/composer.lock app/auth.json /tmp/
RUN mkdir -p database/seeds \
    mkdir -p database/factories \
    && composer install \
        --no-interaction \
        --no-plugins \
        --no-scripts \
        --prefer-dist \
    && rm -rf composer.json composer.lock auth.json \
        database/ vendor/
```

```
# Add the project
ADD app /var/www/html

WORKDIR /var/www/html

RUN composer install \
    --no-interaction \
    --no-plugins \
    --no-scripts \
    --prefer-dist
```

The new block of code in **Listing 5.21** copies the *composer.** and *auth.dist.json* files into the temporary folder (/tmp/) with the ADD instruction. When using ADD with more than one source file, the destination (/tmp/) **must be a directory and end with a forward slash**.

Additionally, we need the *database/* folder so the installation doesn't fail; the *database/seeds* and *database/factories* paths are defined in the Composer *autoload* and must exist, but we don't need to add the actual files quite yet.

Later in the build, we run *composer install* again to set up the project correctly, including the autoloaded files in the *database/* path. The goal of the caching step is to prime the cache with vendor packages, making the second *composer install* very fast because it uses cached versions of the repositories that we just installed every time. When the *auth.json*, *composer.json*, or *composer.lock* file changes, the cache layer is rebuilt along with all subsequent layers.

The final Docker instruction in the cache step removes the files installed in the */tmp/vendor/* folder. We don't need them anymore because the vendor files get copied from Composer's cache during the second *composer install*. The cached vendor files remain in the layer (*~/.composer/cache*), so we can use them later.

If you run *docker build* a couple of times you should see your dependencies installed from the cache and *composer install* should be relatively quick. Modify a few files, see how the build changes, and which steps are cached based on your changes.

Caching reduces builds for unnecessary steps that haven't changed; however, some people prefer to have a clean build without any Composer cache in their CI environment. If you want to skip caching, you can create a fresh build every time by running *docker build --no-cache*, which makes the entire Docker image from scratch.

Running Composer Locally

Another way you can develop applications with Docker is to run Composer (and other development commands) locally. You might run into issues with required PHP versions and modules not being installed on your machine, however, if you install modern PHP7 tools, and your project requirements are similar, I find that running local Composer commands is decent workflow.

I can trust that the correct platform dependencies get installed at build-time, and it works out for me. Since I develop Docker applications with a mounted volume, my local command line changes reflect immediately in running containers.

Although I prefer to run commands locally, some projects fit running everything through Docker; especially if the environment is complicated or a legacy project. Using Docker for all Composer commands also ensures teams use the same Composer environment.

The context of the project is essential in determining your workflow. For projects with teams, I prefer to contain the entire workflow in Docker, but for smaller projects, on my own, I work locally. These are my own rules; I recommend you come up with your own as you experiment.

If you run commands locally, you can skip platform checks with:

```
$ composer install --ignore-platform-reqs
```

If you ignore platform requirements, be sure you run tests inside the container that verify your platform and have a proper CI process in place.

Composer Gold Edition

Hopefully, this chapter provided a decent primer for working with Composer in Docker. I think this is one of the more confusing and tricky parts of working with PHP in Docker. You should have all the tools you need to build composer dependencies are part of your Docker build, including private repositories.

I introduced the idea of building Composer dependencies outside of the image before running *docker build*. If you're new to Docker, the chances are that your existing build pipelines are already installing Composer dependencies outside of Docker. Keeping your build in-tact and copying the result into Docker could be a good stop-gap as you experiment with different build strategies with Docker.

Chapter 6: Web Servers and PHP-FPM

Up until this point, we've been using Apache as our web server. I like Apache, and sometimes I use it just because it's very convenient to set up and it works well. Another common server choice is Nginx, and so in this chapter, we focus on running Nginx in Docker.

We also look at Caddy server—an HTTP/2 server written in Golang and offering automatic SSL/TLS certifications—which another excellent option for your PHP projects.

Creating the New Project

In this chapter, we design our Nginx project to run a Nginx container and a separate PHP-FPM container. In this scenario, Nginx communicates with the PHP-FPM container through networking in the same way we've linked our application to a database container. We write the necessary Dockerfile configurations for both the PHP and Nginx containers and then hook them up via Docker Compose. Along the way, we cover serving static assets (i.e., JavaScript, CSS, etc.) with Nginx.

For this chapter, we will use the Slim (https://www.slimframework.com/) PHP Framework as we work through the Docker setup (Listing 6.1):

Listing 6.1: Project Setup

```
$ cd ~/Code
$ composer create-project slim/slim-skeleton:^3.0 ch6-nginx-fpm
$ cd ch6-nginx-fpm
```

Hello Nginx

Our first goal is getting Nginx running. The most simple way to do that with Docker is to run the official Nginx image and expose a port mapped to *80* in the container (Listing 6.2), allowing us to dive in quickly and play around:

Listing 6.2: Running the Default Nginx Container

```
$ docker run --name nginx-container -p 8080:80 -d nginx:1.12
Unable to find image 'nginx:stable' locally
...
1a103e90a864aa64ea2cab08b038f16744201bbd8296df11e61b5f41a297286e
```

We created a container with the official Nginx image using the stable tag, which at the time of this writing is 1.12. The *-d* flag detaches the container and prints the container ID and the *--name* flag assigns a name to the container. If you open up http://localhost:8080, you should see something similar to Figure 6.1.

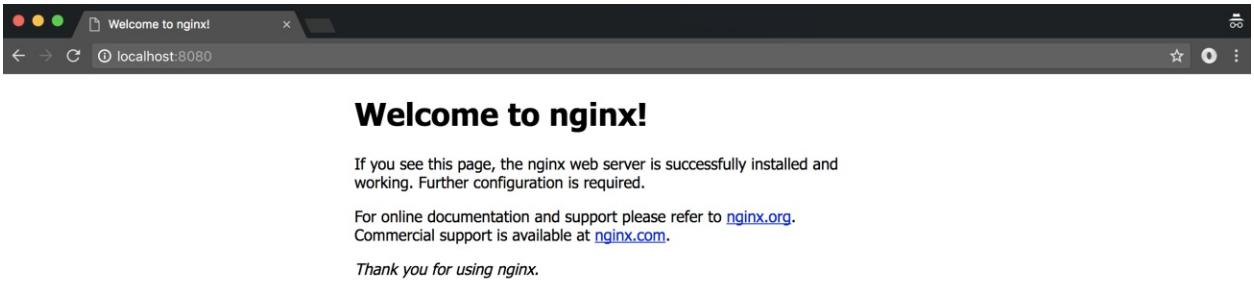


Figure 6.1: Welcome to Nginx

The ability to quickly run Nginx with one command without installing it locally or using a virtual machine is an excellent advantage over other environments. I love that Docker makes hacking on new technologies so easy, but unless we plan on running a static site, what we have won't get us very far. Let's stop and remove the container so we can start working on getting our own Nginx configuration communicating with a separate PHP-FPM container.

The container isn't running in the foreground because we used the *docker run -d* flag, but we can stop it with the *docker stop* command (Listing 6.3):

Listing 6.3: Stop All Running Containers

```
$ docker stop $(docker ps -q)
1a103e90a864

$ docker rm $(docker ps -aq)
6a0e60855eaa
817263bf67d6
d74fbc765bfc
# you might see multiple IDs here...
```

I just introduced the *-q* flag, which stands for quiet and only displays numeric IDs. We take advantage of this flag to stop and remove all containers. You could also run *docker ps*, grab the container ID(s), and then pass the ID to the *docker stop* command.

The second command, *docker rm*, removes the containers using the same trick to remove all containers. The remove command also requires the *-a* flag to return all container IDs, instead of just the running containers.

If you run *docker ps -a* again at this point, you get an empty list. Keep a reference of the commands in Listing 6.3 to clean up your local Docker environment quickly.

Setting up Nginx and PHP-FPM

To use Nginx, we are going to run two containers: one for the PHP-FPM container and the other for Nginx serving as a reverse-proxy for the PHP container. The Nginx container communicates with PHP over the network on port 9000 on a default internal network that Docker Compose automatically creates.

Let's start by creating the Docker project files (Listing 6.4):

Listing 6.4: Creating the Skeleton Files

```
$ cd ~/Code/ch6-nginx-fpm

$ mkdir -p .docker/php .docker/nginx/conf.d/
$ touch \
  .docker/nginx/Dockerfile \
  .docker/php/Dockerfile \
  .docker/nginx/nginx.conf \
  .docker/nginx/conf.d/app.conf \
  docker-compose.yml
```

After you create the files, here's what the project tree should look like:

```
|--- .docker
|    |--- nginx
|    |    |--- Dockerfile
|    |    |--- conf.d
|    |    |--- nginx.conf
|    |--- php
|    |    |--- Dockerfile
|--- .gitignore
|--- CONTRIBUTING.md
|--- README.md
|--- composer.json
|--- composer.lock
|--- docker-compose.yml
```

```
|--- logs
|--- phpunit.xml
|--- public
|--- src
|--- templates
|--- tests
|--- vendor
```

This chapter is the first time we are building multiple images in one project. I like to organize each image into a separate subfolder inside of the main *.docker/* folder, keeping things tidy and easy to follow.

We created a few Nginx config files that we copy into the Nginx image, including the main nginx.conf file and the app.conf file that allows us to customize the Nginx configuration.

The Nginx Image

Were are going to create the Nginx image first to serve static assets like images, CSS, and JavaScript, and everything else is proxied to the PHP-FPM container. We extend the official Nginx image and add our configuration to provide some flexibility in our setup and allow you to add custom configuration like different MIME types, enabling GZIP. Our application has a versioned server configuration that we'll copy into the image as well to serve the application.

Here's what the Dockerfile looks like for our Nginx image (Listing 6.5):

Listing 6.5: The Nginx Dockerfile

```
FROM nginx:1.12

LABEL maintainer="Paul Redmond"

RUN rm /etc/nginx/conf.d/default.conf
```

```
COPY .docker/nginx/nginx.conf /etc/nginx/nginx.conf
COPY .docker/nginx/conf.d/*.conf /etc/nginx/conf.d/
COPY . /srv/app/
```

We extend the *nginx:1.12* tag, which is the stable version of Nginx at the time of this writing. Next, we remove the *default.conf* server configuration, so we can version our own Nginx settings in the project. The last three lines copy configuration files that are version controlled in the project: the main nginx.conf and all files located in *conf.d/* that end in *.conf*.

Because we are overriding the main *nginx.conf* file contained in the Nginx image, let's grab the contents of the *nginx.conf* as a starting point that we'll version. We can get the configuration file contents by running a Nginx container and using *docker cp* to copy the default files to our project (Listing 6.6):

Listing 6.6: Copy the Nginx Config Files from the Container

```
$ docker run --name nginx-container -p 8080:80 -d nginx:1.12

$ docker cp nginx-container:/etc/nginx/nginx.conf
.docker/nginx/nginx.conf

$ docker stop nginx-container
$ docker rm nginx-container
```

Next, update the *.docker/nginx/conf.d/app.conf* file with the following contents for our server configuration (Listing 6.7):

Listing 6.5: The Nginx Dockerfile

```
server {
  listen 80;
  server_name localhost;
  index index.php;
```

```
  root /srv/app/public;

  location / {
    try_files $uri /index.php$is_args$args;
  }

  location ~ \.php {
    try_files $uri =404;
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME
$document_root$fastcgi_script_name;
    fastcgi_param SCRIPT_NAME $fastcgi_script_name;
    fastcgi_index index.php;
    fastcgi_pass app:9000;
  }
}
```

We've taken the Nginx configuration from the Slim documentation (https://www.slimframework.com/docs/start/web-servers.html#nginx-configuration) and adjusted a few things to match our environment. The *root* points to the application's *public* directory, and we've updated *fastcgi_pass app:9000;* to match the service name of the PHP container.

The PHP Image

Our next task is creating a PHP Dockerfile using the base *php:7.1-fpm* tag that runs the application with PHP-FPM. Because we are using the PHP image, we don't have to worry about the details of running the php-fpm process; we only need to copy our application code into the container and make sure the www-data user owns the files. The base image is doing most of the work for us!

The Dockerfile should look familiar to you at this point, enter the following in the *.docker/php/Dockerfile* file (Listing 6.8):

</> **Listing 6.8: The PHP Container**

```
FROM php:7.1-fpm

LABEL maintainer="Paul Redmond"

COPY . /srv/app/

WORKDIR /srv/app/

RUN chown -R www-data:www-data /srv/app
```

The Docker Compose File

The final step before we try out the code is defining our containers in the *docker-compose.yml* file. As of this writing Slim ships with a docker-compose.yml file in the root of the project, but we are going to replace it with our own services.

We need to link our containers together so that Nginx can proxy requests to PHP through the network that we'll define in the *docker-compose.yml* file (Listing 6.9):

</> **Listing 6.9: The PHP Container**

```
version: "3"
networks:
  app-tier:
    driver: bridge
services:
  app:
    build:
      context: .
      dockerfile: .docker/php/Dockerfile
    networks:
      - app-tier
    ports:
      - 9002:9000
```

```
  nginx:
    build:
      context: .
      dockerfile: .docker/nginx/Dockerfile
    networks:
      - app-tier
    ports:
      - 8080:80
```

We've introduced the *networks* key, and configured our nginx and php services to use this network. We've also added the *context* and *dockerfile* keys which allow us to reference the root folder as the context of the Docker build, yet keep our Dockerfile for each image tucked away in the *.docker/* folder.

The *context* and *dockerfile* keys are equivalent to *"docker build -f .doker/php/Dockerfile -t <the-tag> .*", the last dot being the current folder (context). By default, *docker build* looks on the same path for the *Dockerfile*, so when you need to build multiple images in a project, our technique works well to keep things organized.

Another thing to note in *Listing 6.9* is the non-standard PHP-FPM port number on the host machine. If you run a local environment containing PHP-FPM, port 9000 isn't available. We can use 9002 instead which is relatively arbitrary but avoids conflicts.

Running the Containers

We have everything in place needed to run our networked containers and verify that things are working as expected. This chapter is the first time we are building multiple images, so you see build output for both php and nginx images (Listing 6.10):

Listing 6.10: Running Docker Compose

```
$ docker-compose up --build
...
Attaching to ch6nginxfpm_nginx_1, ch6nginxfpm_app_1
```

```
app_1  | [28-May-2017 08:02:32] NOTICE: fpm is running, pid 1
app_1  | [28-May-2017 08:02:32] NOTICE: ready to handle
connections
```

With any luck, you should see the default Slim response (Figure 6.2):

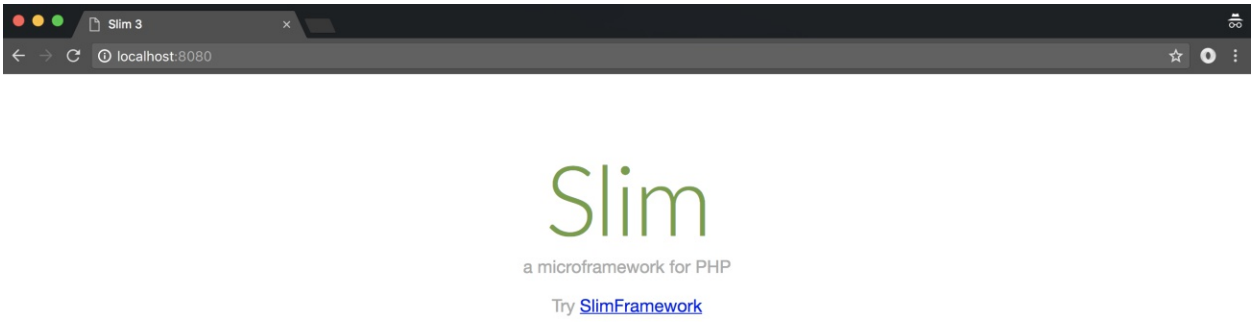


Figure 6.2: Nginx + PHP-FPM Success!

Now that things are working let's add a volume so we can edit files locally while developing (Listing 6.11):

Listing 6.11: Add Volumes for Local Development

```
version: "3"
networks:
  app-tier:
    driver: bridge
services:
  app:
    build:
      context: .
```

```
      dockerfile: .docker/php/Dockerfile
    networks:
      - app-tier
    ports:
      - 9002:9000
    volumes:
      - .:/srv/app
  nginx:
    build:
      context: .
      dockerfile: .docker/nginx/Dockerfile
    networks:
      - app-tier
    ports:
      - 8080:80
    volumes:
      - ./public:/srv/app/public
```

You need to restart the containers for the volumes to take effect. You can do so by hitting "Ctrl+c" if you are running docker-compose in the foreground.

If you are running Docker Compose in the background you need to run the following:

```
$ docker-compose down
$ docker-compose up
```

The volumes (https://docs.docker.com/compose/compose-file/#volumes) configuration option includes a host volume mounting your project code in the container path */srv/app* in the PHP container, and */srv/app/public* in the Nginx container, allowing you to edit files locally and have them reflect immediately.

File Permissions

Depending on your environment, you might get write permission errors. If you run into this issue, run *chmod -R o+rw the/folder/* locally.

Serving Static Assets

The primary role of the Nginx container is serving static assets like JavaScript and CSS, and proxying requests to the PHP container. We are copying the whole application into both images, but all Nginx needs is the contents of the public folder.

Along the way, we focus on making Nginx configuration tweaks, learn how to define additional custom mime types, and enable gzip compression for static assets. You will start to see how versioning the Nginx configuration pays off.

The Nginx configuration might be a review for you, but let's focus on our Docker workflow around configuration, updating images, and running and debugging containers.

Before we start changing configuration, let's first change the Nginx Dockerfile to only copy the *public/* folder (Listing 6.12):

Listing 6.12: Update the COPY step in the Nginx Dockerfile

```
FROM nginx:1.12

LABEL maintainer="Paul Redmond"

RUN rm /etc/nginx/conf.d/default.conf
COPY .docker/nginx/nginx.conf /etc/nginx/nginx.conf
COPY .docker/nginx/conf.d/*.conf /etc/nginx/conf.d/
COPY public/ /srv/app/public
```

Let's see how the change from Listing 6.12 affects the image (Listing 6.13):

Listing 6.13: Rebuild the Nginx Image and Inspect It

```
$ docker-compose down
$ docker-compose build nginx
$ docker-compose up

# Find the nginx container id
$ docker ps
$ docker exec -it ca3f5aaab045 bash

# Inside the container
$ ls -la /srv/app/public/
total 16
drwxr-xr-x 2 root root 4096 May 29 19:08 .
drwxr-xr-x 3 root root 4096 May 29 19:08 ..
-rw-r--r-- 1 root root  313 Sep  4  2016 .htaccess
-rw-r--r-- 1 root root  725 May 29 18:31 index.php
```

Now we don't have any unnecessary PHP code in the Nginx image, keeping it small and tidy.

Adding Gzip Compression

By default, Nginx doesn't enable Gzip compression, but we can easily modify our configuration to reduce file size for faster network transfers, which speeds up your applications and save on bandwidth costs.

To get a feel for working with static files, let's download jQuery and use it to verify configuration changes we are going to make to enable Gzip compression (Listing 6.14):

Listing 6.14: Downloading jQuery for Testing

```
$ mkdir -p public/js

# Download jQuery
$ curl -sS https://code.jquery.com/jquery-3.2.1.js \
  > public/js/jquery.js
```

Before working on getting Gzip enabled, let's verify that Nginx is serving jQuery correctly by running the updated Nginx container and making a request (Listing 6.15):

Listing 6.15: Verify Nginx is Serving Static Files

```
$ curl -H "Accept-Encoding: gzip" \
  -I http://localhost:8080/js/jquery.js

HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Wed, 15 Nov 2017 14:09:59 GMT
Content-Type: application/javascript
Content-Length: 268039
Last-Modified: Wed, 15 Nov 2017 14:09:16 GMT
Connection: keep-alive
ETag: "5a0c4a8c-41707"
Accept-Ranges: bytes
```

Nginx isn't serving our JavaScript with gzip encoding because our *nginx.conf* file has gzip commented out (*#gzip on;*). We want to be able to serve compressed responses, so let's adjust our *nginx.conf* file and enable gzip (Listing 6.15):

Listing 6.15: Adding gzip Compression to Nginx

```
user  nginx;
worker_processes  1;
```

```
error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
}

http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main
        '$remote_addr - $remote_user [$time_local] "$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    #tcp_nopush     on;

    keepalive_timeout  65;

    gzip  on;
    gzip_disable "msie6";
    gzip_min_length 256;
    gzip_types
        text/plain
        text/css
        application/json
        application/x-javascript
        application/javascript
        text/xml
        application/xml
        application/xml+rss
        text/javascript
        application/vnd.ms-fontobject
```

```
        application/x-font-ttf
        font/opentype
        image/svg+xml
        image/x-icon
    ;

    include /etc/nginx/conf.d/*.conf;
}
```

Note that we disable gzip for older Internet Explorer browsers and target specific mime types. Other than that, you can reference the Nginx documentation if you need a refresher on gzip configuration or want to expand on what we have.

As you probably have guessed by now, we need to rebuild the image before we can verify that our changes work (Listing 6.16):

Listing 6.16: Testing Out gzip

```
$ docker-compose down
$ docker-compose build nginx
$ docker-compose up -d

$ curl -H "Accept-Encoding: gzip" \
       -I http://localhost:8080/js/jquery.js

HTTP/1.1 200 OK
Server: nginx/1.12.2
Date: Wed, 15 Nov 2017 14:12:34 GMT
Content-Type: application/javascript
Last-Modified: Wed, 15 Nov 2017 14:09:16 GMT
Connection: keep-alive
ETag: W/"5a0c4a8c-41707"
Content-Encoding: gzip
```

We have compressed assets in Nginx, excellent work! I recommend exploring the rest

of the Nginx configuration, such as the */etc/nginx/mime.types* file, to get familiar with tweaking Nginx. I love how Docker makes it easy to experiment with technologies without affecting my local computer.

Removing Server and PHP Version

You might have noticed the *Server: nginx* and *X-Powered-By: PHP* headers while we were testing out gzip. Let's quickly disable those before we call our Nginx setup good.

First, disable server tokens in the Nginx server configuration to mask the Nginx version (Listing 6.17):

</> Listing 6.17: Remove the Nginx Version from the Server Header

```
server {
  listen 80;
  server_name localhost;
  server_tokens off;
  index index.php;
  root /srv/app/public;

  location / {
    try_files $uri /index.php$is_args$args;
  }

  location ~ \.php {
    try_files $uri =404;
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME
$document_root$fastcgi_script_name;
    fastcgi_param SCRIPT_NAME $fastcgi_script_name;
    fastcgi_index index.php;
    fastcgi_pass app:9000;
  }
}
```

Next is the *X-Powered-By* PHP header. We haven't created a *php.ini* file for this project yet, so let's create that file locally (Listing 6.18):

>_ Listing 6.18: Create the php.ini File

```
$ echo "expose_php = off" > .docker/php/php.ini
```

The "expose_php" setting removes the "X-Powered-By" header sent back in responses from the application.

The last step is copying the *php.ini* file into the image (Listing 6.19):

</> Listing 6.8: The PHP Container

```
FROM php:7.1-fpm

LABEL maintainer="Paul Redmond"

COPY . /srv/app/
COPY .docker/php/php.ini /usr/local/etc/php/php.ini

WORKDIR /srv/app/

RUN chown -R www-data:www-data /srv/app
```

After you rebuild your images, the *X-Powered-By* PHP header is gone, and the Nginx header doesn't expose the Nginx version anymore. Hiding this server information from prying eyes is ideal because would-be attackers have less knowledge about the specific setup.

