BitPress

DOCKER

For PHP Developers

BY PAUL REDMOND

A Guide to Using Docker for PHP Development

Docker for PHP Developers

A guide to using Docker for PHP development

Paul Redmond

This book is for sale at http://leanpub.com/docker-for-php-developers

This version was published on 2020-01-08

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Paul Redmond by spreading the word about this book on Twitter!

The suggested tweet for this book is:

https://twitter.com/paulredmond/status/955815200166326272

The suggested hashtag for this book is #dockerphp.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#dockerphp

Contents

Introduction

Docker is a software container platform that allows you to run isolated applications that are highly repeatable between different environments. It's different from virtual machines in the fact that it's not a full operating system. Docker provides the essential libraries needed to run software that will run the same on Windows, Mac OS X, and Linux operating systems.

Docker's adoption and growth over the last couple of years is staggering, along with a booming ecosystem surrounding everything from container management, build pipelines, and developer tools for every major platform.

I believe now is the perfect time to pick up Docker and see for yourself how it can improve your PHP development environment.

Why is that?

Docker is Lighter than Virtual Machines

Software tools like Vagrant make it easier to produce repeatable *development* environments through a virtualized image. Vagrant is still very much a tool with valid uses, don't fall into the trap of pitting them against each other. The Vagrant website[1] has a concise write-up that compares Vagrant. Both solutions aim to provide repeatable, consistent environments, and both have a large community ecosystem.

One thing I like about Docker is how quickly you can start and stop containers. I like fast and local development environments, and Docker feels more natural in that way. Building Docker environments are faster than provisioning a Vagrant box, and I also like that production images will be identical to my local images.

In my opinion, Docker encourages developers to play with software on their machine without the fear of installing it locally. You will be more confident in your platform and thus will feel encouraged to improve upon it. Upgrading to new versions of PHP will be more reliable because you will eventually ship the same image to production without surprises.

Docker is Maturing

In the early days of Docker support was limited on Mac OS X and Windows platforms. On OS X, tools like "boot2docker" provided Docker support through a virtual OS image. An official Docker toolbox[2] package for Mac still provided Docker through a virtual Linux server, but Docker for Mac now gives you a truly native Docker experience.

[1] https://www.vagrantup.com/intro/vs/docker.html
[2] https://www.docker.com/products/docker-toolbox

In July 2016, Docker announced stable versions[3] of Docker for Mac and Docker for Windows to provide native virtualization experiences to match Linux.

Considering that all three major software platforms can natively run Docker, using it as a development tool is vastly easier. Removing the virtual machine aspect of Windows Pro and Mac makes running Docker as fast as running it on a Linux OS.

At the same time that the Docker platform has been maturing, automation tools for orchestrating containers have evolved too. Initially, a project called fig[4] made it easier to run and link containers you were running locally. Fig evolved into the officially supported Docker compose[5] tool distributed with Docker on all platforms.

Who is this Book for

This book is for PHP developers that haven't had exposure to Docker yet or current Docker users that want an opinionated text on how to develop applications locally with PHP and Docker.

A working understanding of PHP, experience running commands from a command line interface (CLI), and a basic understanding of setting up a PHP project are assumed. Even if you haven't set up a complete PHP environment, the examples in the book should be complete enough that you can follow along.

This book's primary focus is on using Docker as a PHP development environment. I cannot deny that deployment is an important part of learning Docker, but this text cannot accomplish the goal of focusing on development and give deployment justice. We do, however, cover deployment of Docker to Digital Ocean using Rancher to get you started. We will provide plenty of "where to go next" references to explore at the conclusion of this book too, but consider this book's focus on getting you comfortable and confident using Docker for development.

Conventions Used in this Book

This book is a hands-on guide to using Docker as a PHP developer, and thus it's important to understand the conventions used to display the source code, command line examples, and how to submit code errata.

Code Examples

A typical PHP code snippet looks like this:

[3] https://blog.docker.com/2016/07/docker-for-mac-and-windows-production-ready/
[4] http://www.fig.sh/
[5] https://docs.docker.com/compose/

Example PHP Code Snippet

```
/**
 * A Hello World Example
 */
$app->get('/', function () {
    return 'Hello World';
});
```

To guide readers, new code added to an existing file will be bold:

Example PHP Code Snippet

```
/**
 * A Foobar Example
 */
$app->get('/foo', function () {
    return 'bar';
});
```

Longer lines end in a backslash (\) and continue to the next line:

Example of Long Line in PHP

```
$thisIsALongLine = 'Lorem ipsum dolor sit amet, consectetur adipisicing elit. Quos u\
nde deserunt eos?'
```

When you need to run terminal commands to execute the test suite or create files, the snippet appears as plain text without line numbers. Lines starting with `$` which represents the terminal prompt.

Example Console Command

```
$ touch the/file.php
```

Console commands are sometimes executed locally on your development machine and other times within a Docker container. The text tries to emphasize both cases so you are aware of where you should be running a given command.

Console Command in a Docker container

```
$ docker exec -it c0ee14f0c047 bash
# Inside the container, run php --ini
root@c0ee14f0c047:/var/www/html# php --ini
```

Code Errata and Feedback

Submit errata to errata@bitpress.io. Feel free to send typos, inaccurate descriptions, code issues, praise, feedback, and code suggestions on better ways of doing something. Please don't be shy, **these things make the book better**!

Tools

We will cover installing Docker and other tools needed to work with Docker. There are other recommended tools not included in the text that will help you along the way to learning Docker.