```
$ docker-compose down
$ docker-compose up -d --build

$ curl -I http://localhost:8080
```

```
HTTP/1.1 200 OK
Server: nginx
Date: Wed, 15 Nov 2017 14:24:01 GMT
Content-Type: text/html; charset=UTF-8
Connection: keep-alive
Set-Cookie: PHPSESSID=9e0ca0a6e324c92c38ab625e04c642b1; path=/
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate
Pragma: no-cache
```

We have an excellent foundation for a Nginx and PHP setup in that our images are simple and easy to extend. You can add complexity as you need it, but I think this chapter proves that Docker can provide an extensible setup that is not complicated.

Next, we'll check out another up-and-coming HTTP/2 web server, Caddy, which is another alternative you can use besides Apache and Nginx.

Caddy Server

Another server you can use with PHP is my personal favorite right now: Caddy (https://caddyserver.com/), an HTTP/2 server written in Golang. Caddy is a pleasure to work with, and configuration feels clean and straightforward to me. I also feel that it simplifies my application containers even further and I am going to show you a technique to run Caddy and PHP-FPM without multiple containers.

You can also run Caddy just like we ran Nginx with a separate Caddy container and a PHP-FPM container, but I'll let you figure that out on your own (hint, you just need to modify the FastCGI directive to point to a separate PHP container).

Caddy Setup

We're going to pick up the pace a little in this section, but it should be a breeze! First things first, let's create a new Laravel project for this section and start fresh (Listing 6.20):

Listing 6.20: Creating the Laravel Project

```
$ cd ~/Code
$ composer create-project --prefer-dist \
  laravel/laravel:5.5.* ch6-caddy
$ cd ch6-caddy
$ mkdir .docker/
$ touch .docker/Dockerfile .docker/Caddyfile
$ touch docker-compose.yml
```

We created a Dockerfile for our PHP code and Caddy binary, and a Caddyfile to configure the Caddy server.

File Permissions

If you are running Docker on Linux, you might get write permission errors on volume directories. If you run into this issue, run *chmod -R o+rw storage/bootstrap/* locally.

Next, let's create the docker-compose.yml file (Listing 6.21):

Listing 6.21: The Docker Compose File

```
version: "3"
services:
  app:
    build:
      context: .
      dockerfile: .docker/Dockerfile
    ports:
      - 2015:2015
    volumes:
      - .:/srv/app
```

We map port 2015, which is the default port that Caddy uses to accept HTTP requests, but you can quickly change which port Caddy runs on through the Caddyfile.

The Caddy Dockerfile

We don't need to extend a Caddy image like we are doing with Nginx, we can just use the php-fpm image and install the Caddy binary.

Add the following in the Dockerfile to install Caddy (Listing 6.22):

</> Listing 6.21: The Docker Compose File

```
FROM php:7.1-fpm

LABEL maintainer="Paul Redmond"

# Install application dependencies
RUN curl --silent --show-error --fail --location \
      --header "Accept: application/tar+gzip, application/x-gzip,
application/octet-stream" -o - \

"https://caddyserver.com/download/linux/amd64?plugins=http.expire
s,http.realip&license=personal" \
    | tar --no-same-owner -C /usr/bin/ -xz caddy \
    && chmod 0755 /usr/bin/caddy \
    && /usr/bin/caddy -version \
    && docker-php-ext-install mbstring pdo pdo_mysql

COPY .docker/Caddyfile /etc/Caddyfile
COPY . /srv/app/

WORKDIR /srv/app/

RUN chown -R www-data:www-data /srv/app

CMD ["/usr/bin/caddy", "--conf", "/etc/Caddyfile", "--log",
"stdout"]
```

First, we download Caddy via cURL, using query string parameters to define the Caddy plugins we want to use and the license type. You can adjust the plugins based on your needs by visiting https://caddyserver.com/download and copying the download link after selecting your configuration. We include the *http.expires*, *http.git*, and *http.realip* plugins here. Make sure to read the EULA if you plan on using Caddy in a commercial project!

After downloading Caddy, the command extracts the binary from the archive, makes it executable, and moves it to */usr/bin/caddy*. Last, we defined the CMD instruction with a *--conf* flag pointing to our *Caddyfile* which we need to write, and the *--log* flag instructs Caddy to send logs to stdout, which show up in the Docker console.

The Caddyfile

The Caddyfile is a text file that configures how Caddy runs, and I think you'll like the simple syntax. Here's a basic Caddfile that will run our Laravel application (Listing 6.23):

Listing 6.23: The Caddyfile

```
0.0.0.0
root /srv/app/public
gzip
fastcgi / 127.0.0.1:9000 php
rewrite {
    regexp .*
    ext /
    to /index.php?{query}
}

header / -Server

log stdout
errors stdout
on startup php-fpm --nodaemonize
```

The first line defines the site address of our application, which is 0.0.0.0, or localhost. We can use this line to specify the site address, which can take many forms. To learn more, you should check out the HTTP Caddyfile (https://caddyserver.com/docs/http-caddyfile) documentation.

The next line, *gzip*, is known as a Caddy directive. We configure the gzip directive to use the defaults, but you could customize the configuration with a block:

```
gzip {
    ext         extensions...
    not         paths
    level       compression_level
    min_length min_bytes
}
```

Next, we are defining a FastCGI proxy so we can communicate with PHP-FPM. After FastCGI, we define a main rewrite (https://caddyserver.com/docs/rewrite) rule that rewrites everything to our app's *public/index.php* file. We wrap up the Caddyfile by sending access and error logs to stdout and stderr which show up in Docker's console.

The last line starts PHP-FPM in the background with *on startup* event. The startup event is triggered just before the server starts listening.

Learning more about the Caddyfile

You should check out the Caddyfile tutorial (https://caddyserver.com/tutorial/caddyfile) and then the user guide documenation (https://caddyserver.com/docs/http-caddyfile) to learn more about the Caddyfile.

Running the Application

We have all the components we need to run the application, so let's try it out (Listing 6.24):

Listing 6.24: Running Caddy

```
$ docker-compose up --build
...
Attaching to ch6caddy_app_1
app_1 | Activating privacy features... done.
app_1 | 2017/06/11 07:48:01 [INFO] Nonblocking Command:"php-fpm "
app_1 | http://0.0.0.0:2015
app_1 | 2017/06/11 07:48:01 http://0.0.0.0:2015
app_1 | [11-Jun-2017 07:48:01] NOTICE: fpm is running, pid 14
app_1 | [11-Jun-2017 07:48:01] NOTICE: ready to handle
connections
app_1 | 127.0.0.1- 11/Jun/2017:07:48:26 +0000 "GET /index.php"
200
```

If you decipher the logs, you can see that Caddy is running on port 2015. You are free to change it in the configuration, but we just used the default for this application.

If you visit http://localhost:2015/, you should see the Laravel welcome page, which means our application is working.

Logs to Stdout

In Laravel, you can configure the app log settings to errorlog to send logs to the Docker console.

You can set the *APP_LOG=errorlog* environment variable in the *docker-compose.yml* file to try it out. Afterwards, try running *\Log::debug("Test Log");* to see your application logs from the Docker CLI.

Next let's test out our rewrite rule to make sure that other URLs route to the laravel application correctly. We can do that by adding a new route to the bottom of *routes/web.php* (Listing 6.25):

</> Listing 6.25: Add a Route to Test the Caddy Rewrite

```
Route::get('/hello', function () {
    return ['hello' => request('name', 'world')];
});
```

The route returns an array, which Laravel will automatically convert into a JSON response (Listing 6.26):

>_ Listing 6.26: Verify that the Rewrite is Working

```
$ curl http://localhost:2015/hello\?name\=paul
{"hello":"paul"}
```

That completes our whirlwind tour of using Caddy in Docker; I highly recommend that you dig deeper into the plugin documentation and tinker around with Caddy to learn more! I've been using Caddy in production for over a year, and it's an excellent server.

You've Been Served

Hopefully, this chapter was an excellent foundation for running PHP applications in a variety of ways. We expanded beyond running Docker with Apache and learned how to use containers to run Nginx and PHP-FPM separately. I hope you also give Caddy a try, it is a pleasure to work with and I use it on a daily basis.

Now, we move on to working with legacy PHP applications you still might have in your life. Hopefully, you can wrangle your applications in a contained Docker environment, which is a big step into refactoring and replacing them.

Chapter 7: Legacy PHP Applications

We all have that legacy application in our lives. You know, the one of which you are afraid to restart Apache? Yeah. We've all been there! I find it odd that I am dedicating a whole chapter in my brand-spanking-new book on legacy code?! In this case, Docker might help you wrangle that project (and learn how to set it up) without destroying your local machine.

It's annoying when you have the latest-and-greatest PHP version on your laptop, and you get that dreaded request to update the legacy project. Maybe you are using Vagrant, but Vagrant is only helping you in development. You still have to make sure you have environment parody.

This chapter is handy when you need to set up an environment from scratch. We've been relying on the official PHP image—I recommend you stick with that in your PHP projects—but you probably noticed that the official PHP image doesn't support PHP <= 5.4. In fact, the official image only goes back to PHP 5.6 at the time of writing. We'll have to use an older OS version that supports our madness.

Working with older code gets harder as time goes on (and less secure); requiring you to build EOL versions of PHP from source, or using an older OS version (i.e., CentOS 6) that ships with an older PHP package.

End of Life (EOL) Software

Running end-of-life software is bad. We've all been there, but I still think it's worth noting.

I am not showing you these techniques to encourage you to keep old software around longer. I am teaching you these methods because realistically everyone deals with these types of projects.

Setting Up

As we work through this chapter, we use an example PHP project to build a Docker environment and retrofit configuration to work well with a container paradigm. Specifically, we are going to work with an older version of CakePHP—CakePHP 2 to be exact (https://book.cakephp.org/2.0/en/index.html). Newer versions of CakePHP 2 work on PHP 7, but older versions also work with PHP 5.3 which is perfect for this lesson.

I have a place in my heart for CakePHP. It was my first MVC framework, and I am not trying to pick on it by any means. In fact, I chose it for this chapter since the documentation is still available, and they do a great job of keeping older releases accessible. CakePHP is under active development at the time of this writing and is a robust MVC PHP framework.

As always, the first thing we do is create a project directory structure and add a few files we'll use to build the Docker environment (Listing 7.1).

Listing 7.1: Project Setup

```
$ cd ~/Code
$ curl -LOk https://github.com/cakephp/cakephp/archive/2.6.0.zip
$ unzip 2.6.0.zip -d ./
$ rm 2.6.0.zip
```

```
$ cd cakephp-2.6.0
$ mkdir .docker/
$ touch docker-compose.yml
$ touch .docker/Dockerfile \
    .docker/vhost.conf \
    .docker/httpd-foreground
```

According to the CakePHP documentation, CakePHP 2.6 and below support PHP >=5.2.8 (https://book.cakephp.org/2.0/en/installation.html#requirements). We use a version of PHP 5.3 in this chapter, but you could adapt this to other versions, which requires building PHP and needed extensions from source.

The Dockerfile

We have been extending from the official Docker PHP image up to this point, but in this chapter, we are going to extend the official Ubuntu image.

There are two common ways we can install older versions of PHP. First, we can use a PHP package from an older Linux version. Second, we can build PHP (and the required modules) from source.

In our case, we are going to use Ubuntu 12 LTS to install a more hardened version of PHP 5.3, but the tradeoff is that the package version installs PHP 5.3.10. A huge positive for taking this route is that you benefit from PHP security patches provided by Canonical maintainers, and the package already includes Suhosin (https://suhosin.org/). I highly suggest you take this route if possible.

If your application depends on >= 5.3.10, you need to install a newer version of PHP 5.3 from source. I'm not going to cover that here, but if you are in this predicament, I am guessing you're used to building PHP from source. The bad news is that you'll miss out on some of the security patching prepared by Canonical.

With that bit of explanation out of the way, let's start by installing PHP 5.3 from a package using Ubuntu 12.04 as the base image (Listing 7.2):

Listing 7.2: Install PHP 5.3 from Ubuntu 12.04 Packages

```
FROM ubuntu:12.04

LABEL maintainer="Paul Redmond"

RUN apt-get -yqq update \
    && apt-get -yqq install \
        apache2 \
        libapache2-mod-php5 \
        php5 \
        php5-mysql \
        php5-mcrypt \
        php5-suhosin \
    && a2enmod rewrite
```

We are starting out by installing *apache2*, *mod_php*, some PHP packages. Lastly, we enable *mod_rewrite* using the *a2enmod* command.

Let's build the image and investigate a little to see what we get. This command will take a little longer the first time because Docker needs to pull Ubuntu 12 from the official Docker repository (Listing 7.3):

Listing 7.3: Build the Docker Image

```
$ docker build -t cakephp-app -f .docker/Dockerfile .
# ...
# Let's run a container
$ docker run --rm -it cakephp-app /bin/bash

# Inside the container, run `php -v`
root@55318cb78516:/# php -v
PHP 5.3.10-1ubuntu3.26 with Suhosin-Patch (cli) (built: Feb 13
2017 20:37:53)
Copyright (c) 1997-2012 The PHP Group
```

```
Zend Engine v2.3.0, Copyright (c) 1998-2012 Zend Technologies
    with Suhosin v0.9.33, Copyright (c) 2007-2012, by SektionEins
GmbH

# Let's look at Apache
$ ls -la /etc/apache2/
```

Running Apache

We've already included the apache2 package in our Dockerfile, so let's get Apache running in the container. We are going to do a couple of things to make this happen:

1. Add an application service to the *docker-compose.yml* file
2. Create a script that will run Apache in the container

I am of the opinion that running Apache with a legacy version of PHP is probably the best choice because it's easy to install and many legacy PHP applications were designed to work with Apache.

Let's get started by creating a *docker-compose.yml* file that should look familiar to you at this point. You can try and create it on your own, or use the following (Listing 7.4):

</> Listing 7.4: Adding the docker-compose.yml File

```
version: "3"
services:
  app:
    build:
      context: .
      dockerfile: .docker/Dockerfile
    ports:
      - 8888:80
    volumes:
      - .:/srv/cakephp
```

We are setting the build context to the root of the project and specifying the path to the Dockerfile. We've already used this style, which allows you to organize your Docker build files in a subfolder.

The next file we need to work on is the script we created in Listing 7.1, *.docker/httpd-foreground*, that will be used to run Apache (Listing 7.5):

</> Listing 7.5: The httpd-foreground File

```
#!/bin/bash
set -e

# Apache gets grumpy about PID files pre-existing
rm -f /usr/local/apache2/logs/httpd.pid
source /etc/apache2/envvars && exec apachectl -D FOREGROUND
```

We make sure that the Apache environment variables are sourced and run Apache in the foreground so that the Docker container won't exit.

Next, we need to copy this file into the container and make it executable (Listing 7.6):

</> Listing 7.6: Copy the http-foreground script into the image

```
FROM ubuntu:12.04

LABEL maintainer="Paul Redmond"

RUN apt-get -yqq update \
    && apt-get -yqq install \
        apache2 \
        libapache2-mod-php5 \
        php5 \
        php5-mysql \
        php5-mcrypt \
        php5-suhosin \
```

```
    && a2enmod rewrite

COPY .docker/httpd-foreground /usr/local/bin/

RUN chmod +x /usr/local/bin/httpd-foreground

EXPOSE 80

CMD ["httpd-foreground"]
```

Because the *http-foreground* script is in the $PATH, we can reference it without the full path.

The CMD provides defaults for an executing container, and in our case, the default is our custom bash script. Up to this point, we've relied on the CMD instruction from the base PHP images we extend, which is why this is the first time you've seen it in this text. We go over providing custom commands more in-depth in the next chapter.

Now that we have a way to run Apache in the container by default, we are ready to rebuild the image and verify that Apache is working as expected (Listing 7.7):

Listing 7.7: Run the Container to Verify Apache

```
$ docker-compose up --build
```

If you visit http://localhost:8888 you should see the default Apache response which means Apache is running in the container (Figure 7.1):

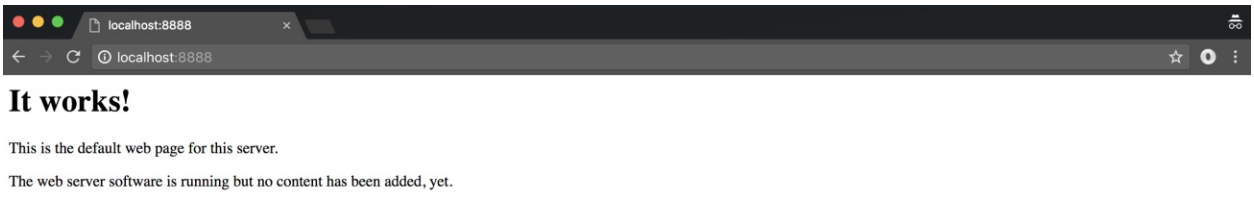


Figure 7.1: The Default Apache Page

The default Apache response means our setup is working and we can move on to creating a Virtual Host configuration for our application. Add the following VirtualHost configuration to *.docker/vhost.conf* (Listing 7.8):

Listing 7.8: The Virtual Host File

```
<VirtualHost *:80>
    # ServerName www.example.com

    DocumentRoot /srv/cakephp/app/webroot

    <Directory "/srv/cakephp/app/webroot">
        DirectoryIndex index.php
        Options -Indexes
        Order allow,deny
        allow from all
        RewriteCond %{REQUEST_FILENAME} !-d
        RewriteCond %{REQUEST_FILENAME} !-f
        RewriteRule ^ index.php [L]
```

```
    </Directory>

    ErrorLog /dev/stderr
    CustomLog /dev/stdout combined
</VirtualHost>
```

We are matching up the DocumentRoot to the webroot folder of the application. We have yet to define the COPY instruction to copy the application into the image, which we address shortly.

The Ubuntu 12.04 Apache package is Apache version ~ 2.2, so the "allow" syntax matches that version. Last, we configure the ErrorLog and CustomLog to go to stderr and stdout respectively so that logs go to the Docker console instead of piling up in the container.

Next, we need to update the Dockerfile to copy the source code into the path we defined for the DocumentRoot. We also need to enable the VirtualHost configuration too so that Apache can serve our application (Listing 7.9):

Listing 7.9: The Updated Dockerfile

```
FROM ubuntu:12.04

LABEL maintainer="Paul Redmond"

RUN apt-get -yqq update \
    && apt-get -yqq install \
        apache2 \
        libapache2-mod-php5 \
        php5 \
        php5-mysql \
        php5-mcrypt \
        php5-suhosin \
    && rm -f /etc/apache2/sites-available/* \
    && rm -f /etc/apache2/sites-enabled/* \
```

```
    && a2enmod rewrite

COPY .docker/httpd-foreground /usr/local/bin/
COPY .docker/vhost.conf /etc/apache2/sites-available/000-
default.conf
COPY . /srv/cakephp

RUN ln -s /etc/apache2/sites-available/000-default.conf \
        /etc/apache2/sites-enabled/000-default.conf \
    && chmod +x /usr/local/bin/httpd-foreground \
    && chown -R www-data:www-data /srv/cakephp

WORKDIR /srv/cakephp

EXPOSE 80

CMD ["httpd-foreground"]
```

Our latest Dockerfile adds in two *rm* commands to remove the Apache default v-host file; we don't need it. Next, we copy the all the source code to */srv/cakephp* which matches the path in our Vhost. Because our code's webroot is at *app/webroot*, the final path is */srv/cakephp/app/webroot*.

Although the vhost.conf is the only virtual host, we name it *000-default.conf* in the image because we want Apache to make this the default. Apache determines the first Vhost (alphabetically sorted) as the default.

The last RUN command symlinks our Virtual Host file into the *sites-enabled* folder and enables the Vhost. Lastly, we change ownership of the application files to the www-data user, which allows Apache read and write access.

We should now be able to run our CakePHP application after rebuilding the Docker image with *docker-compose up --build* (Figure 7.2).

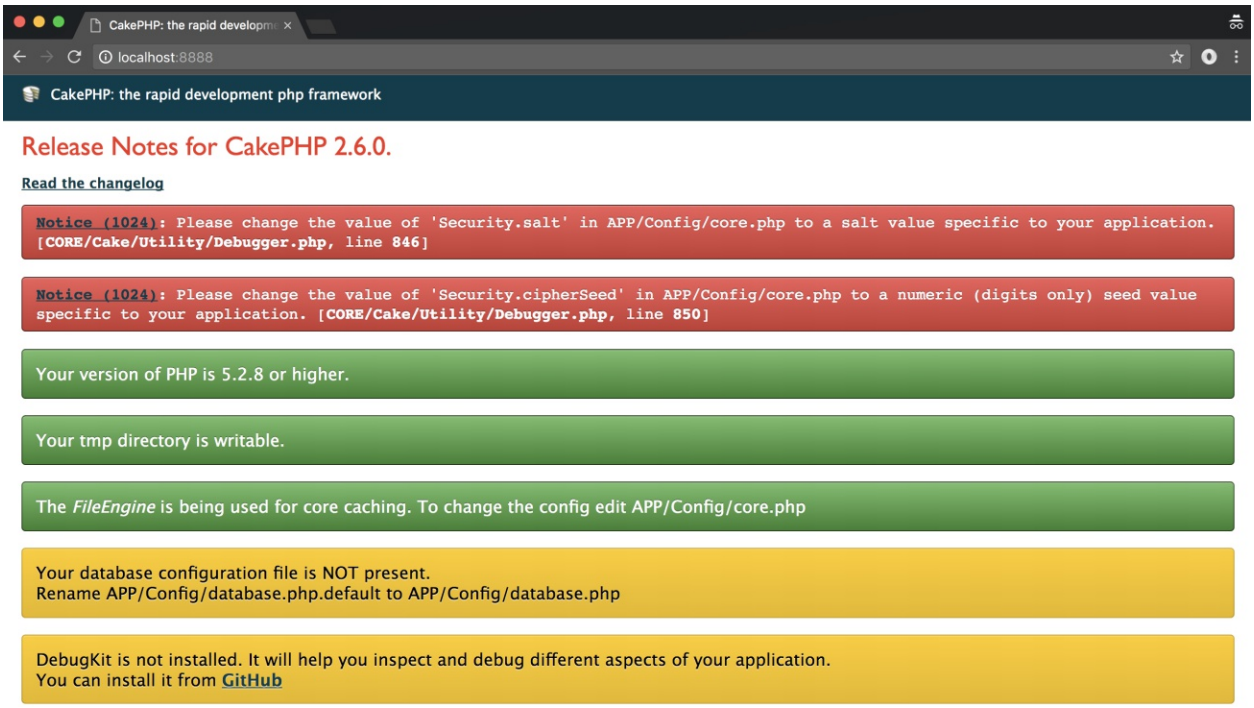


Figure 7.2: CakePHP Working with PHP 5.3

As you can see, our application is running, but there are a few warnings and suggestions that need some work. Let's look at how we might retrofit modern configuration patterns into the application.

Development Workflow and Commands

You probably don't have PHP 5.3 on your local machine, so will need to run all of our console commands inside of the container to work with PHP. You've already seen plenty of examples jumping into a running container within this book.

We can also run commands in our container from a local terminal with *docker run*. Let's create another service in our *docker-compose.yml* file to see how it all works together (Listing 7.10):

Listing 7.10: Adding a Cake Service to docker-compose.yml

```
version: "3"
services:
```

```
  app:
    image: cakephp-app
    container_name: cakephp-app
    build:
      context: .
      dockerfile: .docker/Dockerfile
    ports:
      - 8888:80
    volumes:
      - .:/srv/cakephp
  cake:
    image: cakephp-app
    container_name: cakephp-console
    volumes:
      - .:/srv/cakephp
    entrypoint: [
      "/srv/cakephp/app/Console/cake",
      "-app", "/srv/cakephp/app"
    ]
```

The *cake* service is the biggest change here. We added an *entrypoint* which changes the way the container works when we run it. Because we didn't define an ENTRYPOINT in our Dockerfile, we have some flexibility in how we run the application container. By default, we can run Apache, and then we can customize other containers to run the *cake* CLI.

According to the CakePHP 2 console documentation (https://book.cakephp.org/2.0/en/console-and-shells.html), you can pass the *-app* argument to customize the location of the application. We add it to the entrypoint just to be sure the Cake CLI knows how to locate the application.

We also added an *image*: key to both the *app* and *cake* services, so they run from the same image instead of building the same image twice. Using the *container_name* key on both services was added make the names of containers a little more comfortable on the eyes.

If you run the *cake* service you can see how the entrypoint setting works in Docker Compose (Listing 7.11):

Listing 7.11: Running the Cake Console

```
$ docker-compose run cake

Welcome to CakePHP v2.6.0 Console
# ...
Current Paths:

 -app: app
 -working: /srv/cakephp/app
 -root: /srv/cakephp
 -core: /srv/cakephp/lib
# ...
```

Since the entrypoint points to the CakePHP console you can pass arguments that will run in the container and use the cake shell as expected (Listing 7.12):

Listing 7.12: Running cake Commands

```
# From your local machine
$ docker-compose run cake command_list
```

The other way we could run this without defining a docker-compose service is using the *--entrypoint* flag. You can start a container with bash and execute console commands within the container, which feels similar to SSH (Listing 7.13):

Listing 7.13: Customize the Entrypoint with the Compose Command

```
# From your local machine
$ docker-compose run --entrypoint=/bin/bash app
```

You are now free to move about the container, and you benefit from the volume mount defined in the Docker Compose file to keep your local files in sync. To exit the

container, hit *Control + d* or type "exit" at the prompt.

Improving Configuration

Let's jump into working with legacy configuration and making it work well with Docker. We don't have any databases or application configuration yet, and we should go over a few things related to a configuration issue that I see in most legacy projects migrating to Docker.

You might have noticed a few configuration warnings in Figure 7.2, and one standard challenge I've faced on every project is dealing with configuration between environments. There are a couple of strategies I've used to make configuration easier to deal with that I want to cover.

Those strategies include:

1. Having separate configuration files for each environment (ie. *prod.config.php*, *dev.config.php*, etc.).
2. Provide a tokenized configuration file that gets updated during build-time.
3. Integrating updated configuration patterns like environment variables (https://en.wikipedia.org/wiki/Environment_variable).

I personally dislike having production settings in the repository; it's not a right way of promoting security on an already aging codebase. In my experience, I've often seen a combination of configuration files per environment in a code repository combined with tokenized replacements for sensitive values like database passwords. However, in my experience, retrofitting your code to work with environment variables is probably the best approach if you can salvage it.

Using environment variables gets around annoying build-time token replacements of sensitive passwords or using something like Ansible to generate the configuration file that gets copied into an image. With environment variables, you just define them, and the code adapts.

Let's go with the last option, and retrofit our codebase to work with environment variables. At the time of this writing, there is a composer package, *vlucas/phpdotenv*, which supports our application's version of PHP! CakePHP 2.6 doesn't use Composer for its autoloading, but we can use it for adding the PHP dotenv package (Listing 7.14):

>_ Listing 7.14: Installing Composer Dependencies and Adding phpdotenv

```
$ composer install
# Ignore vendor/
$ echo "vendor/" >> .gitignore
$ composer require vlucas/phpdotenv:~2.4.0
```

Mcrypt Extension

You need to install the Mcrypt extension locally to run composer commands with CakePHP. For example, if you are on OS X with PHP 7.1 installed from Homebrew, you can run *brew install php71-mcrypt*.

Using the strategies we learned in Chapter 6, you could also run all the commands—including composer—inside a container with a mounted volume.

If you are following along, you now have a new *composer.lock* file, a *Plugins* folder created from the *cakephp/debug_kit* dependency, and the *vendor/* folder with PHPUnit.

Now that we have the package installed, we need to include Composer's autoloader. In CakePHP 2.6, the framework does not use Composer's autoloader, so we add the autoloader to the top of the entrypoint of the application.

Edit the *app/webroot/index.php* file with the following at the very top (Listing 7.15):

Listing 7.15: Adding dotenv to app/webroot/index.php

```
<?php

// Load Dotenv
require __DIR__.'/../../vendor/autoload.php';
$dotenv = new Dotenv\Dotenv(__DIR__.'/../../');
$dotenv->load();

// ...
```

The *Dotenv* instance looks in the root of the project for the *.env* file and fail if it cannot find the file.

Next, let's create our *.env* file, add an example *.env.example* file, and ignore the *.env* file in Git (Listing 7.16):

Listing 7.16: Adding the .env files

```
$ cd ~/Code/cakephp-2.6.0
$ echo "APP_DEBUG=1" > .env > .env.example
$ echo ".env" >> .gitignore
```

Now we have a way of setting environment variables in our application on the system and through a *.env* file. A system-defined variable takes precedence over a value in the *.env* file, and like modern applications, we can now set some defaults and override things with environment variables for production.

Now that we've brought the environment library into the project, we still have some work to do on the integration. The PHP dotenv package doesn't perform type conversions on values, so *MY_ENV=true* is a string value "true."

Let's work on a helpers file that includes a few utility functions for working with our environment configuration. We borrow a little from the Laravel Support helpers.php

file to make our integration support type conversions (https://git.io/vb9sx).

We will provide some helper functions to read environment variables and autoload the file through composer's autoloader. Create a *app/helpers.php* file and add the following helper functions (Listing 7.17):

Listing 7.17: Adding a helpers.php File

```
<?php

if (! function_exists('value')) {
    /**
     * Return the default value of the given value.
     *
     * @param  mixed  $value
     * @return mixed
     */
    function value($value)
    {
        return $value instanceof Closure ? $value() : $value;
    }
}

if (! function_exists('cakeenv')) {
    function cakeenv($key, $default = null) {
        $value = getenv($key);

        if ($value === false) {
            return value($default);
        }

        switch (strtolower($value)) {
            case 'true':
            case '(true)':
                return true;
            case 'false':
```

```
            case '(false)':
                return false;
            case 'empty':
            case '(empty)':
                return '';
            case 'null':
            case '(null)':
                return;
        }

        if (
            strlen($value) > 1 &&
            strpos($value, '"') === 0 &&
            strpos(strrev($value), '"') === 0
        ) {
            return substr($value, 1, -1);
        }

        return $value;
    }
}
```

Our helper file defines two functions: *value()* and *cakeenv()*. The *value()* function allows the *cakeenv()* function to get the default value from a string or a Closure, and lucky for us, closures are supported as of *>= PHP 5.3.0*. Whew!

In order to get the helper loaded, we need to add an *autoload* key to our project's *composer.json* file (Listing 7.18):

</> Listing 7.18: Autoloading the helpers.php File to composer.json (Partial)

```
"require-dev": {
    "phpunit/phpunit": "3.7.*",
    "cakephp/debug_kit": "2.2.*"
},
```

```
"autoload": {
    "files": ["app/helpers.php"]
},
"bin": [
    "lib/Cake/Console/cake"
]
```

We are ready to rebuild the image and verify our environment configurations. Remember to run *composer dump-autoload* after you add the helpers file, so the autoloader picks it up:

```
$ composer dump-autoload
$ docker-compose down
$ docker-compose up --build
```

After you are running the latest image, you can test things out by adding a *var_dump* to the top of *app/Config/core.php* (Listing 7.19):

</> Listing 7.19: Testing out Our Environment Config in app/Config/core.php

```
<?php
var_dump(cakeenv("APP_DEBUG")); exit;
```

At this point, you should see *string(1) "1"* output, which means that our *cakeenv* helper is picking up environment configuration from the *.env* file.

Basic Configuration with Environment

Now that we have the PHP dotenv library in place let's use our helper function to configure the debug setting found in *app/Config/core.php*. CakePHP accepts the following configuration for debug: 0 (production), 1 (development), or 2 (full debugging).

Update the debug line in your core.php file with the following code to configure the

debug setting from environment (Listing 7.20):

</> Listing 7.20: Using Environment for the Debug Setting

```
Configure::write('debug', (int) cakeenv('APP_DEBUG', 0));
var_dump(Configure::read('debug')); exit;
```

You should see *int(1)* if you refresh your browser. If the *APP_DEBUG* value isn't defined, it defaults to 0. Note that we have to cast the value to an *integer* because PHPDotenv doesn't convert them for you.

I would recommend updating the *.env* file and *.env.example* files to *APP_DEBUG=2* now, which is CakePHP's default value. You should see your debug settings change after tweaking your .env file.

Adding More Configuration

We have a basic example in place for using environment variables, so let's use our new helpers to go back to the CakePHP warnings found when visiting http://localhost:8888 and update them with environment variables.

Before we update our .env file, we need to generate a random salt value that we can add to our .env file (Listing 7.21):

>_ Listing 7.21: Generate a random Salt

```
$ openssl rand -base64 40
1aEimZhGcF09PGHiuGKlT8Oe3i1JeuHpwxpa19WNGJWgZlgsg0Tjeg==
```

Feel free to use the OpenSSL command like I've done, or generate the random salt in any manner that you prefer, such as mashing your keyboard.

For the security cipher, be sure that the value only includes **numbers**.

Next, grab those values and add them to *.env* (Listing 7.22):

</> Listing 7.22: Adding the Security Values to .env

```
APP_DEBUG=2
APP_SECURITY_SALT="77VqmY4DzX5UKRnYIZjeLe2RVtbc7Il3Aq5I3Y9SKxb8PDF
OVmULnA=="
APP_SECURITY_CIPHER="76490122635709458230511540876 60"
```

Next, find the configuration for *Security.salt* and *Security.cipherSeed* (they are right next to each other) in the *app/Config/core.php* file and update them with the following (Listing 7.23):

</> Listing 7.23: Using the .env Values in core.php

```
/**
 * A random string used in security hashing methods.
 */
Configure::write(
    'Security.salt',
    cakeenv(
        'APP_SECURITY_SALT',
        'DYhG93b0qyJfIxfs2guVoUubWwvniR2G0FgaC9mi'
    )
);

/**
 * A random numeric string (digits only) used to encrypt/decrypt
strings.
 */
Configure::write(
    'Security.cipherSeed',
    cakeenv(
        'APP_SECURITY_CIPHER',
        '76859309657453542496749683645'
    )
);
```

The default values we've set are the defaults that ship with CakePHP, which means that we'll have the same warnings when a new environment hasn't defined these values in an environment configuration. The warnings are helpful to guide developers just setting up an environment, and the application setup works as you'd expect.

At this point, you should set the salt and cipher environment variables in *.env.example* with the default values. New developers have the values already defined when they copy the example file, and they can change them during setup. It's also a good idea to continually keep the *.env.example* file updated as you add new values.

Database Configuration

If you look closely at *app/Config/database.php.default*, notice that the database connections are properties of the *DATABASE_CONFIG* class. We need to update it slightly because we can't set the array properties with a dynamic *cakeenv()* function.

Before we change the file, we need to copy it to *app/Config/database.php*, the filename that CakePHP expects (Listing 7.24):

Listing 7.24: Copy the Default Database Config

```
$ cp app/Config/database.php.default app/Config/database.php
```

The CakePHP project ignores */app/Config/database.php* for a good reason, but since we are using environment variables for connections, we can safely remove it from the *.gitignore* file and adapt the class to the following (Listing 7.25):

Listing 7.25: Update the database.php file with environment config

```
<?php

class DATABASE_CONFIG {
    function __construct()
    {
```

```
        $this->default = array(
            'datasource' => cakeenv(
                'DB_DATASOURCE',
                'Database/Mysql'
            ),
            'persistent' => false,
            'host' => cakeenv(
                'DB_HOST',
                'localhost'
            ),
            'login' => cakeenv(
                'DB_USER',
                'user'
            ),
            'password' => cakeenv(
                'DB_PASSWORD',
                'password'
            ),
            'database' => cakeenv(
                'DB_DATABASE',
                'database_name'
            ),
            'prefix' => cakeenv('DB_PREFIX', ''),
            // 'encoding' => cakeenv('DB_ENCODING', 'utf8'),
        );

        $this->test = array(
            'datasource' => 'Database/Mysql',
            'persistent' => false,
            'host' => 'localhost',
            'login' => 'user',
            'password' => 'password',
            'database' => 'test_database_name',
            'prefix' => '',
        );
    }
}
```

We moved the public class properties to the constructor and set them dynamically. We didn't use the environment to change the test values, but you can adjust those on your own if you'd like. I recommend using the same environment variables and overriding them with a *phpunit.xml* file. You can also create separate environment variables for your testing database connection; I'll leave that up to you.

Next, let's get a working database configuration; CakePHP is warning us that the application is not able to connect to the database. We have the class configuration in place; now we just need to update our environment with working connection information.

First, let's update the *.env.example* file with our new database connection variables that help people get started with the application (Listing 7.26):

Listing 7.26: Database .env.example Variables

```
APP_DEBUG=2
APP_SECURITY_SALT="77VqmY4DzX5UKRnYIZjeLe2RVtbc7Il3Aq5I3Y9SKxb8PDF
OVmULnA=="
APP_SECURITY_CIPHER="76490122635709458230511540876 60"

# Database
DB_DATASOURCE="Database/Mysql"
DB_HOST=localhost
DB_USER=user
DB_PASSWORD=password
DB_DATABASE=database_name
DB_PREFIX=""
```

We define the default CakePHP database values in our example file, but our Docker Compose file takes care of assigning these values automatically. The values in the *.env.example* provide an example template for other non-development environments.

We have the example environment variables defined, but we need to either define

proper connection strings in the *.env* file, or through the Docker compose file to configure the connection (Listing 7.27):

Listing 7.27: Add a MySQL Database to docker-compose.yml

```
version: "3"
services:
  app:
    image: cakephp-app
    container_name: cakephp-app
    build:
      context: .
      dockerfile: .docker/Dockerfile
    ports:
      - 8888:80
    volumes:
      - .:/srv/cakephp
    environment:
      DB_HOST: mysql
      DB_USER: root
      DB_PASSWORD: password
      DB_DATABASE: cakephp_example
  cake:
    image: cakephp-app
    container_name: cakephp-console
    volumes:
      - .:/srv/cakephp
    entrypoint: [
      "/srv/cakephp/app/Console/cake",
      "-app", "/srv/cakephp/app"
    ]
  mysql:
    image: mysql:5.5
    ports:
      - "13306:3306"
    environment:
      MYSQL_DATABASE: cakephp_example
```

```
      MYSQL_ROOT_PASSWORD: password
```

I've purposely demonstrated the *mysql:5.5* version because it's possible that your application code only supports an older version of MySQL. Being able to use different versions of software on the same development system is an excellent benefit of containers, and can also aide your effort in upgrading parts of your infrastructure.

We've set the database environment variables in the container configuration to match the MySQL environment variables for the database and root password. These environment variables get defined on the system; therefore, PHP dotenv will not mutate these values (see https://github.com/vlucas/phpdotenv#immutability). What this means, is that you can override values found in the .env file by setting system environment variables.

With our environment settings in place, new developers picking up our codebase will get a working database without needing to make any changes. Making Docker easy to use by developers is an often overlooked, but important part of using Docker. Try to make setting up Docker as convenient and comfortable in development with as little manual work as possible.

The *docker-compose.yml* file in our project is for developers; therefore, it's fine to set environment variables in the database container. You can use the *.env* file instead if you don't prefer to define environment values directly in the Compose file. I just wanted to demonstrate that you can override settings through system variables.

Verifying the Database Connection

The environment variable configuration for the database is all set, and we can finally verify that the application can connect to the database. If you rebuild the application and run your containers (you should be familiar with how to do that now), you should see the confirmation "CakePHP is able to connect to the database" if you visit http://localhost:8888 (Figure 7.3).

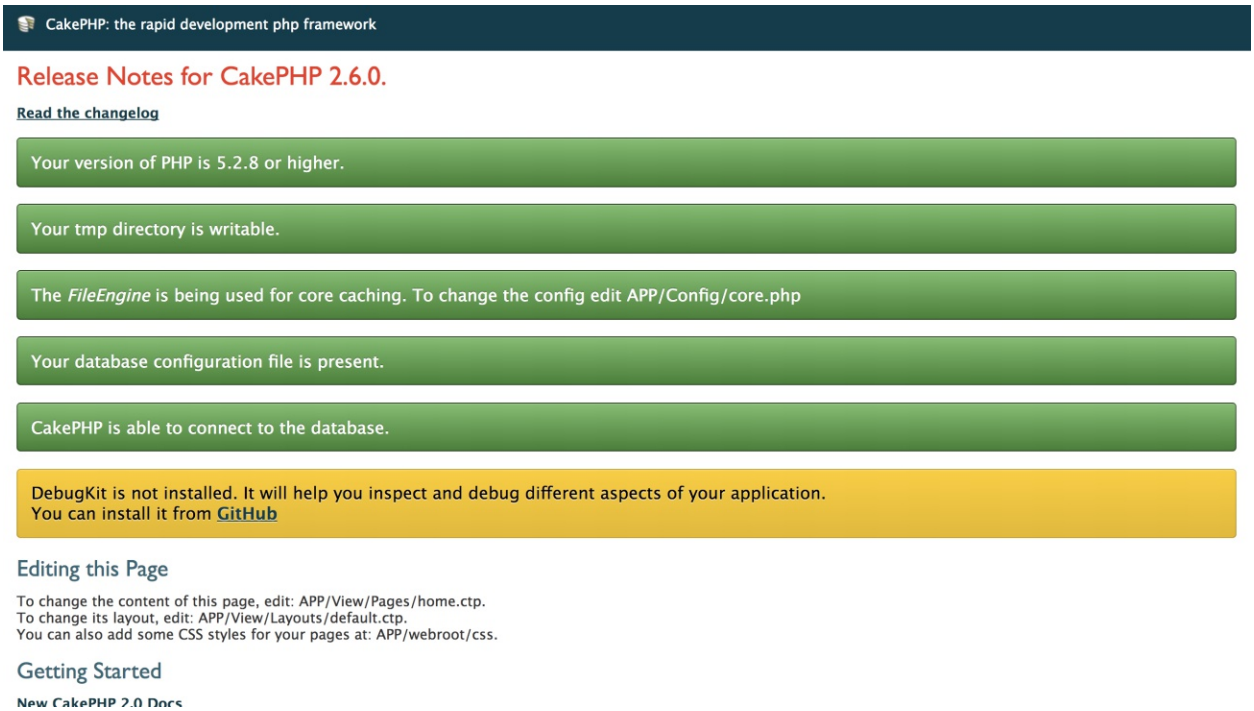


Figure 7.3: CakePHP is able to connect to the database

Suhosin

PHP continues to patch security vulnerabilities until a version end of life (EOL), which is one reason why a legacy PHP application running an EOL version of PHP can become a growing liability. That's one reason why I've shown you the "most secure" version of PHP 5.3 I can think of, with the Canonical patches.

Suhosin describes itself as "an advanced protection system for PHP installations." Suhosin is another essential part of protecting an older version of PHP, and ships with smart defaults, including a simulation mode that helps learn how your application breaks while running Suhosin. We'll cover simulation mode in more detail later in this section.

There are a few common things you'll need to deal with to get your application to work with Suhosin, so let's go over them real quick.

Running PHAR Files

As part of a security measure, Suhosin will not allow you to run phar (PHP Archive)

files. If you run the application container and try to install Composer, for example, you will get a similar error (Listing 7.28):

Listing 7.28: Trying to Run a .phar File

```
$ docker-compose down
$ docker-compose run --rm app /bin/bash
Creating network "cakephp260_default" with the default driver

# Inside the container
$ apt-get -y install curl
$ curl -S https://getcomposer.org/installer | php -- \
    --install-dir=/usr/local/bin \
    --filename=composer \
  && chmod +x /usr/local/bin/composer

Some settings on your machine make Composer unable to work
properly.
Make sure that you fix the issues listed below and run this
script again:

The value for `suhosin.executor.include.whitelist` is incorrect.
Add the following to the end of your `php.ini` or suhosin.ini
(Example path [for Debian]: /etc/php5/cli/conf.d/suhosin.ini):
    suhosin.executor.include.whitelist = phar
# ...

# Exit the container
$ exit
```

It's pretty clear that we need to make an INI file change. You can either update the *php.ini* file or *suhosin.ini* file, but let's just create a separate configuration file just for the CLI.

You can organize your INI files however you want; the following is just one way of many. Create an *app.ini* file from your local machine that we'll copy into the image

(Listing 7.29):

Listing 7.29: Create the app.ini file for the CLI

```
$ mkdir -p .docker/php/cli/
$ touch .docker/php/cli/app.ini
```

Add the following configuration to the newly created *app.ini* file (Listing 7.30):

Listing 7.30: Enable .phar on the CLI

```
; Application PHP settings

; Allow phar files
suhosin.executor.include.whitelist = phar
```

Last, copy the new file into the container (Listing 7.31):

Listing 7.31: Copy the app.ini file

```
FROM ubuntu:12.04

LABEL maintainer="Paul Redmond"

RUN apt-get -yqq update \
    && apt-get -yqq install \
        apache2 \
        libapache2-mod-php5 \
        php5 \
        php5-mysql \
        php5-mcrypt \
        php5-suhosin \
    && rm -f /etc/apache2/sites-available/* \
    && rm -f /etc/apache2/sites-enabled/* \
    && a2enmod rewrite

COPY .docker/httpd-foreground /usr/local/bin/
```

```
COPY .docker/vhost.conf /etc/apache2/sites-available/000-
default.conf
# Copy INI files for the command line
COPY .docker/php/cli/*.ini /etc/php5/cli/conf.d
COPY . /srv/cakephp

RUN ln -s /etc/apache2/sites-available/000-default.conf \
        /etc/apache2/sites-enabled/000-default.conf \
    && chmod +x /usr/local/bin/httpd-foreground \
    && chown -R www-data:www-data /srv/cakephp

WORKDIR /srv/cakephp

EXPOSE 80

CMD ["httpd-foreground"]
```

If you rebuild the image and run the container, you should be able to download and execute the *composer.phar* file from the CLI like we tried in Listing 7.28:

```
$ docker-compose build app
$ docker-compose run --rm app /bin/bash

# Inside the container
$ apt-get -y install curl
$ curl -S https://getcomposer.org/installer | php -- \
    --install-dir=/usr/local/bin \
    --filename=composer \
  && chmod +x /usr/local/bin/composer
# ...
All settings correct for using Composer
Downloading...

Composer (version 1.5.2) successfully installed to:
/usr/local/bin/composer
Use it: php /usr/local/bin/composer
```

We just configured the Suhosin to whitelist the *phar* stream wrapper with Suhosin, which allows us to run composer commands from the CLI environment. The executor error is one example of how Suhosin locks down your PHP environment, which is a good thing when working with EOL versions of PHP.

Simulating Suhosin

You might run into other issues that prevent your application from working with Suhosin. Some of these matters might be that your application is using functionality that Suhosin prevents due to security risks.

One thing you can do is run your application with the *suhosin.simulation = On* setting. The description of the simulation setting (https://suhosin.org/stories/configuration.html#suhosin-simulation) is defined as follows:

> *If you fear that Suhosin breaks your application, you can activate Suhosin's simulation mode with this flag. When Suhosin runs in simulation mode, violations are logged as usual, but nothing is blocked or removed from the request.*

Using simulation mode is an excellent way to run your application and collect violations so that you can address them. If your application doesn't work with Suhosin, I would suggest reading through the extensive configuration options to at least get as much security as possible without disabling Suhosin completely. The simulation might reveal some critical security fixes that you should address.

The Born Legacy

This chapter was chalk-full of examples, and it was vital that we go through more than just getting Docker running. Porting legacy applications to Docker requires thinking about configuration differently. When working with an older application, be prepared to deal with issues unique to your codebase and level of technical debt. Be patient in getting Docker working with legacy systems because years of bad decisions start to rear their ugly head when trying to shift things into Docker.

Another consideration we didn't cover in this chapter is writing to the file system. You need to adjust your application to either start uploading files to Amazon S3 for example or persist your uploads to a host machine with a volume.

I want to reiterate how important you should treat upgrading your applications. Sometimes upgrading can be a daunting process, but building a consistent environment is an excellent way to get the process in motion. Hopefully, I've provided enough tools to help you get your legacy applications running Docker so you can more easily replicate the environment and make it a little more portable between environments.

Chapter 8: Custom Commands

In this chapter, we'll look more in depth at how we can customize the way our PHP containers start up and run. Up to this point, we have mostly been relying on the official PHP Docker image to run our containers. Under the hood, however, the PHP image we extend from is running either Apache or PHP-FPM.

The way that we can do this in our Dockerfile is by defining a CMD instruction—which you caught a glimpse of in the last chapter. If you view the source of the official php-fpm Dockerfile you will notice the instruction *CMD ["php-fpm"]* at the bottom. Our Dockerfiles have been inheriting this CMD instruction, but we can define our own to override it.

According to the official Docker CMD documentation:

> *The main purpose of a CMD is to provide defaults for an executing container. These defaults can include an executable, or they can omit the executable, in which case you must specify an ENTRYPOINT instruction as well.*

We won't go into how the CMD and ENTRYPOINT interact in this text, but you can learn more about CMD, ENTRYPOINT, and all the other instructions in the Dockerfile reference documentation (https://docs.docker.com/engine/reference/builder/).

For our purposes, we are going to create a custom bash executable that allows us more customization in running our applications in Docker. To demonstrate, we are going to use a CLI program Confd (https://git.io/nCjQ3w) to manage application configuration

files to provide some configuration setup before running a web server. For the web server we use Caddy for the web server, which I introduced in Chapter 6.

Introduction to Confd

Confd is a lightweight configuration management tool that allows you to keep configuration files up to date from data stored in backends like environment variables, Consul (https://www.consul.io/), Etcd (https://github.com/coreos/etcd), Redis (https://redis.io/), and others. You can also reload applications to pick up changes during runtime without restarting the container. With Confd, we can separate our configuration management from infrastructure code.

To start out, we are going to use environment variables with Confd to simplify the example and then improve upon it using Consul.

Confd Quickstart

I recommend going through the quick start guide (https://git.io/vbrK8) to get an overview of setting up Confd. We cover the basics in this chapter, but the overview is highly recommended reading.

Let's get started by creating a project for our work (Listing 8.1):

Listing 8.1: Creating the Project

```
$ mkdir -p ~/Code/ch8-custom-commands
$ cd $_
$ touch Dockerfile docker-compose.yml start.sh index.php
$ mkdir -p confd-configs/{conf.d,templates}
$ touch confd-configs/conf.d/caddyfile.toml
$ touch confd-configs/templates/caddyfile.tmpl
```

We've created our typical Docker files and Confd configuration files and templates.

Lastly, we created the *start.sh* file that serves as our custom CMD command in our Dockerfile.

Installing Confd

Let's start out by installing the *confd* binary in our image (Listing 8.2):

</> Listing 8.2: Installing Confd

```
FROM php:7.1-fpm

ENV CADDY_HOSTNAME=0.0.0.0
ADD
https://github.com/kelseyhightower/confd/releases/download/v0.11.
0/confd-0.11.0-linux-amd64 /usr/local/bin/confd

RUN chmod +x /usr/local/bin/confd \
    && mkdir -p /etc/confd/conf.d /etc/confd/templates
```

Listing 8.2 is the first example we've shown using a URI with the ADD instruction. We link to the Confd binary for Linux (64 bit) in the first argument, and the second argument is the path to which we want to add Confd. Last, we make confd executable and create the *conf.d* and *templates* folders which houses our Confd configuration and template files.

Let's build the image and take Confd for a spin (Listing 8.3):

>_ Listing 8.3: Building the Docker Image

```
$ docker build -t custom-commands .