Command Line

I can only make a few recommendations here. If you are on Linux, you already have an excellent shell. On Mac OS X, I prefer to use Iterm2[6]. Hyper[7] is another terminal that works on Mac, Linux, and Windows.

Version Control

If you want to work along in the book and commit your code as you go (recommended) you need to install a version control system. I recommend git[8], but anything you want will do.

Editor / IDE

Most readers will already have a go-to editor. I highly recommend PhpStorm[9]—which is not free—but it pays for itself. Other common IDE options are Eclipse PDT[10] and NetBeans[11].

If you don't like IDE's, I recommend Sublime Text[12] or Atom[13] or Visual Studio Code[14]. They are all great editors and have Docker support.

[6] https://www.iterm2.com/
[7] https://hyper.is/
[8] https://git-scm.com/
[9] https://www.jetbrains.com/phpstorm/
[10] https://eclipse.org/pdt/
[11] https://netbeans.org/
[12] https://www.sublimetext.com/
[13] https://atom.io/
[14] https://code.visualstudio.com/

About the Authors

Paul Redmond

Salut, Hoi, Hello

I am a web developer writing highly available applications powered by PHP, JavaScript, and RESTful Web Services. I am the author of the self-published Writing APIs with Lumen[15] which is now published as Lumen Programming Guide [16] (Apress). I also write for Laravel News[17].

I live in Scottsdale, Arizona with my wife and three boys. I enjoy reading, writing, movies, golf, and basketball.

I'd love for you to follow me on Twitter @paulredmond[18]. Please send me your feedback (good and bad) about the book so that I can make it better. If you find this book useful, please recommend it to others!

BitPress

BitPress provides high-quality screencasts, books, and training materials for PHP, Java, and DevOps. You won't find boring presentations or hard-to-hear talks ported to video. Dive into engaging one-on-one coding screencasts and books.

Visit BitPress.io[19] to sign up for our newsletter to receive insider information, tutorials, and new product launches.

[15] https://leanpub.com/lumen-apis
[16] http://www.apress.com/book/9781484221860
[17] https://laravel-news.com/@paulredmond
[18] https://twitter.com/paulredmond
[19] http://bitpress.io

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[20]. 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[21] 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):

[20] https://hub.docker.com/_/php/
[21] https://docs.docker.com/engine/reference/builder/

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 name|id`:

```
# 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[22], and you should see the output from phpinfo() (Figure 2.1):

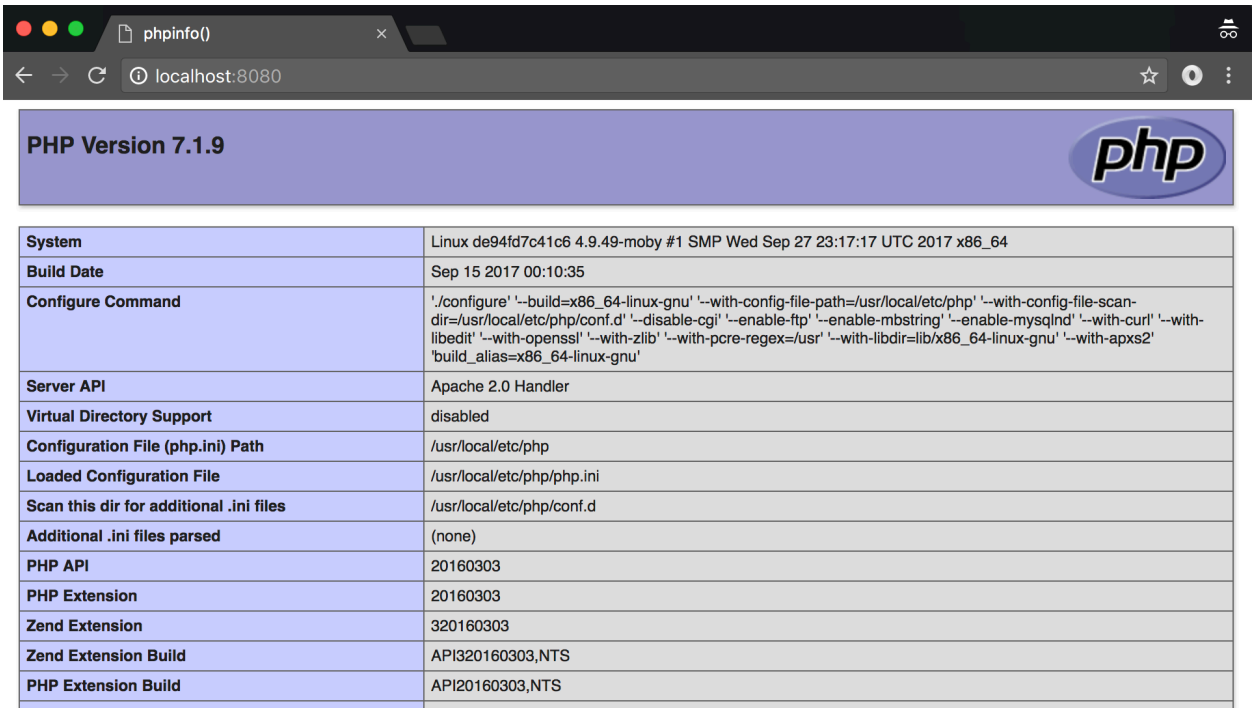

Figure 2.1: phpinfo

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.

[22] http://localhost:8080

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[23]:

> 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[24], 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):

[23] https://docs.docker.com/compose/overview/
[24] https://docs.docker.com/compose/

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[25], 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[26].

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

[25] https://docs.docker.com/compose/compose-file/
[26] http://localhost:8080

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

```
$ 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 Dockerfile (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!