$ docker run --rm -it custom-commands bash
root@30f95181fed3:/var/www/html# confd --version
confd 0.11.0
```

Confd Templates

Let's add a template that we can use with Confd so you can see how it works. To demonstrate, let's generate a Caddyfile template that can be dynamically changed with configuration—even during runtime.

Enter the following in a new file created at *confd-configs/conf.d/caddyfile.toml* in your project (Listing 8.4):

</> Listing 8.4: Create the TOML configuration file for Caddy

```
[template]
src = "caddyfile.tmpl"
dest = "/etc/Caddyfile"
owner = "www-data"
mode = "0644"
```

We created a TOML (https://github.com/toml-lang/toml) file, which is a configuration file that describes the actual template-generated file. The destination file is generated from the *src = "caddyfile.tmpl"*. The *owner* and *mode* set the owner of the source file and the file permissions.

So what is TOML?

On the TOML Github page provides the following description:

> *TOML aims to be a minimal configuration file format that's easy to read due to obvious semantics. TOML is designed to map unambiguously to a hash table. TOML should be easy to parse into data structures in a wide variety of languages.*

Next, we need the create *caddyfile.tmpl* as referenced in the TOML configuration, which is used to generate our web server Caddyfile. Confd template files are golang text templates (https://golang.org/pkg/text/template/#pkg-overview) using variables that make the template dynamic based on configuration.

We've already seen an example of a Caddyfile in Chapter 6, so the following Caddyfile template should look familiar. Open the file we created in Listing 8.1, *confd-configs/templates/caddyfile.tmpl*, and add the following (Listing 8.5):

</> Listing 8.5: Editing the caddyfile.tmpl

```
http://{{getenv "CADDY_HOSTNAME"}}:80
root /srv/app/public
gzip
fastcgi / 127.0.0.1:9000 php
rewrite {
    regexp .*
    ext /
    to /index.php?{query}
}

log stdout
errors stdout
on startup php-fpm --nodaemonize
```

The first line is the *getenv* function, which replaces everything within the double curly braces (*{{ }}*) with the value of the *CADDY_HOSTNAME* environment variable. The rest is pretty much the same Caddyfile we used in Chapter 6.

Before we test out our new files, we need to copy them into the container by updating the Dockerfile (Listing 8.6):

</> Listing 8.6: Copy the Confd Config and Template Files

```
FROM php:7.1-fpm

ENV CADDY_HOSTNAME=0.0.0.0
ADD
https://github.com/kelseyhightower/confd/releases/download/v0.11.
0/confd-0.11.0-linux-amd64 /usr/local/bin/confd
```

```
RUN chmod +x /usr/local/bin/confd \
    && mkdir -p /etc/confd/conf.d /etc/confd/templates

COPY confd-configs/conf.d/ /etc/confd/conf.d/
COPY confd-configs/templates/ /etc/confd/templates/
```

We added two lines copying the new Confd configuration files into their respective folders within the image. The *src = "caddyfile.tmpl"* line looks in */etc/confd/templates* for the file when generating the file by default. Also, note the *ENV CADDY_HOSTNAME=0.0.0.0* defined in the Dockerfile, which is the default environment value our confd template uses when creating the Caddyfile.

Let's build the latest image and jump back into our container so we can experiment with our new changes (Listing 8.7):

Listing 8.7: Updating the Image and Testing the Changes

```
$ docker build -t custom-commands .
$ docker run --rm -it custom-commands bash

# Run Confd to apply the changes
$ confd -onetime -backend env
2017-07-04T17:16:55Z 6b6cd93aa161 confd[15]: INFO Backend set to
env
2017-07-04T17:16:55Z 6b6cd93aa161 confd[15]: INFO Starting confd
2017-07-04T17:16:55Z 6b6cd93aa161 confd[15]: INFO Backend nodes
set to
2017-07-04T17:16:55Z 6b6cd93aa161 confd[15]: INFO Target config
/etc/Caddyfile out of sync
2017-07-04T17:16:55Z 6b6cd93aa161 confd[15]: INFO Target config
/etc/Caddyfile has been updated

$ cat /etc/Caddyfile
http://0.0.0.0:80
root /srv/app/public
```

```
gzip
fastcgi / 127.0.0.1:9000 php
rewrite {
    regexp .*
    ext /
    to /index.php?{query}
}

log stdout
errors stdout
on startup php-fpm --nodaemonize
```

By running confd once with the *-backend env*, we are using environment variables to populate our template. If you exit the image and jump back in, notice that the */etc/Caddyfile* is gone, because of the ephemeral nature of Docker containers.

While still in the image, let's change the environment variable and see what happens (Listing 8.8):

Listing 8.8: Changing the Environment Variable and Running Again

```
$ export CADDY_HOSTNAME=foo.com

$ confd -onetime -backend env
INFO Backend set to env
INFO Starting confd
INFO Backend nodes set to
INFO /etc/Caddyfile
has md5sum a4b9b46b6130bc9ec6123361eb0452c9
should be b393c5c1e80f466749e4797a2ebd71a5
INFO Target config /etc/Caddyfile out of sync
INFO Target config /etc/Caddyfile has been updated

$ cat /etc/Caddyfile
http://foo.com:80
```

```
root /srv/app/public
gzip
fastcgi / 127.0.0.1:9000 php
rewrite {
    regexp .*
    ext /
    to /index.php?{query}
}

log stdout
errors stdout
on startup php-fpm --nodaemonize
```

As you can see, we can separate our configuration management (infrastructure config) from infrastructure code. We've used environment variables to make the example simple, but we'll expand on that later with a backend like Consul where we poll for changes and apply them automatically while the container is running.

Before we expand on our Confd usage, let's shift focus to the goal of this chapter: providing a custom CMD instruction when we need a little more advanced strategy in running our containers.

The Custom CMD

Now that we have our Confd templates in place, it's time to automate running them when the container starts. For this to work, our bash script needs to run a process in the foreground because as soon as the main process exits, our container stops.

Let's first fill in the start.sh file that we created in Listing 8.1 with our Confd command so you can see what happens to the container when our process exits (Listing 8.9):

</> Listing 8.9: The custom start.sh File

```
#!/usr/bin/env bash

set -e

confd -onetime -backend env
```

We just run the same Confd command we ran earlier in the new bash script during container startup.

Next, let's update the Dockerfile in order to get the start.sh file into the image and define it as the CMD instruction (Listing 8.10):

</> Listing 8.10: Copy the start.sh File and Define a CMD

```
FROM php:7.1-fpm

ENV CADDY_HOSTNAME=0.0.0.0
ADD
https://github.com/kelseyhightower/confd/releases/download/v0.11.
0/confd-0.11.0-linux-amd64 /usr/local/bin/confd

RUN chmod +x /usr/local/bin/confd \
    && mkdir -p /etc/confd/conf.d /etc/confd/templates

COPY confd-configs/conf.d/ /etc/confd/conf.d/
COPY confd-configs/templates/ /etc/confd/templates/
COPY start.sh /usr/local/bin/start.sh

RUN chmod +x /usr/local/bin/start.sh

CMD ["/usr/local/bin/start.sh"]
```

We copy the *start.sh* script into */usr/local/bin*, make it executable and reference it in the CMD instruction. If we don't make it executable you will get an error like

"exec: /usr/local/bin/start.sh: Permission denied."

Let's try to run a container with our updates (Listing 8.11):

>_ Listing 8.11: Run the Container with the new start.sh CMD

```
$ docker build -t custom-commands .

$ docker run -it custom-commands
INFO Backend set to env
INFO Starting confd
INFO Backend nodes set to
INFO Target config /etc/Caddyfile out of sync
INFO Target config /etc/Caddyfile has been updated

# The container exits and brings us back to our prompt
$
```

If you run *docker ps -a* you should see that the container has exited.

Let's update our start.sh file to keep a process running in the foreground so we can demonstrate how our custom CMD script works. The current state of our script is just for demonstration purposes, but eventually, we run the *caddy* binary process in the foreground (Listing 8.12):

</> Listing 8.12: Update the start.sh to Keep a Process Running

```
#!/usr/bin/env bash

set -e

confd -onetime -backend env

trap : TERM INT; sleep infinity & wait
```

We are running sleep infinitely and then exiting on an interrupt signal. As I mentioned,

this is just for our debugging purposes, and you wouldn't run a container like this in a real project. However, it's a helpful tool for debugging purposes.

Let's rerun the container and verify that the confd command applies the changes (Listing 8.13):

Listing 8.13: Running the Container with an Infinite Sleep

```
$ docker build -t custom-commands .

$ docker run --rm -it custom-commands
INFO Backend set to env
INFO Starting confd
INFO Backend nodes set to
INFO Target config /etc/Caddyfile out of sync
INFO Target config /etc/Caddyfile has been updated
```

Our container is running now because of the *sleep* process in the foreground, so open a new terminal window or tab and run the following to verify that our script applied the changes to the Caddyfile (Listing 8.14):

Listing 8.14: Verify the Confd Changes from start.sh

```
$ docker ps # get the container id
$ docker exec -it c617831ae756 cat /etc/Caddyfile
http://0.0.0.0:80
root /srv/app/public
gzip
fastcgi / 127.0.0.1:9000 php
rewrite {
    regexp .*
    ext /
    to /index.php?{query}
}

log stdout
```

```
errors stdout
on startup php-fpm --nodaemonize
```

When the container starts, our Confd command creates the template from the passed environment configuration, and we learn how to keep the container running with our sleeping foreground process. If you get stuck, the sleep technique is an excellent way to debug a custom CMD.

Running Caddy

We are in a position to swap the sleep command with a Caddy process running in the foreground. Our goal is to run Caddy using the generated Caddyfile based on environment configuration.

First, let's install Caddy and update a few other things in the Dockerfile (Listing 8.15):

</> Listing 8.15: Update the Dockerfile with Caddy

```
FROM php:7.1-fpm

ENV CADDY_HOSTNAME=0.0.0.0
ADD
https://github.com/kelseyhightower/confd/releases/download/v0.11.
0/confd-0.11.0-linux-amd64 /usr/local/bin/confd

RUN chmod +x /usr/local/bin/confd \
    && mkdir -p /etc/confd/conf.d /etc/confd/templates

RUN curl --silent --show-error --fail --location \
      --header "Accept: application/tar+gzip, application/x-gzip,
application/octet-stream" -o - \

"https://caddyserver.com/download/linux/amd64?plugins=http.expire
s,http.realip&license=personal" \
    | tar --no-same-owner -C /usr/bin/ -xz caddy \
```

```
    && chmod 0755 /usr/bin/caddy \
    && /usr/bin/caddy -version \
    && docker-php-ext-install mbstring pdo pdo_mysql

COPY confd-configs/conf.d/ /etc/confd/conf.d/
COPY confd-configs/templates/ /etc/confd/templates/
COPY start.sh /usr/local/bin/start.sh
COPY index.php /srv/app/public/index.php

RUN chmod +x /usr/local/bin/start.sh \
    && chown -R www-data:www-data /srv/app

EXPOSE 80

CMD ["/usr/local/bin/start.sh"]
```

Just like Chapter 6, we install the *caddy* binary and some PHP modules. The next addition is copying an index.php file into the image that is served by Caddy. We change ownership of the */srv/app* path to the *www-data* user and expose port 80.

The contents of the index.php file is just *<?php phpinfo(); ?>* in the root of your project for this chapter. It doesn't matter what is running—we are focused on the startup script in this chapter—but the PHP info is a helpful indicator.

In order to get Caddy working, next we need to replace the infinite loop with the caddy executable in our *start.sh* script (Listing 8.16):

</> Listing 8.16: Adding Caddy to the start.sh File

```
#!/usr/bin/env bash

set -e

confd -onetime -backend env
```

```
/usr/bin/caddy -validate --agree=true --conf=/etc/Caddyfile

exec /usr/bin/caddy --agree=true --conf=/etc/Caddyfile
```

We added two lines to the end of the file:

1. A validation check of the generated Caddyfile that ensures our configuration is valid
2. Running caddy in the foreground with our generated configuration

If there's a validation error, start.sh will exit, otherwise, Caddy will run in the foreground. The *exec* command replaces the current process without forking a new process.

Running the Container

Everything is in place to test out our custom CMD script with Caddy. Let's build the image and then execute the container with the *docker run* command (Listing 8.17):

Listing 8.14: Verify the Confd Changes from start.sh

```
# Build the image
$ docker build -t custom-commands .

# Run the container
$ docker run --rm -it -p 8080:80 custom-commands
INFO Backend set to env
INFO Starting confd
INFO Backend nodes set to
INFO Target config /etc/Caddyfile out of sync
INFO Target config /etc/Caddyfile has been updated
Caddyfile is valid
Activating privacy features... done.
http://0.0.0.0
NOTICE: fpm is running, pid 25
NOTICE: ready to handle connections
```

I've shortened the output a little, but you can see that start.sh runs *confd* to generate the */etc/Caddyfile* file, validates the Caddyfile, and then runs caddy. The Caddyfile defines http://0.0.0.0:80 so our *docker run* command maps port 80 to 8080. If you open http://localhost:8080 in your browser, you should see the *phpinfo()* output (Figure 8.1):

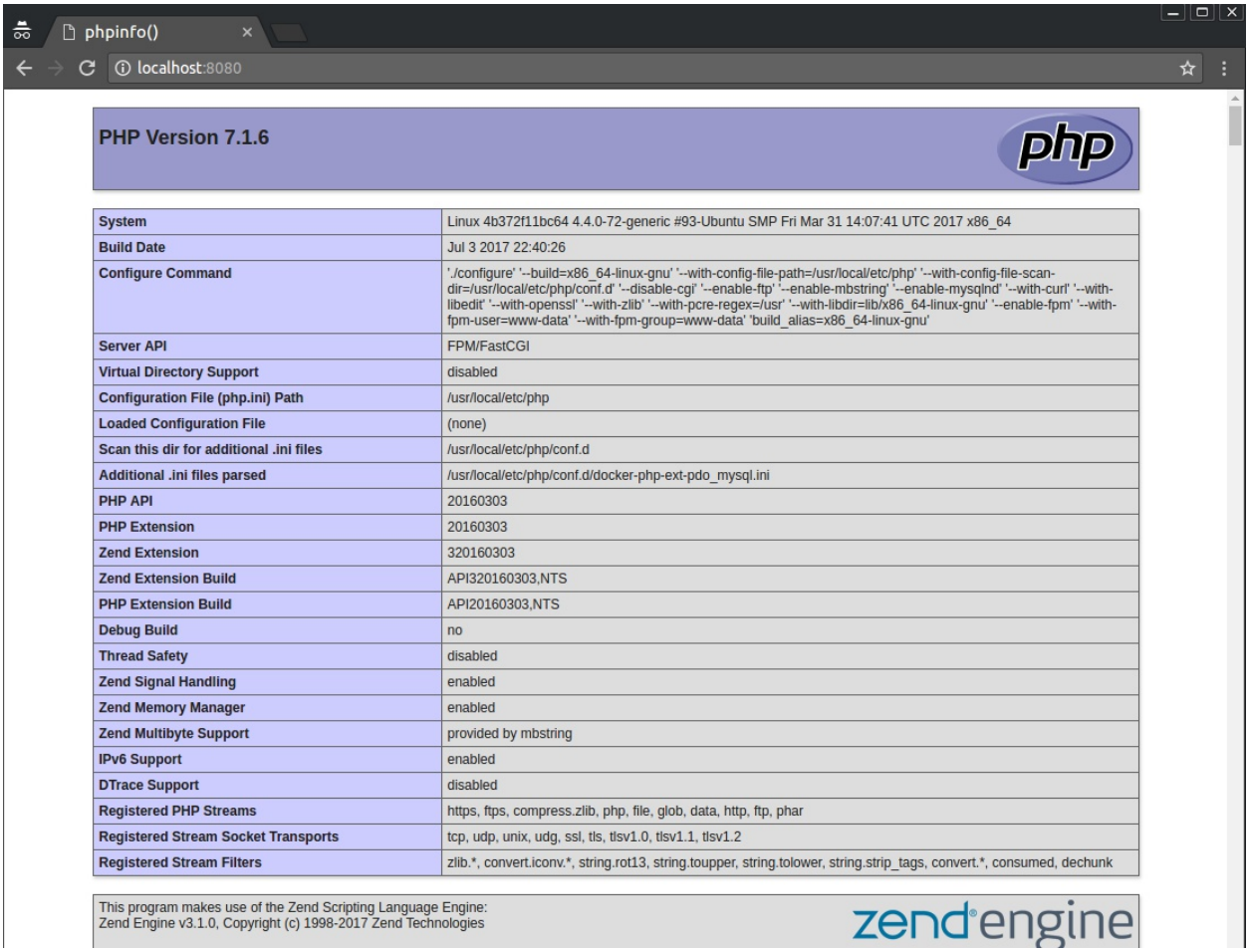


Figure 8.1: PHP Info Output

If you change the CADDY_HOSTNAME environment variable when you run *docker run* you can dynamically change the hostname defined in the Caddyfile:

```
$ docker run \
  -e "CADDY_HOSTNAME=example.dev" \
  --rm -it -p 8080:80 \
  custom-commands
```

We've successfully created a custom CMD script and learned about Confd in the process. I think you can see the power Confd brings to our setup, allowing us to change

the hostname(s) in the Caddyfile based on environment. Let's improve upon our Confd implementation adding in a different backend.

Confd With a Consul Backend

Now that we have a working CMD script let's expand on the Confd setup with Consul. We are going to swap out our Confd backend with consul's key/value storage (https://www.consul.io/intro/getting-started/kv.html).

You might be familiar with Consul's service discovery and integrated health checking, but in addition to those features, Consul also provides an easy-to-use key-value store.

We are going to run consul in a development mode so that we can demonstrate how to hot-swap a configuration in Confd and restart Caddy after the configuration is updated—all without restarting the container.

Our Dockerfile doesn't need to be updated, but we are going to make small changes to our Confd template and config files, and add in a *docker-compose.yml* file so we can run the official consul (https://hub.docker.com/_/consul/) Docker image and link containers more efficiently.

Let's get to work!

Docker Compose File

Docker Hub provides an official Consul image, which we can define in our Docker Compose file. Create the *docker-compose.yml* file if you haven't already and let's get our services going (Listing 8.18).

Listing 8.18: The docker-compose.yml file

```
version: "3"
services:
  app:
```

```
    build: .
    ports:
      - 8080:80
  consul:
    image: consul:1.0.1
    ports:
      - 8500:8500
```

We are mapping port 80 to 8888 because our Caddyfile template uses port 80 instead of 2015 (the default). We also defined a Consul service and map port 8500 so we can make requests to Consul locally on port 8500.

Consul Backend

We need to wire up our startup script to use the Consul backend instead of environment variables. Because we are starting up a fresh consul container, our Confd command cannot succeed until we define the keys on which the template is dependent. Thus, we need to wait until the Confd command succeeds before starting Caddy process (Listing 8.19):

Listing 8.19: Changing the Confd Backend to Consul in start.sh

```
#!/usr/bin/env bash

set -e

function shutdown {
  kill -s SIGTERM $CONFD_PID
  wait $CONFD_PID
  kill -s SIGTERM $PHP_FPM_PID
  wait $PHP_FPM_PID
}

# Wait until the initial configuration succeeds
until confd -onetime -backend consul -node consul:8500; do
```

```
    echo "Waiting for the initial confd configuration"
    sleep 5
done

confd -interval 10 -backend consul -node consul:8500 &
CONFD_PID=$!

php-fpm &
PHP_FPM_PID=$!

trap shutdown SIGTERM SIGINT

/usr/bin/caddy -validate --agree=true --conf=/etc/Caddyfile

exec /usr/bin/caddy --agree=true --conf=/etc/Caddyfile
```

We introduced a few things here, in fact, most lines have been updated. We define a shutdown function so that we can properly kill the Confd and php-fpm background processes running when an interrupt signal is sent (Ctrl+C). Next, we run the *confd -onetime* command every 5 seconds until it succeeds. Our script isn't perfect, but you can expand upon it and exit on "X" number of retries if you prefer.

The Confd command references *consul:8500*, which is how the application container makes requests to the Consul service on the network, but this could be configurable via environment in a similar way that we defined environment on *docker run* to change the Caddyfile.

Once Confd succeeds, we run it in the background and poll consul every 10 seconds (*-interval 10*). When the Consul backend is updated, Confd detects the changes and regenerates the Caddyfile.

Revisiting the Confd Config

Now that start.sh updates the Caddyfile on a value change in Consul, we need a way to restart Caddy after a template update. You might have experience with Apache or Nginx

requiring a restart when configuration changes and Caddy has a similar feature.

The way we trigger a change with Confd is with the *reload_cmd* property in the TOML file. When Confd reloads the template, the *reload_cmd* sends the proper signal to Caddy for a restart. We also need to add a *keys=* config which matches the consul key/value store (Listing 8.20):

</> Listing 8.4: Create the TOML configuration file for Caddy

```
[template]
src = "caddyfile.tmpl"
dest = "/etc/Caddyfile"
owner = "www-data"
mode = "0644"
keys = [
  "/example.com/hostname",
]
reload_cmd = "pkill -USR1 caddy"
```

The *keys=* config is defined as an array of template keys that match the path in Consul, or any other backend store that you use, except environment variables.

The *reload_cmd* configuration is triggered when the template is updated; in our case when Confd polls Consul, notices a change, and updates the Caddyfile.

To restart Caddy, we need to send the *USR1* signal to the running caddy process using *pkill* to reference the process by name.

Revisiting the Confd Template

Using the Consul backend with Confd means that we need to update our template to use the *getv* function instead of *getenv*, which is specific to using environment variables. The other Confd backends use *getv*, so you could change out Consul for another backend, and the template wouldn't change.

Let's update our *confd-configs/templates/caddyfile.tmpl* file to use *getv* with the new consul key, and also remove the *on startup php-fpm* event (Listing 8.21):

Listing 8.21: Update the caddyfile.tmpl with the New Key Reference

```
http://{{getv "/example.com/hostname"}}:80
root /srv/app/public
gzip
fastcgi / 127.0.0.1:9000 php
rewrite {
    regexp .*
    ext /
    to /index.php?{query}
}

log stdout
errors stdout
```

We needed to remove the *on startup php-fpm* event because when we send the restart signal to Caddy, it tries to start php-fpm again. We avoid this by just starting php-fpm in the *start.sh* file. You can keep the startup script if you'd like instead of moving *php-fpm* to the script, however, you get warnings each time Caddy restarts that php-fpm can't start because it's already running and bound to port 9000.

Putting it All Together

We are ready to start using the Consul backend in a running container with Docker Compose. When we first run the containers, the application cannot find the consul key, so the CMD script runs the *until* loop until it running confd succeeds.

Let's run the container and manually populate consul to get the container running (Listing 8.22):

Listing 8.22: Building the Image and Running the Containers

```
$ docker-compose up --build

app_1: INFO Backend set to consul
app_1: INFO Starting confd
app_1: INFO Backend nodes set to consul:8500
app_1: ERROR Get
http://consul:8500/v1/kv/example.com/hostname?recurse=: dial tcp
172.21.0.2:8500: connection refused
app_1: Waiting for the initial confd configuration
```

You should notice some error output and our CMD script output that it's waiting for an initial valid Confd configuration. The loop continues to run until we populate Consul with the template keys.

Let's use *curl* to create the keys and values utilized in the template. Open another terminal tab and make the following curl request (Listing 8.23):

Listing 8.23: Create a hostname key in Consul

```
$ curl -X PUT -d 'www.example.com' \
  http://localhost:8500/v1/kv/example.com/hostname
true
```

You should be able to see and edit the value from the UI after running the *curl* command above (http://localhost:8500/ui/#/dc1/kv/example.com/hostname/edit). Once you run the command you should also see something similar to the following in your Docker logs (Listing 8.24):

Listing 8.24: Confd Getting the Configuration from Consul and Starting Caddy

```
app_1: INFO Backend set to consul
app_1: INFO Starting confd
app_1: INFO Backend nodes set to consul:8500
```

```
app_1: INFO /etc/Caddyfile has md5sum
7c1c08a6be1130f9a3ff0bb16d9126ce should be
bc4277f5647f438ec22c9c142288bc7f
app_1: INFO Target config /etc/Caddyfile out of sync
app_1: Waiting for the initial confd configuration
app_1: 2017-07-05T06:17:26Z e22a6dae61ce confd[269]: INFO Backend
set to consul
app_1: 2017-07-05T06:17:26Z e22a6dae61ce confd[269]: INFO
Starting confd
app_1: 2017-07-05T06:17:26Z e22a6dae61ce confd[269]: INFO Backend
nodes set to consul:8500
app_1: Caddyfile is valid
app_1: http://www.example.com
app_1: [05-Jul-2017 06:17:26] NOTICE: fpm is running, pid 290
app_1: [05-Jul-2017 06:17:26] NOTICE: ready to handle connections
```

If you update your hosts file and point www.example.com to 127.0.0.1, you should be able to see your *phpinfo()* screen when you request http://www.example.com:8080.

If you update the key with another value via a *curl* request (or through the UI) you should see the Docker logs output that the */etc/Caddyfile* file has been updated (Listing 8.25):

Listing 8.25: Updating the Consul Key to Change the Caddyfile

```
$ curl -X PUT -d 'foo.com' \
  http://localhost:8500/v1/kv/example.com/hostname
true

# You should see something similar to the
# following output for the app container...
app_1: INFO /etc/Caddyfile has md5sum
bc4277f5647f438ec22c9c142288bc7f should be
b393c5c1e80f466749e4797a2ebd71a5
app_1: INFO Target config /etc/Caddyfile out of sync
```

```
app_1: INFO Target config /etc/Caddyfile has been updated
```

Now if you request http://www.example.com:8080 you should get the following response from caddy: "404 Site www.example.com:8080 is not served on this interface". Note, that this only works if you update your hosts file to make www.example.com point to 127.0.0.1.

We now have a pretty robust configuration management tool inside of Docker. Granted, we wouldn't want to hot-swap the hostname of an application regularly, but I believe you can see the power of separating the configuration from infrastructure code. We have a flexible template system that allows configuration changes without needing to update infrastructure code in some cases.

One of a Kind

We just dove into customizing the CMD instruction in Docker with a bash script. We just scratched the surface, but don't get too carried away with bash scripts for CMD and ENTRYPOINT. Keep it as simple as possible!

Our next step is learning how to share Docker images with others using Docker registries, and automatically building those images when changes get pushed to a git repository. Onward!

Chapter 9: Docker Registry

A Docker registry is what we have been using to pull down official Docker images like MySQL and PHP. We've already been interacting with a Docker registry transparently: Docker Hub. This registry is the default when you don't specify a registry explicitly. While you can run your own Docker registry, this chapter is about using existing Docker registries to host your images, not running your own. Docker has a zero maintenance, hosted solution which includes free unlimited public repositories, and a paid version if you need to host multiple private images.

At a basic level, think of a Docker registry (https://docs.docker.com/registry/introduction/) as the following:

> *A registry is a storage and content delivery system, holding named Docker images, available in different tagged versions.*

When you run *docker build -t my-image .* you are tagging the image as *my-image:latest*. If you run *docker build -t my-image:1.0.0*, you are tagging *1.0.0*. You have been using Docker image tags in the Dockerfile already: when we use *FROM php:7.1-fpm* we are using the *7.1-fpm* tag.

Let's dive deeper and learn how to interact with registries, push our images to them, and automate the process.

Setting Up a Repository and Project

If you want to follow along in this chapter, you need to register for a Docker account

(https://cloud.docker.com/). We also look at GitLab's private registry so you can learn how to use other registries to host your images, so register for a GitLab account too.

This chapter focuses more on interacting with registries than the actual Docker image, but eventually, you need to host your images somewhere so you can work with them and share them with your team privately. You might even build a base image from which your projects extend, and your projects benefit from sharing common image functionality, for example, installing the base set of PHP modules.

To demonstrate how to create images we need to create a simple Dockerfile. We make a simple PHP-FPM image and continue to use Caddy as the web server, using the same Caddy installation we've already been using, but we re-create the files here, so you don't have to go back and reference them.

First, let's set up the skeleton files for the project folder (Listing 9.1):

>_ Listing 9.1: Setting up the Project

```
$ mkdir -p ~/Code/ch9-php-caddy
$ cd $_
$ touch Dockerfile Caddyfile
```

Next, let's define the Dockerfile (Listing 9.2):

Listing 9.2: The Dockerfile

```
FROM php:7.1-fpm
LABEL maintainer="Paul Redmond"

RUN curl --silent --show-error --fail --location \
      --header "Accept: application/tar+gzip, application/x-gzip,
application/octet-stream" -o - \

"https://caddyserver.com/download/linux/amd64?plugins=http.expire
s,http.realip&license=personal" \
    | tar --no-same-owner -C /usr/bin/ -xz caddy \
    && chmod 0755 /usr/bin/caddy \
    && /usr/bin/caddy -version \
    && docker-php-ext-install mbstring pdo pdo_mysql

COPY Caddyfile /etc/Caddyfile
WORKDIR /srv/app/
RUN chown -R www-data:www-data /srv/app

CMD ["/usr/bin/caddy", "--conf", "/etc/Caddyfile", "--log",
"stdout"]
```

Finally, we define the Caddyfile configuration as follows (Listing 9.3):

Listing 9.3: The Caddyfile

```
0.0.0.0
root /srv/app/public
gzip
fastcgi / 127.0.0.1:9000 php
rewrite {
    regexp .*
    ext /
    to /index.php?{query}
}
```

```
header / -Server

log stdout
errors stdout
on startup php-fpm --nodaemonize
```

Our project could become a stand-alone repository that you use as the foundation. In fact, later in this chapter, we consume our image in a different project as an example so you can see how to extend your images. I use Caddy for most of my applications, so it makes sense to extract my base setup into an image that I extend in my projects.

To push images to Docker Cloud you need to create a repository after you login to Docker Cloud (https://cloud.docker.com/). You should be able to find a "Create Repository" button on the dashboard (at the time of this writing) after you log in. Fill out the required inputs, which are probably something similar to the following (Figure 9.1):

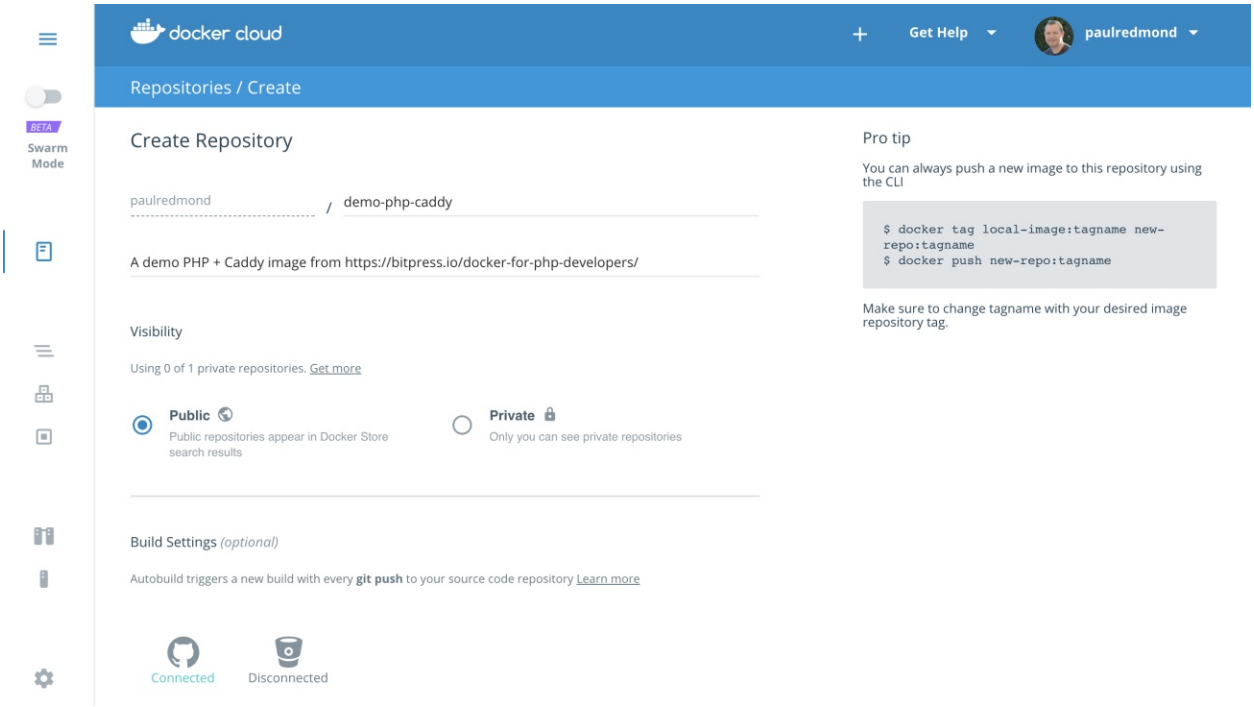


Figure 9.1: Create the PHP Caddy Repository

For now, leave the project as public and ignore the build settings for Github and BitBucket. We come back to these settings later so you can automate Docker builds

with a GitHub webhook.

Once you create the repository, you are redirected to the repository's main page in your Docker Closue account. Take note of the *docker push paulredmond/demo-php-caddy:tagname* instruction (which differs based on your login) for a reminder of how to push images to Docker Hub.

That's it for this section; we have all the foundational pieces in the place to build and push our images to a registry.

Pushing the Image to Docker Hub

In each chapter of the book so far you have been building images on your machine. After you create your images, the next part is pushing your images with the *docker push* command.

Before we automate building images, let's build and tag the image locally and then push it to the Docker repository (Listing 9.4):

>_ Listing 9.4: Build and Tag the Image

```
# Replace `paulredmond` with your own username
$ docker build -t paulredmond/demo-php-caddy:1.0.0 .
$ docker tag paulredmond/demo-php-caddy:1.0.0 paulredmond/demo-
php-caddy:latest
```

After we build the 1.0.0 image, we tag *latest* to that image as well. You should see the tagged image if you run *docker images* after the build completes. Be sure to run the command with your own username: *<your_user>/demo-php-caddy:1.0.0*.

To push the image to Docker Hub, you need to verify your credentials on the command line with the *docker login* command. By default running the login command logs you into the Docker Hub registry. You can also specify the server, for example: *docker login registry.gitlab.com*. You can learn more about the ins-and-outs of docker login

(https://docs.docker.com/engine/reference/commandline/login/) in the documentation.

Next, we will push our image to the Docker Hub registry (Listing 9.5):

Listing 9.5: Login to Docker Hub

```
# This should be your Docker account login
$ docker login -u paulredmond
Password:
Login Succeeded
```

You can also pass the *-p* flag to specify a password or just enter it when prompted.

Now that we've logged in we can push the image that we built in Listing 9.4 to the registry (Listing 9.6):

Listing 9.6: Push the Image to Docker Hub

```
$ docker push paulredmond/demo-php-caddy

The push refers to a repository [docker.io/paulredmond/demo-php-
caddy]
1d74acdd1655: Pushed
9d6fec3ec5e6: Pushed
0fb5e3bf0e56: Pushed
4fd9e9b3007e: Pushed
4d30cdcc06fc: Pushed
8377c955bbe0: Pushed
ad76b6a711fc: Pushed
ba2e080162fa: Pushed
d5231feae7c4: Pushed
b56c638d6e6a: Pushed
958c46160919: Pushed
c4066de46cb2: Pushed
0d960f1d4fba: Pushed
1.0.0: digest:
```

```
sha256:dd312b9f99eb461d369cf1d4ec22b7f3122054ad20f7ee76ef745c0961
43b72d size: 3035
```

The rough image size is about 400mb, which might take a little time depending on your connection speed. When the image finishes, you should see the new version on the Docker Hub registry page.

Now, let's remove the image locally and make sure we can pull it down from the registry (Listing 9.7):

Listing 9.7: Remove the Image

```
$ docker rmi paulredmond/demo-php-caddy:1.0.0
Untagged: paulredmond/demo-php-caddy:1.0.0
Untagged: paulredmond/demo-php-caddy@sha256
...
```

If you run *docker images* you shouldn't see the tagged image anymore. Let's bring it back again with *docker pull* this time, just like our base images when we run a build (Listing 9.8):

Listing 9.8: Pulling Down the Docker Image from Docker Hub

```
$ docker pull paulredmond/demo-php-caddy:1.0.0

1.0.0: Pulling from paulredmond/demo-php-caddy
...
17b671f61b72: Pull complete
f48bdef0222c: Pull complete
e352b2399abc: Pull complete
6c34833cb82a: Pull complete
Digest: sha256:dd312b9f99...
Status: Downloaded newer image for paulredmond/demo-php-
caddy:1.0.0
```

You just built a docker image, pushed it to Docker Hub with your credentials, deleted the local build, and then pulled the image down from the Docker Hub registry. Making images by hand is good practice, but it's time to automate.

Automating the Image Build

You might have noticed the BitBucket and GitHub integration when you created your repository. Let's set up some automation to create a new version of the image when we push changes to Github.

Before you can follow along, you need to create an example repository on GitHub (or Bitbucket) and push your project files to your repository. My source files are located on GitHub at paulredmond/demo-php-caddy (https://github.com/paulredmond/demo-php-caddy) if you want a reference.

Authorizing GitHub

The first step to automating our Docker repository build is approving GitHub. From the settings on cloud.docker.com, you can select source providers and connect GitHub to allow Docker Cloud to access your repositories (Figure 9.2):

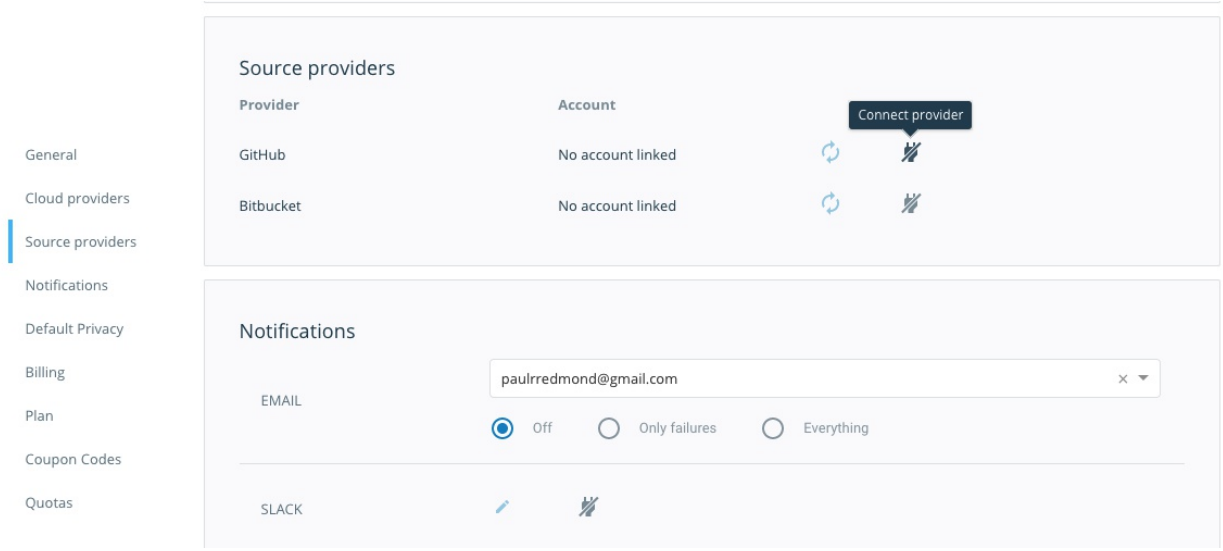


Figure 9.2: Authorize Docker Cloud Access to GitHub

Once you allow GitHub or BitBucket, you can then go to your repository's build settings and select the code repository you want to link to Docker Cloud. For now, choose

Docker Cloud's infrastructure as the build location (the small node size), and you can change it later if you want to experiment with building on nodes that you control (Figure 9.3):

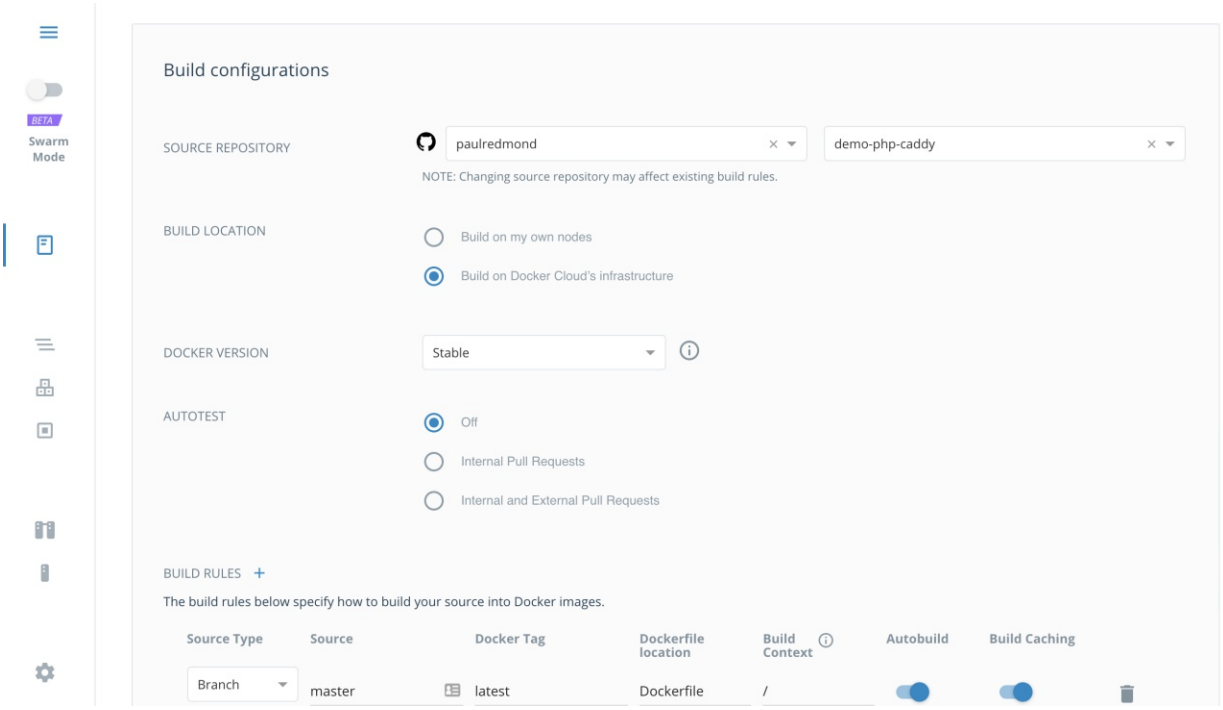


Figure 9.3: Setting up builds

The last part of the setup is configuring build rules. We will have two: master for the *latest* tag and a git tag/release for versioning (Figure 9.4):

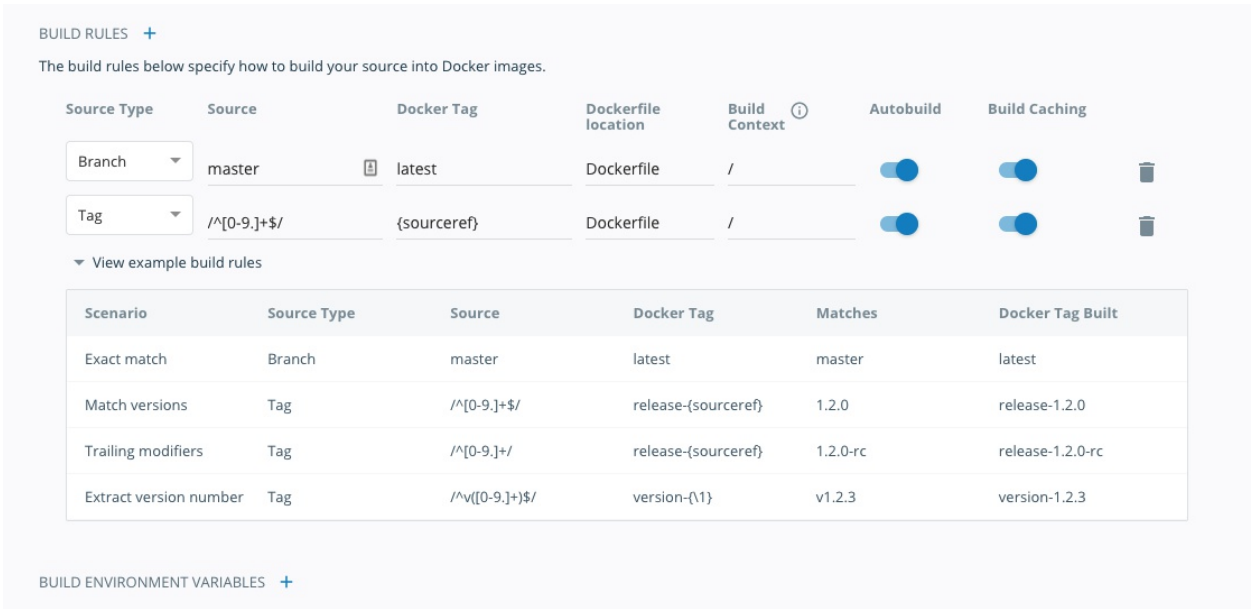


Figure 9.4: Configuring build rules

Once you add the tag build rule, you can click "Save and Build" to start making master; make sure that you push your code to GitHub before starting the build. If all goes well, you should see a successful build!

Free Docker Cloud Build Nodes

At the time of this writing, Docker Cloud build nodes are in beta and free. If Docker Cloud changes the pricing structure in the future, you might have to configure infrastructure nodes that you own to run builds.

Using tags and releases allows you, the developer, to focus the code while Docker Cloud takes care of building and tagging images that match your code versioning scheme automatically.

Releasing a New Version

Let's make a small change to the Dockerfile and tag a release version to test our tag build configuration. We add the *opcache* module to the Dockerfile to trigger a change (Listing 9.9):

Listing 9.9: Adding the Opcache Module

```
FROM php:7.1-fpm

LABEL maintainer="Paul Redmond"

RUN curl --silent --show-error --fail --location \
      --header "Accept: application/tar+gzip, application/x-gzip,
application/octet-stream" -o - \

"https://caddyserver.com/download/linux/amd64?plugins=http.expire
s,http.realip&license=personal" \
    | tar --no-same-owner -C /usr/bin/ -xz caddy \
    && chmod 0755 /usr/bin/caddy \
```

```
    && /usr/bin/caddy -version \
    && docker-php-ext-install mbstring pdo pdo_mysql opcache

COPY Caddyfile /etc/Caddyfile
WORKDIR /srv/app/
RUN chown -R www-data:www-data /srv/app

CMD ["/usr/bin/caddy", "--conf", "/etc/Caddyfile", "--log",
"stdout"]
```

Commit your changes and push them to your repository in the master branch. You should see a build triggered in Docker Cloud for the *master* branch (Figure 9.5):

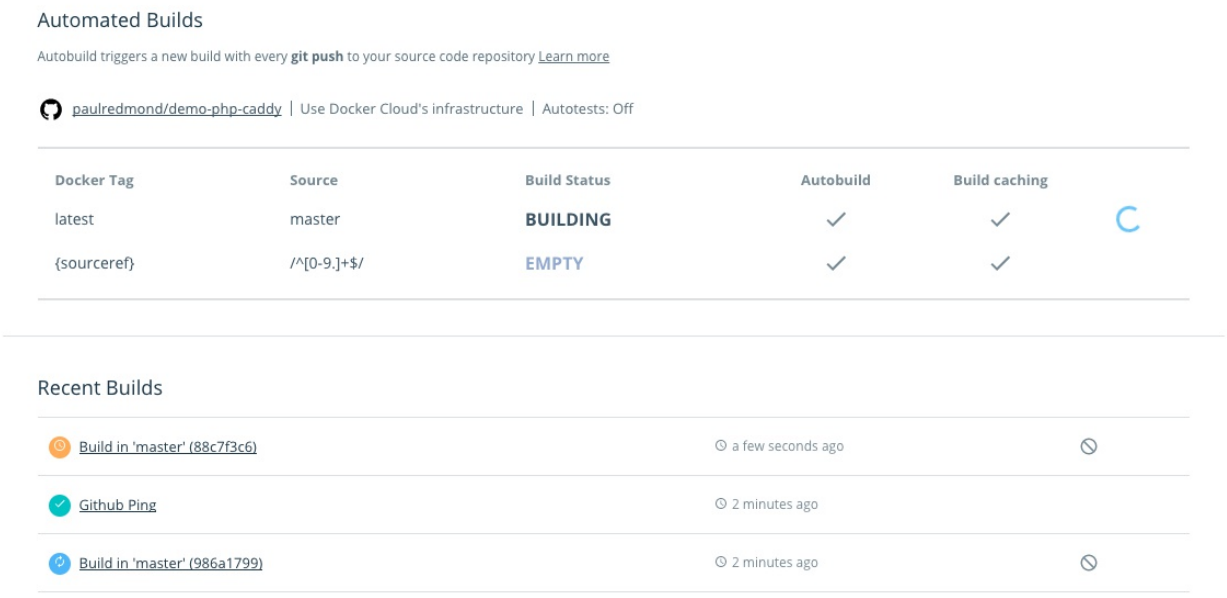


Figure 9.5: Automatic Master Build in Progress

After you push your code, create a tag or release of *1.1.0*. For example, I used GitHub to create the release (https://github.com/paulredmond/demo-php-caddy/releases/tag/1.1.0). You can also tag a release from the command line and push it to your repository.

After the build finishes, you should see a new version of 1.1.0 in your dashboard (Figure 9.6):

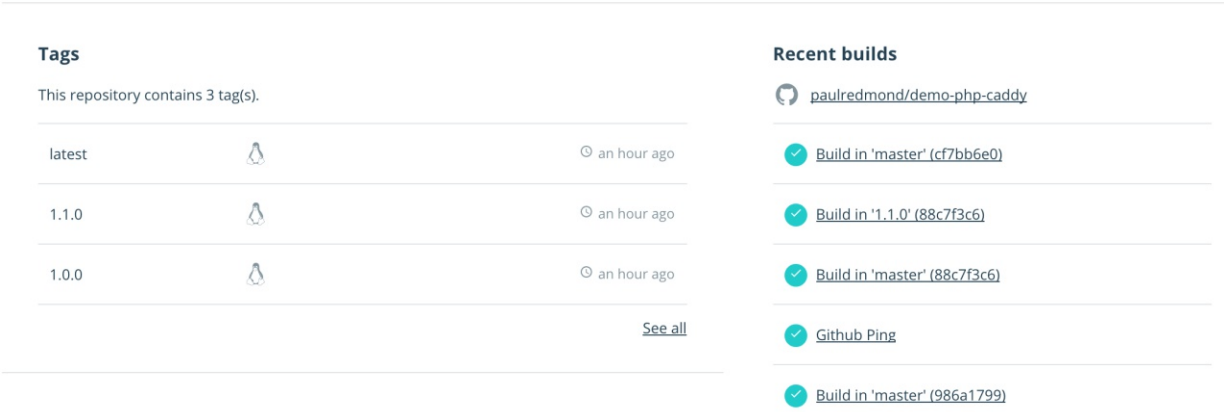


Figure 9.6: Dashboard Build List

We've successfully automated our builds with Docker Hub, wasn't that easy? Docker Hub isn't the only container registry, so let's learn how to work with other registries.

Working with Other Docker Registries

There are many Docker repositories our there, such as Quay, Google Container Registry, Amazon EC2 Container Registry, and GitLab, to name a few. Since GitLab.com has a free plan available at the time of this writing, we use the GitLab registry to work through this section.

You need to register for an account (https://gitlab.com/users/sign_in) if you want to follow along. How you interact with a registry on the command line is the same regardless of which registry you use, so your knowledge applies to other registries.

Setting up a GitLab Repository

We can use the same git repository we already have to work with GitLab. It's just a matter of setting up another remote and creating a private repository in your account. Once you have the repository set up in GitLab, add another remote (Listing 9.10):

Listing 9.10: Adding a new remote to git

```
$ git remote add gitlab \
    git@gitlab.com:paulredmond/demo-php-caddy.git
$ git push gitlab master && git push gitlab --tags
```

If you sign in to GitLab, you should see a "Registry" tab in the project. This tab has some good instructions on how to login to the registry. Let's log in to GitLab's registry and push the latest image in master (Listing 9.11):

Listing 9.11: Pushing the Image to GitLab

```
$ docker login registry.gitlab.com
Username: paulredmond
Password:
Login Succeeded

# Now build the image and tag it for GitLab's Registry
$ docker build -t \
    registry.gitlab.com/paulredmond/demo-php-caddy .
```

We've already built our image locally, so we could also just tag the image with *docker tag*. However, in most situations you only build in one registry, so most of the time you are building and then tagging at the same time. However, it's also useful to tag an image for a registry from an existing image (Listing 9.12):

Listing 9.12: Tagging an Existing Image

```
# We built the 1.0.0 image earlier in the chapter
$ docker tag \
    paulredmond/demo-php-caddy:1.0.0 \
    registry.gitlab.com/paulredmond/demo-php-caddy:1.0.0

# You can also pull down the 1.1.0 image and tag it for GitLab
# Remember, it was built in Docker Cloud
# So you don't have a local build of 1.1.0 yet
$ docker pull paulredmond/demo-php-caddy:1.1.0

$ docker tag \
  paulredmond/demo-php-caddy:1.1.0 \
  registry.gitlab.com/paulredmond/demo-php-caddy:1.1.0
```

```
# Push them to Gitlab
$ docker push registry.gitlab.com/paulredmond/demo-php-caddy
```

In Listing 9.12, we tagged our image with registry.gitlab.com. You must include the hostname, or it is assumed a Docker Hub image. We also pulled down version 1.1.0 from Docker Hub, tagged it for GitLab, and then pushed it up to GitLab's registry.

Automating Builds on GitLab

We can automate our image builds and push to GitLab's registry automatically, just like we did with Docker Cloud. With GitLab we configure our builds with a *.gitlab-ci.yml* file in the root of the project with three separate build scenarios: master, tag, and branch. What's neat about the branch build, is that you can pull down a work-in-progress image and collaborate on it with your team. The other two build scenarios are just like the Docker Cloud builds we've already seen.

First, we need to create the *.gitlab-ci.yml* file for the project based on the GitLab Docker Template (https://gitlab.com/gitlab-org/gitlab-ci-yml/blob/master/Docker.gitlab-ci.yml). This build uses a Docker image (https://hub.docker.com/_/docker/) (yes, Docker running inside of Docker) to build our image and then push it to the GitLab registry automatically (Listing 9.13):

</> Listing 9.13: The .gitlab-ci.yml file

```
# Official Docker image.
image: docker:latest

services:
  - docker:dind

before_script:
  - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
$CI_REGISTRY
```

```
# Master
build-master:
  stage: build
  script:
    - docker build --pull -t "$CI_REGISTRY_IMAGE" .
    - docker push "$CI_REGISTRY_IMAGE"
  only:
    - master

# Tags
build-tag:
  stage: build
  only:
    - tags
  script:
    - docker build --pull -t "$CI_REGISTRY_IMAGE:$CI_COMMIT_TAG" .
    - docker push "$CI_REGISTRY_IMAGE:$CI_COMMIT_TAG"

# Branch Builds
build-branch:
  stage: build
  script:
    - docker build --pull -t
"$CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG" .
    - docker push "$CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG"
  except:
    - tags
    - master
```

Let's break down this configuration file; it's quite straightforward actually. The file contains three build stages: build-master, build-tag, and build-branch. All the variables that start with a dollar sign ($) are GitLab build variables which we can use to reference our registry image URI and the tag (https://docs.gitlab.com/ee/ci/variables/).

Each job builds a Docker image and then pushes it to the GitLab registry in different scenarios. First, the *build-master* build tags the Docker image with *latest* when the code

is pushed to master. Second, the *build-tag* tags the image with the commit tag (i.e. ch09-sample-project:1.0.2). Moreover, last, the *build-branch* build tags with the image branch name.

As an example of how the branch configuration works, let's say that you have the branch *feature/test*. The *$CI_COMMIT_REF_SLUG* variable would end up being *feature-test*, and the tagged image would be *paulredmond/demo-php-caddy:feature-test*. Branch builds are kind of neat in my opinion but might be overkill for your situation. Branch builds can add up when GitLab automatically makes an image when you push code to Gitlab for each branch.

Take note of the *except:* key in the branch build, which excludes tags and the master branch. The *only* and *except* keys help you define where each build runs.

Build Caching

At the time of this writing, I am not sure how to get cached builds working with GitLab. If you recall when we set up our Docker Cloud pipeline, we were able to select a cached image. This speeds up builds significantly, including downloading the base php image.

For base images, this probably won't be a problem on GitLab, but for a more extensive project, builds might take longer. There's a discussion on this topic on GitLab.com (https://gitlab.com/gitlab-org/gitlab-ce/issues/17861).

Now, when you push your image changes to GitLab, an automatic build is triggered, and you can see the build jobs in the "pipelines" section of your project.

Extending Your Images

Now that we have an image in a private GitLab registry let's extend it to see how you can build on top of a base image. It's the same as we have already been doing with the

Now, when you push your image changes to GitLab, an automatic build is triggered, and you can see the build jobs in the "pipelines" section of your project.

Extending Your Images

Now that we have an image in a private GitLab registry let's extend it to see how you can build on top of a base image. It's the same as we have already been doing with the PHP image, but let's go over it real quick before the end of this chapter.

You can create your project in another folder, all we need is a Dockerfile and an index.php file with "<?php phpinfo(); ?>" (Listing 9.14):

</> Listing 9.14: Extending our own image

```
FROM registry.gitlab.com/paulredmond/demo-php-caddy

COPY index.php /srv/app/public/index.php
```

We've extended the GitLab image, and then just copy an *index.php* file into the image. The FROM pulls our base image and then runs the COPY instruction on top of it. If you build this image and run it, you should see the PHP info screen (Listing 9.15):

>_ Listing 9.15: Run the Extended Image

```
$ docker build -t ch09-sample-project .
$ docker run --rm -it -p 8080:2015 ch09-sample-project
# Open http://localhost:8080
```

If all went well, you should see the output from *phpinfo()* in your browser! Although we just added an index.php file, this could be an entire application that has a Dockerfile, except instead of extending the PHP Docker image directly we extend our base image that we can use on multiple projects.

Registered Docker Image Builder

That's it for registries; you made it! You've now played around with pushing your own Docker images to the world. You may likely build public images that you want to open-source, and you have many options for storing your private images too. Either way, learning how to log in and work with repositories enables you to work with any Docker registry. Learning how to automate Docker builds not only takes manual work off your plate but makes it more consistent and automatic. Plus, it merely feels neat to automate Docker image builds.

We also explored GitLab, which is a great place to experiment with private Docker registries and CI build pipelines for Docker images you don't want to share with others. Using GitLab allows you to automate the entire build pipeline of your applications pretty easily!

Chapter 10: Deploying Docker

You are ready to take everything you've learned so far and combine it all into the grand finale: we're going to deploy a Docker application to the cloud.

This book's focus is on developing with Docker—where you probably spend most of your time—but eventually, you'll need to deploy your applications. Even if you are not the primary person deploying Docker to production in your organization, you should learn about the strategies around deploying Docker.

Docker deployment can be as simple as pulling an image from a registry (just like we did in Chapter 9 on a server in the cloud. More complicated setups might need to operate at scale, run clusters of servers, and provide container management tools. The most notable Docker tools (typically for larger deployments) include Rancher, Kubernetes, and Docker Swarm. Both Amazon and Google have container services (Google's is powered by Kubernetes).

These Docker deployment tools have similarities, yet each one has unique capabilities and quirks that we can't possibly cover in this text. Don't worry; I have provided plenty of next steps at the end of this book. The concepts that we cover in this chapter should establish a good foundation to explore other tools.

We are going to walk through an actual deployment of Rancher (http://rancher.com/) and running an application within our Rancher installation. We cover Rancher in this book because the setup can be relatively minimal, yet extensive when you need it, and you can run Rancher in any cloud provider you want. We use Digital Ocean in this text,

but you could even adapt this to work with a VirtualBox VM running Linux if you don't want to use a cloud provider for this chapter.

Rancher Overview

Rancher describes itself as:

> *Simple, easy-to-use container management.*

Rancher provides a nice UI that you can use to deploy containers in the Cloud and scale them; Rancher Server also takes care of issues like networking and load balancing. You can also automate your infrastructure deployments with the Rancher compose (http://rancher.com/docs/rancher/v1.6/en/cattle/rancher-compose/) CLI, giving you lots of automation options to truly optimize your deployments.

Rancher has two primary components: the Rancher server (http://rancher.com/docs/rancher/v1.2/en/installing-rancher/installing-server/) and a Rancher host. As outlined in the Rancher Overview, the server component "provides infrastructure orchestration, container orchestration, an application catalog, and authentication control."

The server works with Rancher hosts (http://rancher.com/docs/rancher/v1.6/en/hosts/) that run an agent and communicate with the Rancher server over HTTP. You can create a new host from within the Rancher server UI in your cloud provider of choice, or by running the Rancher agent image with *docker run* on a server. Rancher doesn't care if your servers running the agent are virtual or physical as long as they meet the requirements outlined in the documentation.

With that brief overview of Rancher out of the way, it's time to get started. We have a server that we need to build!

Setting up the Server

Digital Ocean has a Docker Droplet (https://www.digitalocean.com/products/one-click-apps/docker/) that makes getting Docker in the cloud one click. If you want to install Docker manually on the cloud or server of your choice, you can reference Chapter 1's Linux instructions.

Pick a Droplet with enough RAM (1 GB of RAM is the requirement) to run Rancher Server. The $10 per month droplet is probably barely large enough, so the $20 droplet is more ideal. You can shut it down afterward if you want, so it shouldn't be costly ($0.030/hour at the time of this writing).

I am assuming that you are comfortable using SSH to access a Digital Ocean server, but if you need a little guidance you can add an *~/.ssh/config* file on Linux or Mac with something like the following (change the IP address to the public IP of your droplet):

```
host rancher-server
  hostname 159.203.121.76
  user root
  IdentityFile ~/.ssh/id_rsa
  ForwardAgent yes
```

With the above SSH configuration, you can use the host alias with *ssh rancher-server* from the command line.

Once you have your droplet created, SSH into the box and run the following command (Listing 10.1):

Listing 10.1: Start the Rancher Server

```
# Set firewall rules
$ ufw allow 8080
$ ufw allow 500/udp
$ ufw allow 4500/udp
```

```
$ sudo docker run -d --restart=unless-stopped -p 8080:8080 \
    rancher/server
```

We ran the familiar *docker run* command, which executes the official *rancher/server* (https://hub.docker.com/r/rancher/server/) image with a restart policy that restarts the container unless you manually stop it. Even if you reboot your droplet, the container will restart afterward. Last, we configure two ports Rancher uses for networking between hosts with Ubuntu's UncomplicatedFirewall (UFW).

After the container starts running, be patient as it can take a bit of time to finish initializing the first time. Once the Rancher server image is done downloading and initializing, the first thing you'll do is lock down the Rancher control panel. Find the public IP address of your server and visit it on port 8080. For example, mine while writing this chapter is http://45.55.184.106:8080.

Rancher Server Hostname

You can set up a DNS A record for your server instead of going to the IP address (i.e., rancher.yoursite.com).

If you aren't sure what is happening with the Rancher server setup, you can check out the logs from the command line:

```
# Find the rancher server container
$ docker ps
$ docker logs -f b9c6a69e7eb3
```

Hit *Ctrl + c* to stop following the container logs when you are done to get back to a prompt.

Add Authentication

To lock down the Rancher server, visit "Admin > Access Control" and pick an

authentication scheme. Selecting local authentication works perfectly for this tutorial (Figure 10.1):

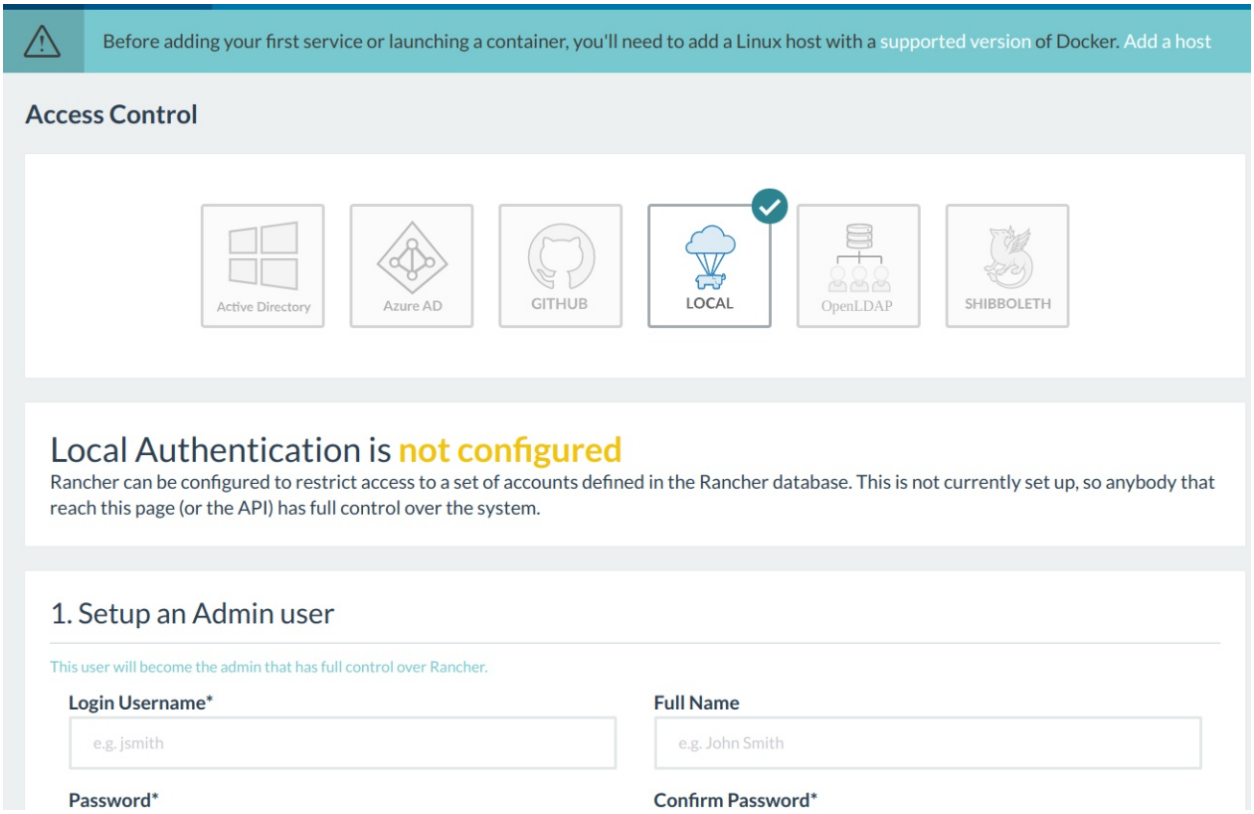


Figure 10.1: Local Authentication

You can select any access control that you want as long as you pick something! Once you select your authentication method, set up an administrator account to secure your control panel.

Setting up Infrastructure

Now that you have a protected server, you need to add some Rancher agents to run Docker containers. In this chapter, we use Digital Ocean to create a new Droplet (virtual machine) to house our demo application containers. For this chapter, we have one server running Rancher and another running the Rancher agent. However, in practice, you might have more than one Rancher agent running, and the setup is the same.

To manage infrastructure in Rancher server, navigate to "Infrastructure > Hosts" and

click the "Add Host" button. You are then prompted to add a host registration URL, with a pre-populated option with your IP address (or hostname if you set up DNS) and all you do is click "Save." Next, pick "Digital Ocean" from the list of machine drivers, and next it prompts you to enter a personal access token from Digital Ocean.

Within your Digital Ocean control pane, visit the API settings (https://cloud.digitalocean.com/settings/api/tokens) and create a new personal access token (both read and write scopes) and then paste it into your admin access token field back in the Rancher UI (Figure 10.2):

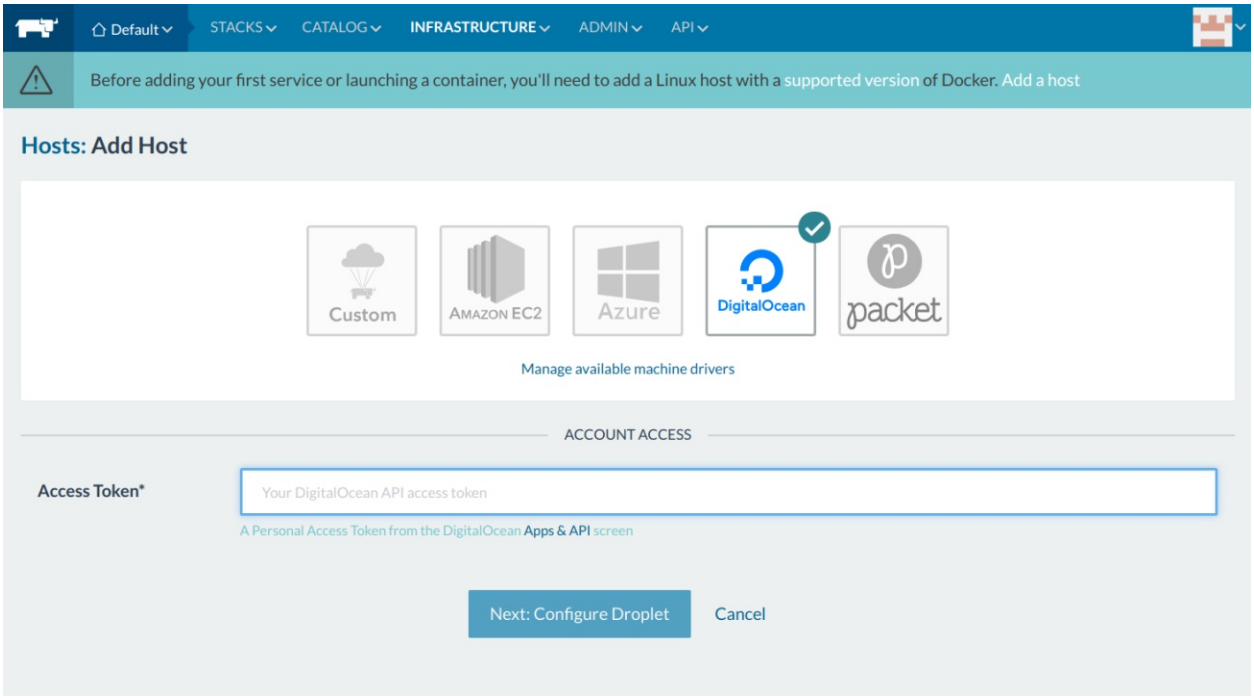


Figure 10.2: Add a Digital Ocean host

Next, you are prompted to configure your droplet. Just name the droplet hostname something like "rancher-agent-01", or whatever you want to identify it as a Rancher host. You can pick different sizes but just go with the defaults for now. Once you confirm the droplet, Rancher polls the host while the Droplet is being created and shows you a bunch of information once the host is active (Figure 10.3):

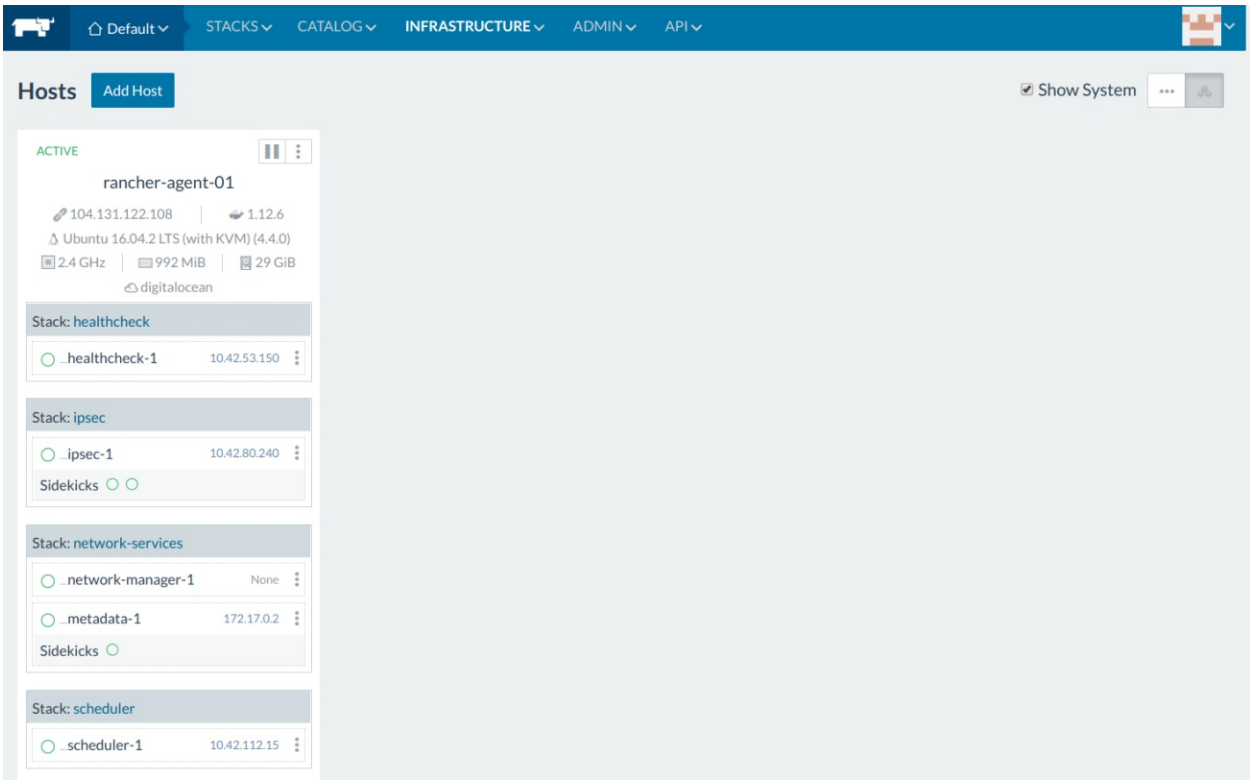


Figure 10.3: Rancher active Digital Ocean host

If you create expensive droplets, you can destroy them at the conclusion of this chapter, so they don't continue to cost you hourly. Don't forget about them!

Tagging Hosts in Rancher

You might have noticed the tag settings when creating the Digital Ocean host in Rancher. You can label hosts to track and organize them however you want. You don't need tags if you're only running a couple of servers.

Introduction to Stacks

So far we have a Rancher server and a Rancher host in Digital Ocean, but we don't have any software running. Rancher provides a bunch of automated software called "stacks" that you can install with the click of a button.

Navigate to "Stacks > All," and then click "Add from Catalog" to see popular stacks. You can browse and install various services from the catalog found in infrastructure stacks,

such as Cloudflare external DNS, Digital Ocean external DNS service, and so on.

Next, if you navigate to "Stacks > User" and click "Browse Catalog," you can install user stacks from a catalog such as a Consul cluster, WordPress, or even a GitLab CE instance.

The catalogs in Rancher are common services and software that you can install by clicking a button from the Rancher interface, which is pretty convenient!

For our purposes, we are going to create our user stack from scratch, but nothing is stopping you from installing premade stacks via the interface if you want to experiment. We could create our custom stack from the user interface, but we are going to get it running through Rancher Compose to demonstrate creating repeatable, custom stacks.

Using Rancher Compose

Rancher Compose (http://rancher.com/docs/rancher/v1.6/en/cattle/rancher-compose/) is very similar to Docker Compose, except Rancher Compose operates across multiple hosts. In our example, we only have one rancher agent host, but Rancher starts containers across hosts based on scheduling rules when you have multiple. You can read more about Rancher compose in the documentation, but suffice it to say that we can automate the creating of stacks through a repeatable rancher-compose YAML file in our project.

The Docker images that run in your containers are from a GitLab registry account if you are following along. We've covered how to create Docker images in the GitLab registry already, but feel free to use a different registry if you want to.

To deploy the stack, you need to have a few sets of credentials:

1. Enter your GitLab registry credentials
2. Create Environment API Key credentials for Rancher Compose

First, we need to provide GitLab credentials so that Rancher can pull our registry images from Gitlab. Navigate to "Infrastructure > Registries," then select "Custom" and enter your login details for the Gitlab registry (Figure 10.4):

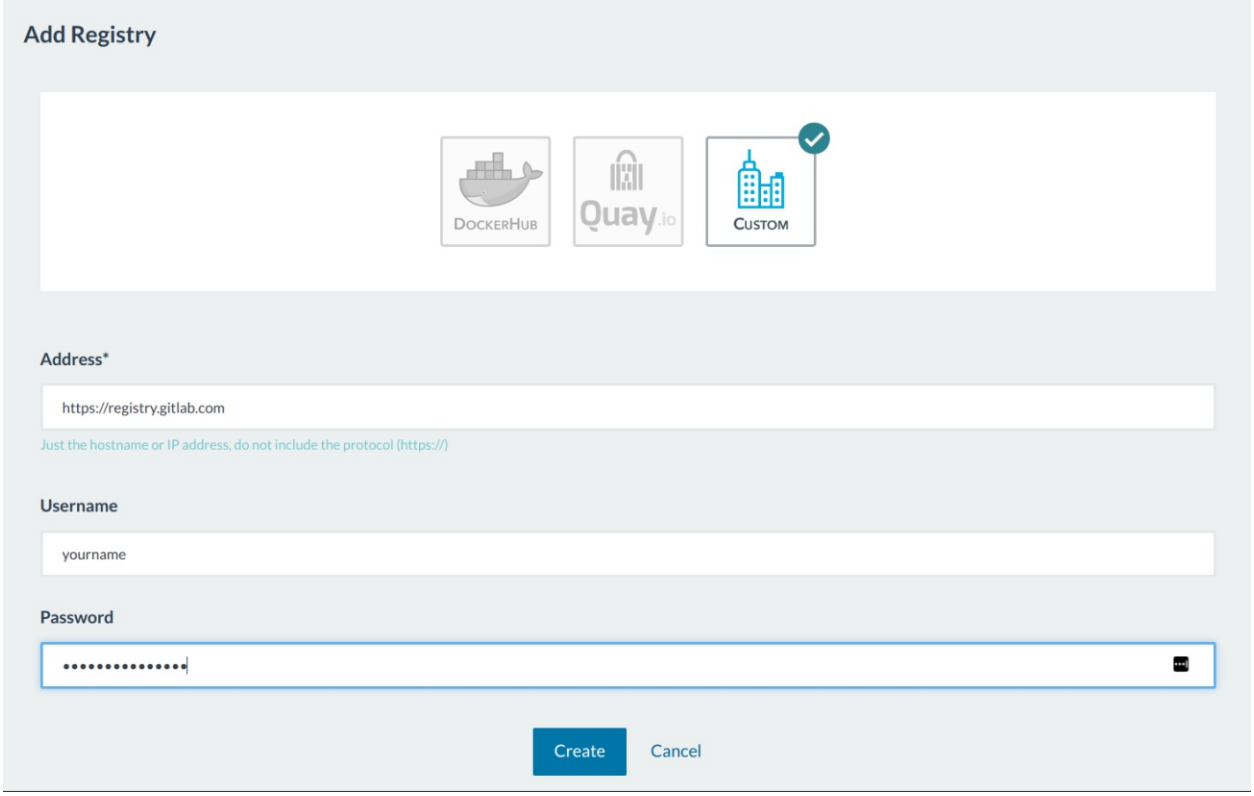


Figure 10.4: Adding the GitLab registry

Now that you've saved your GitLab registry credentials, we need to set up a demo project to run in production. Later, we come back to creating environment-specific API credentials for Rancher Compose. For now, it's time to shift focus from Rancher to creating a demo project to deploy code into Rancher.

Registry Accounts

If you are working in a team environment, it might be advisable to use a separate account for your automation, including registry login. The separate account helps separate the automation from regular user accounts.

Setting Up the Project

Rancher is ready and waiting for a project that we're going to deploy in this section. We'll use a demo Laravel (https://laravel.com/) project, and use Docker compose in combination with a Rancher Compose file to deploy our application. We'll have a typical Dockerfile along with two files that Rancher uses in tandem: *docker-compose.prod.yml* and *rancher-compose.prod.yml*.

Let's kick this off by creating the Laravel and project and the files we need to deploy to Rancher (Listing 10.2):

Listing 10.2: Creating the Rancher Demo Project

```
$ cd ~/Code
$ composer create-project laravel/laravel:5.5.* \
    rancher-laravel-demo

$ cd rancher-laravel-demo/
$ mkdir docker/
$ touch docker/Dockerfile \
    docker/Caddyfile \
    docker-compose.yml \
    docker-compose.prod.yml \
    rancher-compose.prod.yml
```

We are going to use Caddy with PHP-FPM for this application, and the same Dockerfile that we used in Chapter 9. Let's knock out the Dockerfile before we look at the Rancher Compose file (Listing 10.3):

Listing 10.3: The Dockerfile

```
FROM php:7.1-fpm

LABEL maintainer="Paul Redmond"
```

```
RUN curl --silent --show-error --fail --location \
      --header "Accept: application/tar+gzip, application/x-gzip,
application/octet-stream" -o - \

"https://caddyserver.com/download/linux/amd64?plugins=http.expire
s,http.realip&license=personal" \
    | tar --no-same-owner -C /usr/bin/ -xz caddy \
    && chmod 0755 /usr/bin/caddy \
    && /usr/bin/caddy -version \
    && docker-php-ext-install mbstring pdo pdo_mysql

COPY . /srv/app
COPY docker/Caddyfile /etc/Caddyfile

WORKDIR /srv/app/
RUN chown -R www-data:www-data /srv/app

CMD ["/usr/bin/caddy", "--conf", "/etc/Caddyfile", "--log",
"stdout"]
```

Next, we need to create the Caddy server configuration file at *docker/Caddyfile* (Listing 10.4):

</> Listing 10.4: The Caddyfile

```
0.0.0.0
root /srv/app/public
gzip
fastcgi / 127.0.0.1:9000 php
rewrite {
    regexp .*
    ext /
    to /index.php?{query}
}

header / -Server
```

```
log stdout
errors stdout
on startup php-fpm --nodaemonize
```

Before we work with the Rancher Compose files, let's edit the *docker-compose.yml* and make sure that the Docker setup is working as expected (Listing 10.5):

Listing 10.5: The Development docker-compose.yml file

```
version: "3"
services:
  app:
    build:
      context: .
      dockerfile: docker/Dockerfile
    ports:
      - 8080:2015
```

We set the build context to the root of the project so Docker can copy the files correctly, yet the Dockerfile can be tucked away in the *docker/* folder. We've seen a few examples of using context with a Dockerfile already. Lastly, we map Caddy's default port of 2015 to port 8080 on the host machine.

You can run a local version of the application with *docker-compose up --build* and then visit http://localhost:8080 to verify. If all goes well, you should see Laravel's default application page (Figure 10.5):

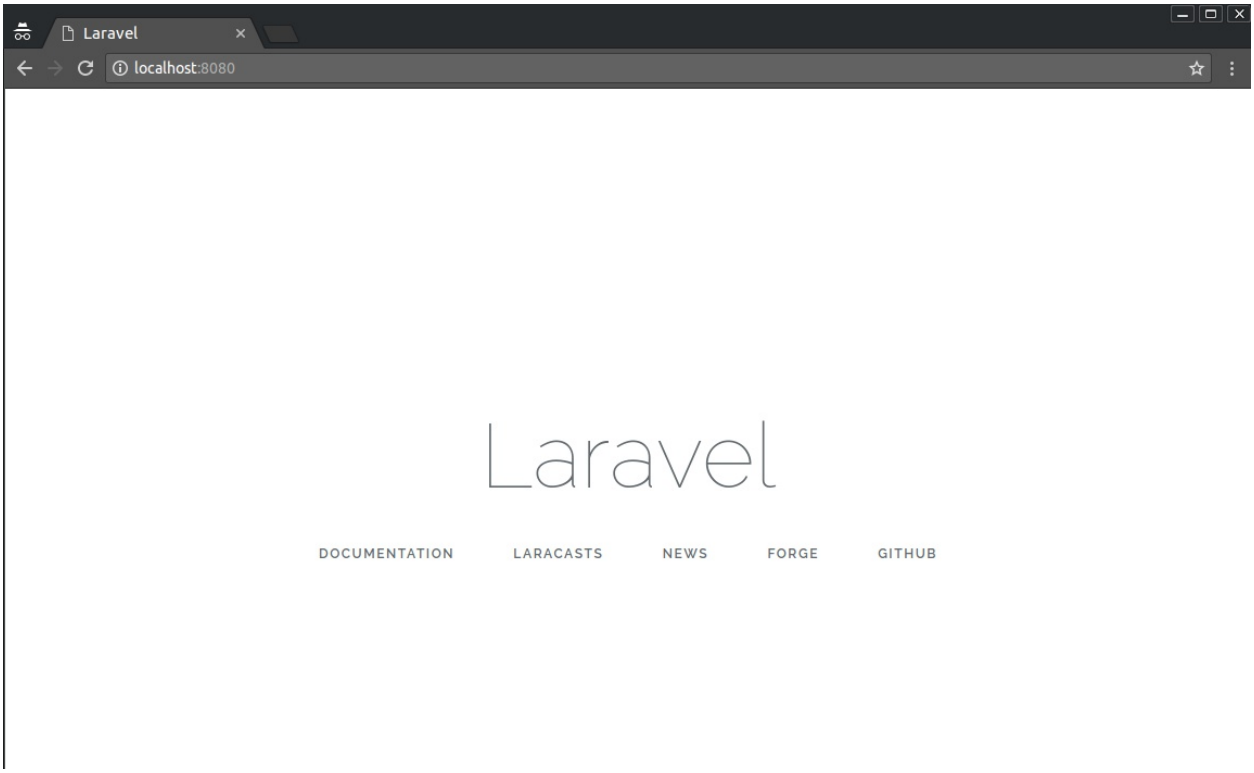


Figure 10.5: Laravel welcome page

In this section, we created a separate Docker Compose file for development and production-like builds that we can use independently with the *docker-compose --file* flag (we briefly demoed this flag in Chapter 4). We use the default *docker-compose.yml* file for development environments and the other environment-specific files for deployment automation. As you work with Docker more, you start to build workflows that might differ slightly (and that is perfectly fine), but this is a convention that has worked well for me.

Project Version Control and Registry Builds

If you recall when we set up Rancher, we logged into the GitLab registry. To pull in our Docker images that we're pushing to GitLab, we need to create a new project in GitLab and select images to GitLab's Docker registry.

Let's initialize a git repository and push our files to GitLab (Listing 10.6):

Listing 10.6: Pushing the Code to a GitLab project

```
$ cd ~/Code/rancher-laravel-demo
$ git init
$ git add .
$ git commit -m"First commit"
$ git remote add origin \
    git@gitlab.com:paulredmond/rancher-laravel-demo.git
$ git push origin master
```

With our code under version control on GitLab, let's create a *build.sh* file in the root of the project in order to automate the build (Listing 10.7):

Listing 10.7: Build and tag the Docker image

```
$ touch build.sh
$ chmod u+x build.sh
```

Next, add the following build script to *build.sh* (Listing 10.8):

Listing 10.8: The Build Script

```
#!/usr/bin/env bash

tag=${1:-latest}

echo "Enter your Gitlab Credentials..."
docker login registry.gitlab.com

docker build -f docker/Dockerfile -t \
  registry.gitlab.com/paulredmond/rancher-laravel-demo:$tag .

docker push \
  registry.gitlab.com/paulredmond/rancher-laravel-demo:$tag

echo "Build $tag complete"
```

First, we define a tag variable with the default of *latest*. Next, we log in to the GitLab registry so the script can interact with private registries. Lastly, the script builds the image with a tag and pushes it to the GitLab registry. Be sure to substitute your registry URL based on the project you created in GitLab!

We're ready to run the build script, which if you execute the build command without any arguments, it tags the image as *latest* by default (Listing 10.9):

Listing 10.9: Build the Image

```
# Tag the latest
$ ./build.sh

# Example of tagging a version
$ ./build.sh 1.0.0
```

Go ahead and build the *latest* tag by running *./build.sh* with no arguments so you can verify that your build is working.

Setting up the Stack Deployment

Our registry build is working, and we are ready to get builds running with Rancher Compose. If you recall, Rancher Compose is a way that we can run Docker in a similar way to Docker Compose. Rancher Compose works with Docker Compose to deploy services to Rancher and allows you to scale containers (multiple instances of the same containers). In the end, we run the *rancher-compose* CLI to deploy our containers.

To run the *rancher-compose* command locally, you will need to download it and put it in your path. You can download the latest version of *rancher-compose* for your operating system by logging into your Rancher server and clicking the link at the very bottom right. Select the appropriate operating system and copy the file to */usr/local/bin/rancher-compose* and make it executable (Listing 10.10):

Listing 10.10: Set up the rancher-compose binary

```
$ sudo cp path/to/rancher-compose /usr/local/bin/rancher-compose
$ sudo chmod u+x /usr/local/bin/rancher-compose

$ rancher-compose --help
```

Now that you have the rancher-compose correctly installed let's work on the production Docker Compose and Rancher Compose files in tandem. As I've already mentioned, rancher-compose collaborates with a Docker Compose file. We haven't edited either file, so let's do so now!

First, let's define the *docker-compose.prod.yml* file (Listing 10.11):

Listing 10.11: The docker-compose.prod.yml file

```
version: "2"
services:
  rancher-laravel-demo:
    image: registry.gitlab.com/paulredmond/rancher-laravel-
demo:latest
    labels: {io.rancher.container.pull_image: always}
    stdin_open: true
    tty: true
```

You might have noticed that we are using Docker compose version two here instead of version three. Rancher is compatible with V1, and V2 of the Docker Compose manifest, so we use V2.

Our Docker Compose file defines one service which points to our GitLab image. The *stdin_open: true* and *tty: true* are the same as *docker run -it* (interactive and tty).

Next, let's fill out our *rancher-compose.prod.yml* file (Listing 10.12):

Listing 10.12: The rancher-compose.prod.yml file

```
version: "2"
services:
  rancher-laravel-demo:
    scale: 2
```

The key matches the service defined in the Docker Compose production file. We define a *scale: 2*, meaning two containers running the *rancher-laravel-demo* service by default.

Writing the Deployment Script

We are ready to deploy our laravel service scaled to two instances. I prefer to create a deploy script (deploy.sh) that we can use to apply deployments quickly (Listing 10.13):

Listing 10.13: Create the deploy script

```
$ touch deploy.sh
$ chmod u+x deploy.sh
```

Here's the contents of the deploy script (Listing 10.14):

Listing 10.14: The Rancher compose deploy script

```
#!/usr/bin/env bash

ACCESS_KEY=$1
SECRET_KEY=$2
RANCHER_URL=$3
ENVIRONMENT=$4

rancher-compose \
    -f docker-compose.$ENVIRONMENT.yml \
    -r rancher-compose.$ENVIRONMENT.yml \
    --url=$RANCHER_URL \
```

```
    --access-key=$ACCESS_KEY \
    --secret-key=$SECRET_KEY \
    --project-name=rancher-laravel-demo \
    up --upgrade --pull --confirm-upgrade -d
```

The deploy script accepts four arguments:

- A Rancher access key
- A Rancher secret key
- A Rancher Endpoint URL
- An environment

We can get the first three from the Rancher server UI, and *$ENVIRONMENT* is used to match our YAML files. We defined the Docker and Rancher files that end in *.prod.yml*, and this allows us to add configuration for other environments (i.e., staging) independently.

The *rancher-compose up* command in the deploy script also has quite a few flags. Some of them are pretty obvious, but I want to point out a few. You can get similar info by running *rancher-compose --help*:

-f specify an alternate Docker compose file (default is *docker-compose.yml*)
-r specify an alternate Rancher compose file (default is *rancher-compose.yml*)
--url the Rancher API endpoint URL
--project-name customize the project name seen in Rancher

After the *up* command, you can learn more about the flags used by running *rancher-compose up --help*.

Setting Up Rancher API Credentials

To get the deploy script working, you need to go back to the Rancher server UI and create an environment API key. The environment key provides an access key

(Username) and an access secret (Password).

Navigate to "API > Keys" and expand the "Advanced Options" section, then click "Add Environment API Key" and enter a name (i.e., rancher-compose) and an optional description. Once you submit, you get a one-time modal with your access key and secret (Figure 10.6):

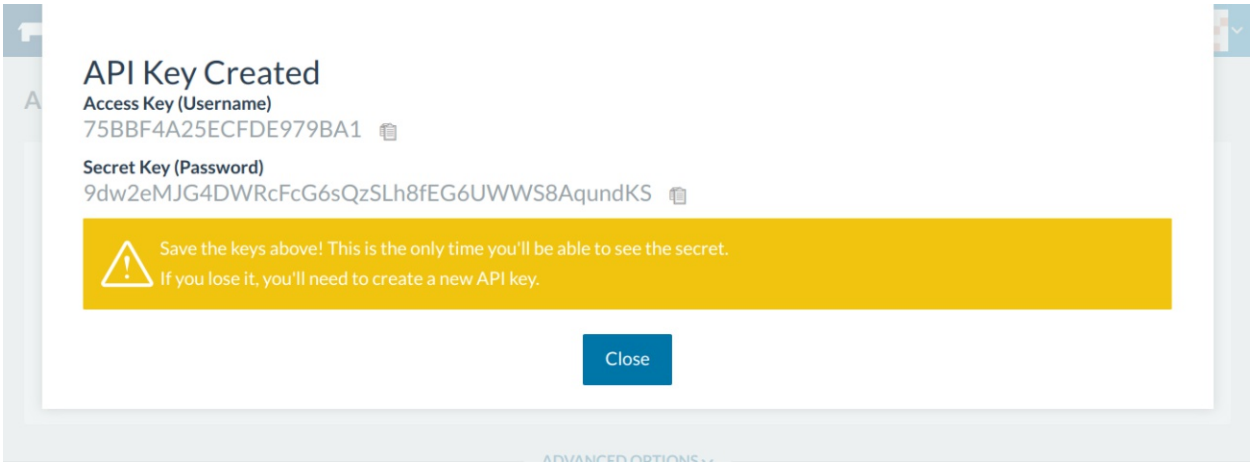


Figure 10.6: Create a Rancher environment API key

Don't lose track of these settings and don't add them to version control! You are required to regenerate these API credentials if you forget them. These credentials typically live in your CI environment, and you need to keep them backed up in a safe place if you plan on keeping your Rancher installation around after this chapter.

After creating the environment API key, take note of the Endpoint (v1) URL, we use that value in the deploy script for the *--url* flag of the Rancher API (Figure 10.7):

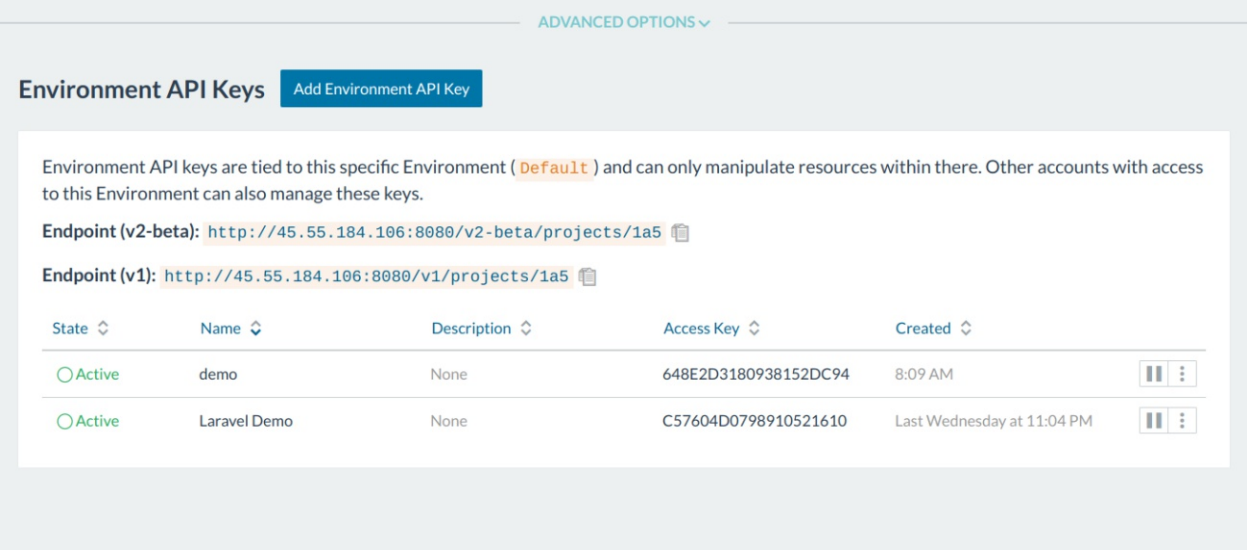


Figure 10.7: Copy the Endpoint URL (V1)

Running the Deployment

Now that we have the deploy script and keys in place, we are finally ready to automate the deployment of our stack. We need to run *rancher-compose* to create the stack via the API, and then we need to set up a load balancer in our Rancher stack to accept web traffic.

Remember to Build the Image

Make sure you build the image and push it to the GitLab registry. You can just build it manually with the *build.sh* for this chapter, or you can set up build automation as we did in Chapter 9.

Here we go! Let's use the *deploy.sh* file to run Rancher compose now. If you forgot to install Rancher Compose earlier, and you want to follow along, download the latest executable from the bottom right of the Rancher interface (Listing 10.15):

Listing 10.15: Running Rancher compose

```
# make sure deploy.sh is executable if you haven't done so
$ chmod u+x ./deploy.sh
```

```
$ ./deploy.sh \
  505D04F4DD9C9E7254FA \
  AYbu2q5CJqwpD64onzq9MkfJrjWEzmTz7ajUc255 \
  http://45.55.184.106:8080/v1/projects/1a5 \
  prod

INFO[0000] [0/1] [rancher-laravel-demo]: Creating
INFO[0001] [0/1] [rancher-laravel-demo]: Created
INFO[0001] [0/1] [rancher-laravel-demo]: Starting
INFO[0001] [1/1] [rancher-laravel-demo]: Started
```

Once again, we pass the API key, API secret, endpoint URL, and the environment name. By the time you read this my API key and the secret will no longer work, and I've used them for a complete demonstration.

If everything went according to plan, you should see a new stack if you navigate to "Stacks > User" (Figure 10.8):

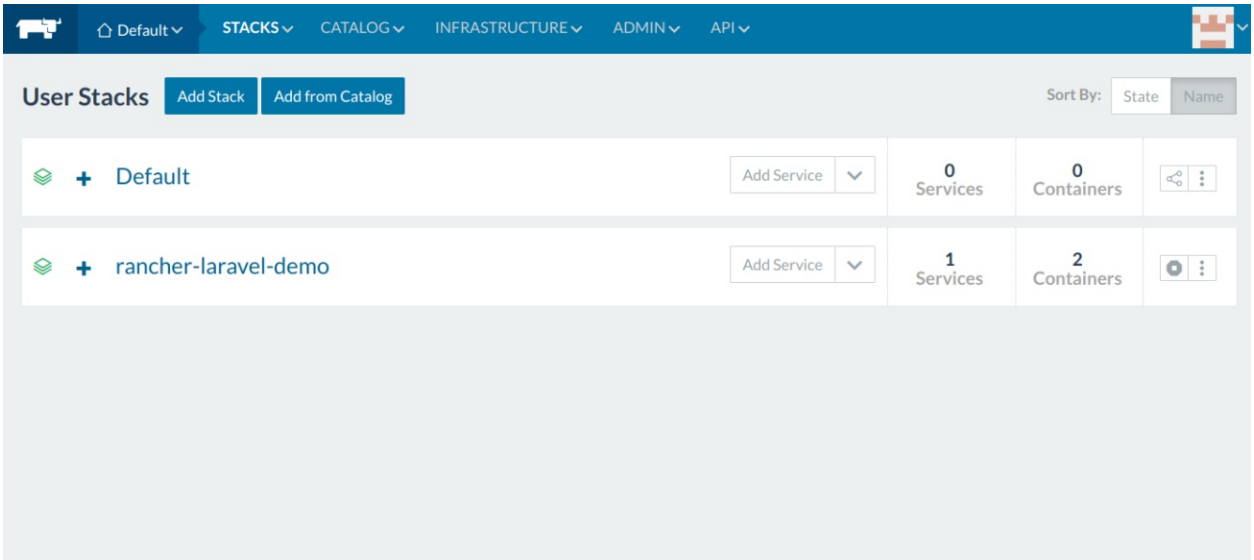


Figure 10.8: Laravel Demo Stack

Based on our Rancher Compose file we have two containers running. Click on the stack you created, and you should see that the status of the service is active.

While the stack is technically running, it's not accessible publicly yet. We need to

define a load balancer service in Rancher that sits in front of our application and routes traffic to each container instance. Think of a load balancer as gateway routing to each of your defined services that you intend to make public.

Load Balancer Service

We have two containers running Laravel, but we need to load balance traffic to them. Rancher uses HAProxy (http://www.haproxy.org/) internally to define load balancers. Simply put, the load balancer service has one IP address and behind the scenes takes care of routing traffic to your pool of application servers in the *rancher-laravel-demo* stack.

You can define a load balancer from the Rancher compose file, but before we do that, let's use the UI to create the load balancer manually. You can get pretty far with Rancher by using the UI and since this is an introductory text that's the simplest way for us to move forward. After we get the load balancer running, I'll show you how to get the equivalent Rancher/Docker compose files from the UI that you can copy and paste into your YAML configuration files.

Navigate to your user stacks from the menu by going to "Stacks > User." In the "rancher-laravel-demo" stack, you should see an "Add Service" button; click the arrow and select "Add Load Balancer" (Figure 10.9):

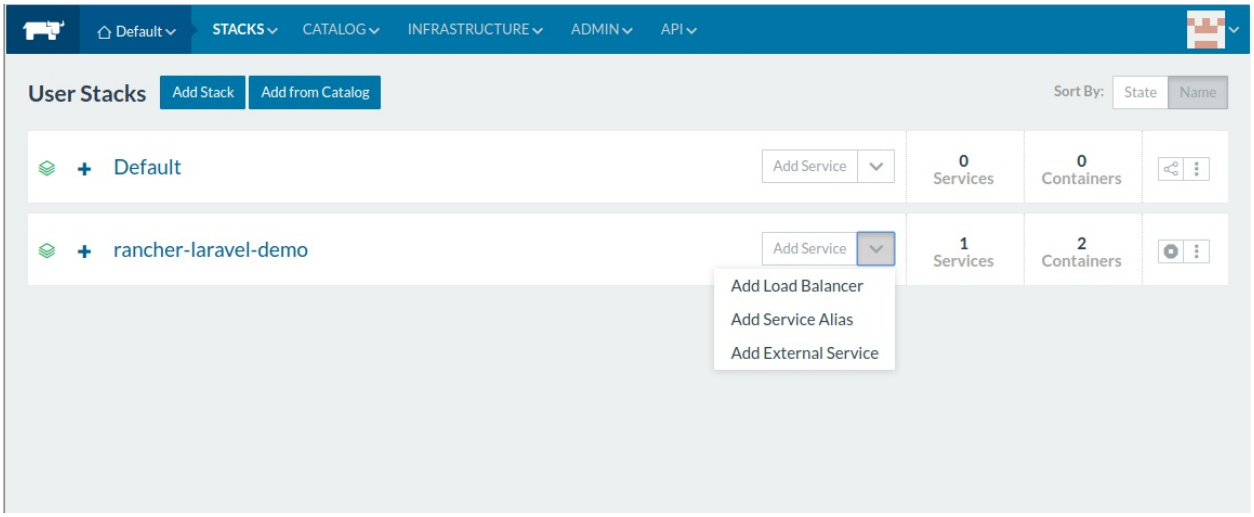


Figure 10.9: Adding a load balancer service

Adding a load balancer brings you to a setup screen. First, for the scale pick "Always run one instance of this container on every host." Next, name the service something like "laravel-demo-load-balancer" and describe if you want.

The main setup for the load balancer is defining port rules and then mapping them to other services. Our demo is simple, so we are only going to pick public HTTP on port 80, and target our rancher-laravel-demo service on port 2015. We leave the "Request Host" input empty, but generally, you would provide your application's DNS hostname here, so Rancher knows how to route it correctly when you have multiple hosts.

Rancher Help

The Rancher UI provides hints that take you to the documentation. Look for the circular question mark links ("?") near headings. The UI might change in the future, but Rancher is good about providing contextual help links.

In the end, you should have something similar to the following load balancer setup (Figure 10.10):

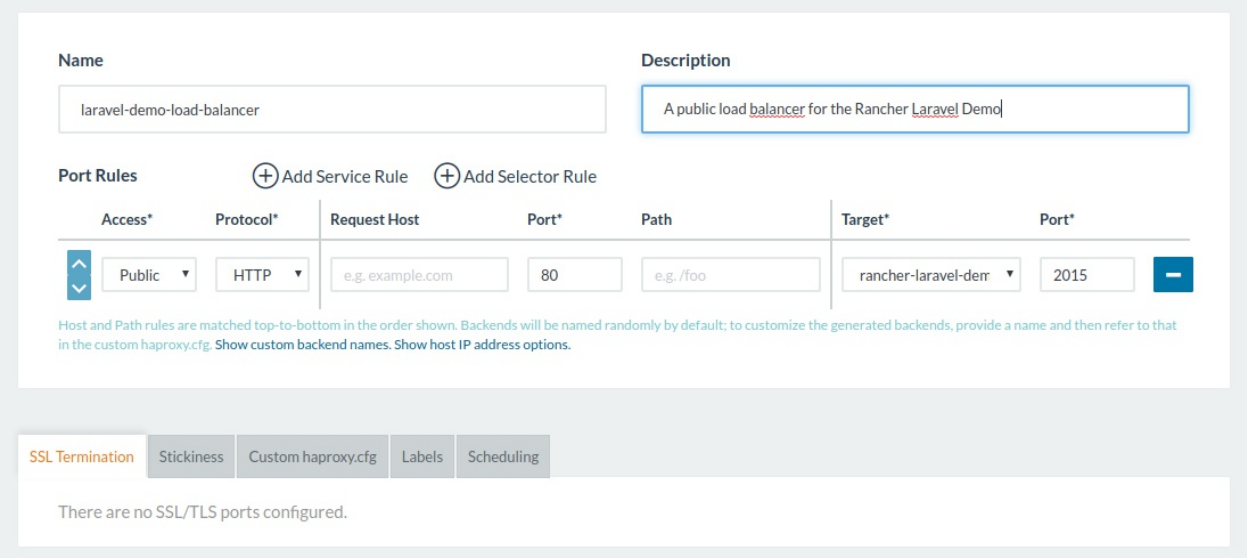


Figure 10.10: Add a Load Balancer

Once you create your load balancer service, you should see it activating from the User

stacks screen, which might take a few minutes to finish. You can click on the load balancer service to get the host IP address on the "Ports" tab (Figure 10.11).

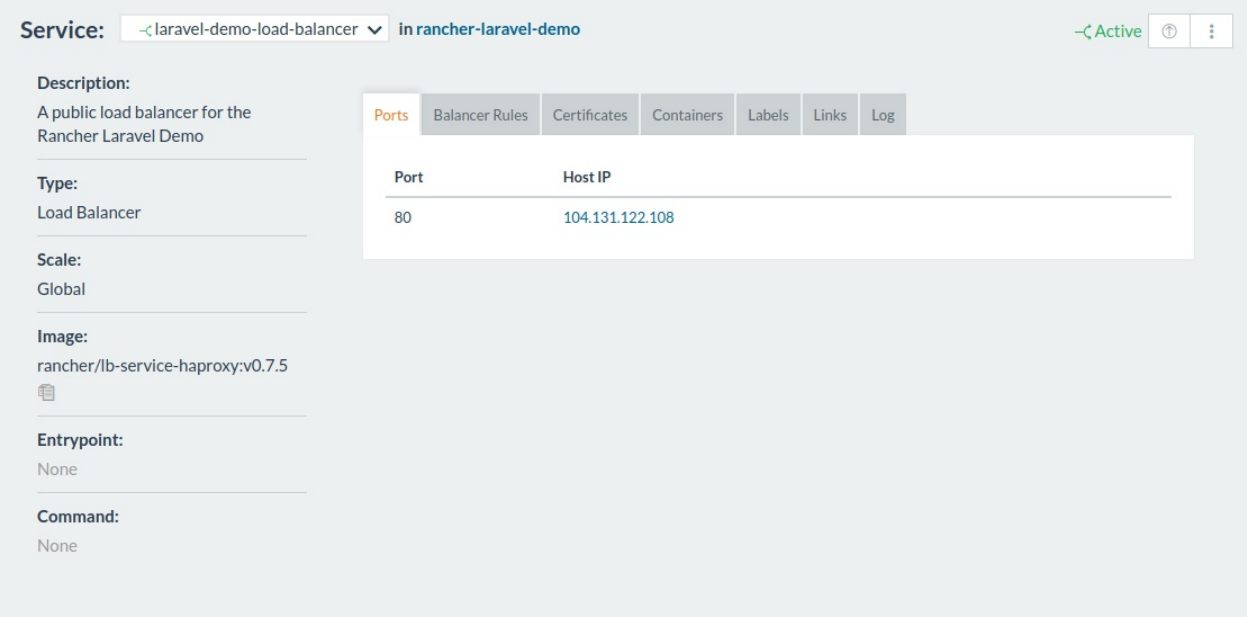


Figure 10.11: Load Balancer Service Details

If defining a hostname for the load balancer rule, you would grab the IP address from the ports tab and add an A record pointing to this IP. If you click the IP address link from the "Ports" tab in the load balancer service you should see the default Laravel homepage. We've successfully deployed a stack and made it publicly accessible!

Rancher UI Tricks

We've seen how the Rancher UI makes deploying and scaling services as simple as clicking a few buttons. One thing we haven't covered in the UI is how to take a stack or service built from the UI and recreate stacks through a Rancher Compose configuration. If you don't quite know how to automate through the Rancher Compose configuration, you can use the UI to experiment. Once you have the setup, you like you can convert it into a repeatable YAML configuration.

Creating the load balancer through the UI was a good exercise, but it breaks down when you want to re-create the stack in a repeatable way. The Rancher UI provides a way for you can copy what you've done in the UI to an equivalent YAML config.

First, navigate to "Stacks > User" and then click on the "rancher-laravel-demo" stack. The main stack page listing the load balancer and service has a document icon on the top right "Compose YAML," click it to see the files (Figure 10.12):

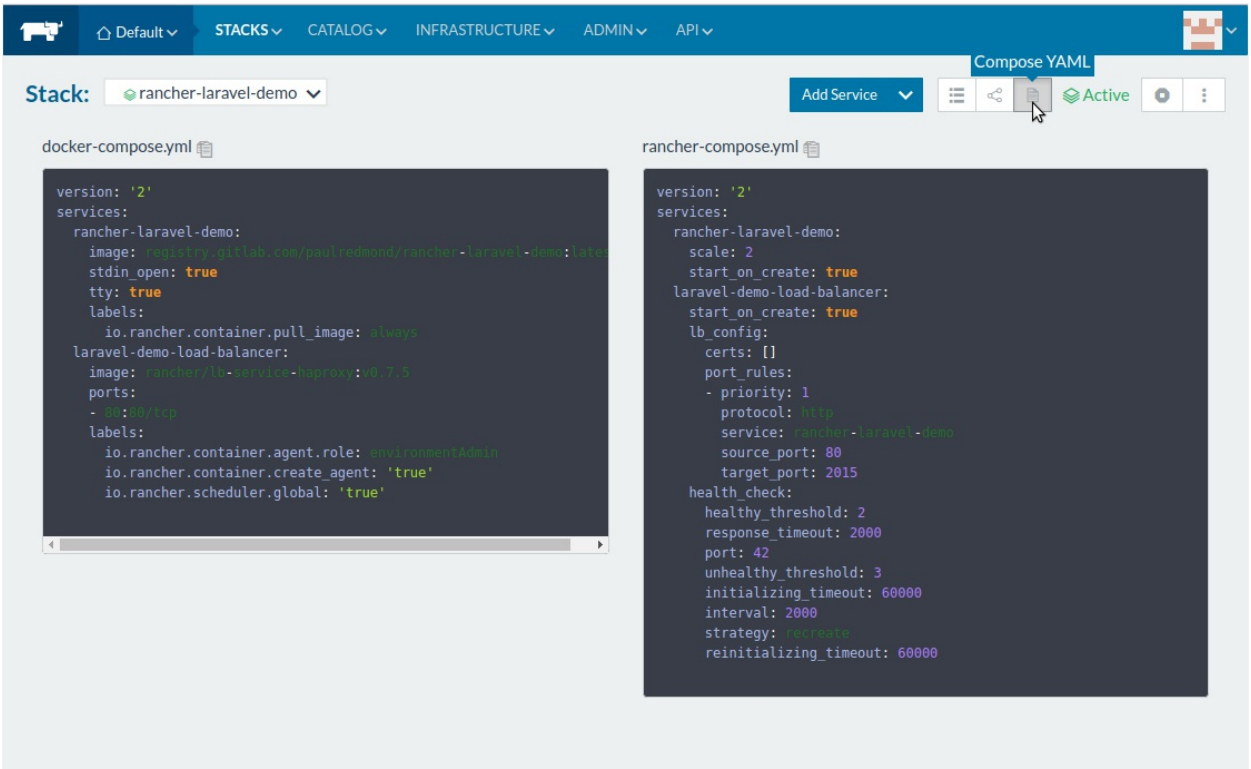


Figure 10.12: Copy the Compose YAML files

If you click the icon next to the YAML, you can copy it to your clipboard. Replace what you have in your compose files inside the project. If you keep your Rancher UI visible inside of the stack page, you can see the services update when you run them from the command line (Listing 10.16):

Listing 10.16: Running the updated Rancher Compose Script

```
$ ./deploy.sh \
  505D04F4DD9C9E7254FA \
  AYbu2q5CJqwpD64onzq9MkfJrjWEzmTz7ajUc255 \
  http://45.55.184.106:8080/v1/projects/1a5 \
  prod

INFO[0000] [0/2] [rancher-laravel-demo]: Creating
```

```
INFO[0001] [0/2] [rancher-laravel-demo]: Created
INFO[0001] [0/2] [laravel-demo-load-balancer]: Creating
INFO[0001] [0/2] [laravel-demo-load-balancer]: Created
INFO[0001] [0/2] [rancher-laravel-demo]: Starting
INFO[0001] [1/2] [rancher-laravel-demo]: Started
INFO[0001] [1/2] [laravel-demo-load-balancer]: Starting
INFO[0001] [2/2] [laravel-demo-load-balancer]: Started
```

Now you can update your entire stack with the Rancher compose command. The updated compose file allows you to re-create your stack from scratch quickly in a more repeatable fashion. In fact, let's do just that!

Delete the entire stack via the UI by going to "Stacks > User" and on the far right vertical ellipsis icon select "Delete" from the drop-down menu of the *rancher-docker-demo* stack. You can also export the YAML files from this menu too (Export Config) if you want to download the compose files. Keep the browser window open so you can magically see it re-appear when you run deploy from the command line.

If you re-run the deploy command after deleting the stack it will create the application containers and the load balancer (Listing 10.17):

Listing 10.17: Re-create the Laravel Demo Stack

```
$ ./deploy.sh \
  505D04F4DD9C9E7254FA \
  AYbu2q5CJqwpD64onzq9MkfJrjWEzmTz7ajUc255 \
  http://45.55.184.106:8080/v1/projects/1a5 \
  prod

INFO[0000] Creating stack rancher-laravel-demo
INFO[0001] [0/2] [rancher-laravel-demo]: Creating
INFO[0001] Creating service rancher-laravel-demo
INFO[0001] [0/2] [rancher-laravel-demo]: Created
INFO[0001] [0/2] [laravel-demo-load-balancer]: Creating
```

```
INFO[0001] Creating service laravel-demo-load-balancer
INFO[0002] [0/2] [laravel-demo-load-balancer]: Created
INFO[0002] [0/2] [rancher-laravel-demo]: Starting
INFO[0006] [1/2] [rancher-laravel-demo]: Started
INFO[0006] [1/2] [laravel-demo-load-balancer]: Starting
INFO[0009] [2/2] [laravel-demo-load-balancer]: Started
```

Now you have a repeatable stack, including the public load balancer. I'd encourage you to update your stack via the UI and then export/copy the compose files so you can learn how to automate what you change.

Environment Variables

If you want to set environment variables in your Rancher stack, you can use environment interpolation (http://rancher.com/docs/rancher/v1.6/en/cattle/rancher-compose/environment-interpolation/) on the machine running Rancher compose. In this chapter, you've been using your computer, but imagine a continuous integration server (i.e., Jenkins) running Rancher compose to deploy a stack.

We can use environment interpolation to our advantage to change application configuration per-environment without changing our Docker Compose file. Let's try it out by updating the *docker-compose.prod.yml* file with the following environment changes (Listing 10.18):

</> Listing 10.18: Adding an Environment Variable

```
version: "2"
services:
  rancher-laravel-demo:
    image: registry.gitlab.com/paulredmond/rancher-laravel-
demo:latest
    stdin_open: true
    tty: true
    labels:
      io.rancher.container.pull_image: always
```

```
    environment:
      RELEASE_VERSION: "${RELEASE_VERSION}"
  laravel-demo-load-balancer:
    image: rancher/lb-service-haproxy:v0.7.15
    ports:
    - 80:80/tcp
    labels:
      io.rancher.container.agent.role: environmentAdmin,agent
      io.rancher.container.agent_service.drain_provider: 'true'
      io.rancher.container.create_agent: 'true'
```

You can reference environment variables in both the Rancher and Docker Compose files defined on the machine running the commands. We added a version variable, but other environment variables could include database credentials, API keys, etc.

Let's re-run the deploy and then check the containers to make sure they have the environment variable defined (Listing 10.19):

>_ Listing 10.19: Adding the Environment Variables to Running Containers

```
$ export RELEASE_VERSION="1.0.0"

$ ./deploy.sh \
  505D04F4DD9C9E7254FA \
  AYbu2q5CJqwpD64onzq9MkfJrjWEzmTz7ajUc255 \
  http://45.55.184.106:8080/v1/projects/1a5 \
  prod

...
```

Once rancher-compose is finished, we can verify that the containers have the release version variable by running a terminal session from the UI. Navigate to your stack and select the "rancher-laravel-demo" service that has two containers. You can run a shell by clicking the far right icon on one of the containers (Figure 10.13):

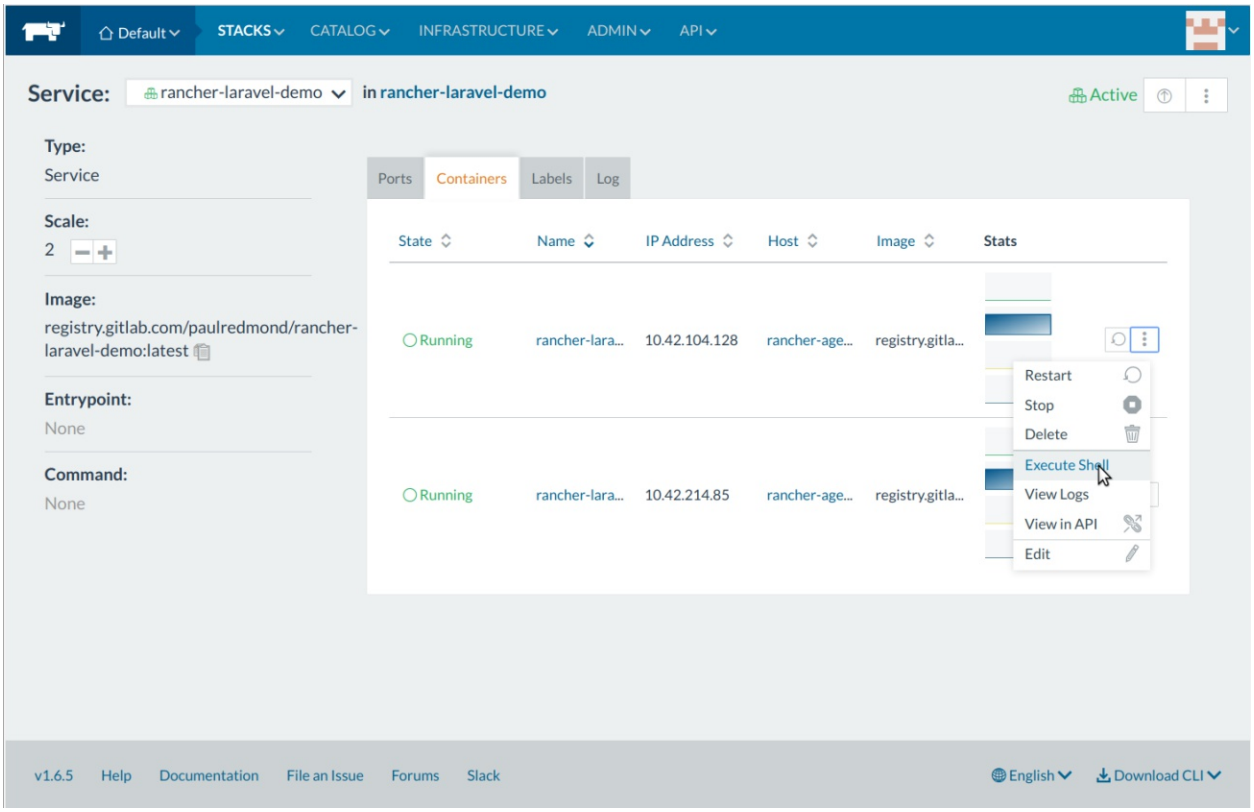


Figure 10.13: Open a container shell

In the shell, run *echo $RELEASE_VERSION* and you should see the output *1.0.0*. Environment variables are an excellent way to power configuration in our Laravel applications, and Rancher makes it a breeze to use them. The shell is also a great way to debug your stacks if you are having issues in production that you need to debug.

Homework

We've covered much ground in this chapter, but just barely scratched the surface at the same time. On your own, you can pick MySQL from the built-in catalog and install it in your cluster. Try setting it up and adding environment variables so that you can connect to MySQL from your Laravel application.

We also didn't install the Rancher server in high availability mode or use an external MySQL database to power our Rancher server. Read the installation instructions (https://rancher.com/docs/rancher/v1.6/en/installing-rancher/installing-server/) on different configurations you can use to install the Rancher server.

Learning more about Rancher secrets is a good exercise in learning more about the Rancher UI and provides a place for you to store your application secrets. You can also use a continuous integration pipeline to run rancher compose and provide your secrets from that service (i.e., GitLab or Jenkins). We've already worked with Gitlab, which you could use to automatically deploy to your stack after a merge to master or a tag.

Yeeeee-haaaa!

You now have an application running Docker in the cloud. Not too shabby! Rancher makes it simple to get going, yet, has many advanced features as you dive deeper into the documentation. We walked through how to deploy stacks with Rancher compose, use images from GitLab in our Rancher cluster, and automate deployments.

Implementing and managing Docker at scale is an advanced topic that deserves a book of its own. The appendix provides you with next steps so you can continue your learning with Docker. While you should have most of the tools you need to develop PHP applications with Docker, there's so much to learn about running Docker at scale in production.

Eject

Congratulations! Making it this far means you have a solid understanding of the fundamentals of using Docker and implementing Docker in your PHP projects. Hopefully, you've got all you need to start using Docker on your projects.

This chapter marks is the end of our journey learning how to use Docker in Development, but I hope that it's just the start of your journey with Docker. As a developer, you might be content with knowing how to work with Docker in development, and that's perfectly fine! However, if you are interested in learning more about deploying and managing Docker, I've compiled a list of resources you can use.

I have no affiliation with any of the included resources, nor do I get any kick-back for recommending them.

Docker Documentation

The official Docker documentation (https://docs.docker.com/) has a wealth of information, and you will need to refer to it often as you build out your images, Docker Compose configurations, and other topics. You can learn about Docker Swarm (https://docs.docker.com/engine/swarm/) in the official documentation.

Docker Tutorials and Labs

Docker labs is a GitHub repository (https://github.com/docker/labs) of labs and tutorials authored by Docker employees and members of the community. This GitHub repo has a beginner tutorial and a tutorial on Docker Swarm mode, and other Docker-related tutorials.

Docker Up and Running

Docker Up and Running (http://shop.oreilly.com/product/0636920036142.do) is a book by Karl Matthias and Sean Kane, published by O'Reilly Media.

Servers for Hackers

The Servers for Hackers website by Chris Fidao has some free Docker screencasts online (https://serversforhackers.com/t/containers). Chris also has an in-depth screencast series called Shipping Docker (https://serversforhackers.com/shipping-docker) which covers developing, testing, and deploying PHP applications with Docker. He is also the author of Vessel (https://vessel.shippingdocker.com/), a simple Docker development environment for Laravel.

Kubernetes

Kubernetes (https://kubernetes.io/) is an open-source project for automating deployment, scaling, and management of containerized applications. The official site has plenty of detailed documentation (https://kubernetes.io/docs/home/), and an interactive tutorial (https://kubernetes.io/docs/tutorials/kubernetes-basics/).

Kubernetes might be a bit of a deep-dive depending on how much you know about deploying server-based applications, so you might also benefit from Kubernetes Up and Running (http://shop.oreilly.com/product/0636920043874.do) by Kelsey Hightower, Brendan Burns, and Joe Beda.

Google Cloud

Google Cloud (https://cloud.google.com) is an excellent place to experiment with Kubernetes using Google's managed Kubernetes installation. Google provides a free tier (https://cloud.google.com/free/) to get started with Google's cloud technologies. At the time of writing the free tier gives you 12 months and a 300 dollar credit to try out their products and they offer an always free tier of specific products.

Another benefit of using Google Cloud is Google's container registry product for storing

and deploying private Docker images. Google's container registry is very fast, and you can use the registry with Google's Kubernetes engine with no authentication setup required.

I recommend a few PHP-specific tutorials that will help you learn using PHP and Docker in Google Cloud. First, go through the Create a Guestbook with Redis and PHP tutorial (https://cloud.google.com/kubernetes-engine/docs/tutorials/guestbook), which shows you how to run a simple PHP application with Redis on a Kubernetes. Second, the Running a PHP Bookshelf on Kubernetes Engine tutorial (https://cloud.google.com/php/tutorials/bookshelf-on-kubernetes-engine).

The Twelve-Factor App

The Twelve-Factor App (https://12factor.net/)—written by Heroku Co-Founder Adam Wiggins—is a document that describes a specific methodology for building web applications, or software-as-a-service that is suitable for deployment on modern cloud platforms.