Setting Up a Local PHP 7 Development Environment With Docker & Compose

Some operating systems come with a preinstalled version of PHP. You can determine if you have PHP installed by opening up your terminal or command line interface and typing the following: `php -v`

This may return something like this.

```
➜  ~ php -v
PHP 5.5.36 (cli) (built: May 29 2016 01:07:06)
Copyright (c) 1997-2015 The PHP Group
Zend Engine v2.5.0, Copyright (c) 1998-2015 Zend
Technologies
```

If PHP is installed, you might have noticed the version is not the latest version available. Instead of trying to update this PHP, we will move forward with using Docker.

Docker will help us set up a contained and platform-independent development environment that will closely, if not exactly, match our production environment.

Keeping this in mind, we can install exactly the server operating system, PHP version, database type, and many other options.

Installing Docker

Let's jump right in and install Docker locally. I will be using OS X for my install, but since Docker works the same on Windows and Linux, it will not be challenging to work along with me. Head over to [Docker's Website] and follow the prompts to download the correct installer for your operating system.

Open the downloaded file and follow the prompts to install Docker. You will most likely want to leave everything at their default values.

Once you have installed the Docker native application, you should be able to open up a terminal and type `docker info`, which will output something that will look like this.

```
➜  ~ docker info
Containers: 0
Running: 0
Paused: 0
Stopped: 0
```

Your numbers may be different, but that's okay—we are just validating that we can access Docker from our terminal.

If you have any issues getting it up and running, here are the getting started links from Docker for two of the major operating systems:

https://docs.docker.com/docker-for-mac/

https://docs.docker.com/docker-for-windows/

Creating a Local Development Environment

Now that we have Docker installed and working, it is time to create our local development environment.

Docker has a lot of options to use and understand, which in time you will want to understand, but for now you can download or clone a repository that has been set up for you.

Clone or download the following project onto your computer at the location where you would like to keep all of your PHP development projects.

https://github.com/hamptonpaulk/php7-dockerized

Once it is cloned, or downloaded and extracted, change directory into that folder.

Let's walk through these folders to get a basic understanding of the function of each.

CONF: The conf folder holds the configuration files for some of the server technologies. [Nginx] (the web server), PHP (the interpreter), and [Supervisord] (a process monitor) each have their own file here to configure to our needs.

LOGS: The logs folder holds all the logs for each of our containers, such as PHP and nginx.

SITES: The sites folder holds the nginx configuration for each of the websites we want to work on. In this development environment we are configuring, each new site will have its own vhost file that matches the folder inside of www/.

WWW: The www folder holds all of our site files—each website will have its own folder under www. Each folder will have a matching vhost file in the sites folder. This is to our advantage as each site is contained and configured independently of the others.

DOCKERFILE: The next couple of files are the two that interact directly with Docker to create and build our server. The Dockerfile is the base configuration of our server and what we want to install on it. It also copies all of our local configuration files onto the server. There is a lot to learn about Dockerfiles, so when you have time and lots of coffee, visit the [Dockerfile Documentation].

DOCKER-COMPOSE.YML: The docker-compose.yml file is our configuration and interface for using Docker Compose. When we are using Docker Compose, we are specifying how our application is structured and each item linked together. Docker Compose is a powerful tool that allows us to run some simple command in our terminal to build, start, and stop our server. The Docker Compose workflow comes down to three basic steps:

1. Define your app's environment with a Dockerfile so it can be reproduced anywhere.
2. Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
3. Lastly, run `docker-compose up` and Compose will start and run your entire app.

Let's take a quick look at the Dockerfile and the docker-compose.yml.

Dockerfile

`FROM nginx` This line is setting up our base container, which is a pre-built nginx container running Alpine. You can see the official images such as nginx at hub.docker.com.

```
# Remove default nginx configs.
RUN rm -f /etc/nginx/conf.d/*
```

Any lines with a # in front is a comment. Here we are to RUN a command just like we would on a unix command line, removing all folders and files within the specified path.

```
# Install PHP 7 Repo
RUN apt-get update && apt-get install -my wget
RUN sh -c "echo 'deb http://packages.dotdeb.org jessie
all' >> /etc/apt/sources.list"
RUN sh -c "echo 'deb-src http://packages.dotdeb.org
jessie all' >> /etc/apt/sources.list"
RUN wget https://www.dotdeb.org/dotdeb.gpg -O - | apt-
key add -
RUN wget https://nginx.org/keys/nginx_signing.key -O - |
apt-key add -
```

Here we are installing a program called wget to fetch some keys so that we can install PHP 7 from a custom url as well as updating a key for nginx.

```
#Install packages
RUN apt-get update && apt-get install -my \
    supervisor \
    curl \
    php7.0-curl \
    php7.0-fpm \
    php7.0-mysql \
    php7.0-mcrypt \
    php7.0-sqlite \
    php7.0-xdebug \
    php-apc
```

Here we are running an update and then an install of several applications and packages using the unix package manager apt.

```
# Ensure that PHP7 FPM is run as root.
RUN sed -i "s/user = www-data/user = root/"
/etc/php/7.0/fpm/pool.d/www.conf
RUN sed -i "s/group = www-data/group = root/"
/etc/php/7.0/fpm/pool.d/www.conf


# Prevent PHP Warning: 'xdebug' already loaded. XDebug
loaded with the core
RUN sed -i '/.*xdebug.so$/s/^/;/' /etc/php/7.0/mods-
available/xdebug.ini
```

Next, we are updating the user and group for PHP5 FPM in the existing configuration files on the server.

```
# Add configuration files
COPY conf/nginx.conf /etc/nginx/
COPY conf/supervisord.conf /etc/supervisor/conf.d/
COPY conf/php.ini /etc/php/7.0/fpm/conf.d/40-custom.ini
```

These last lines are making a COPY of the local configuration files for PHP, nginx, and supervisord.

All of this in the Dockerfile is setting up the base configuration for our server. Once this is built, we only need to revisit it if we change our three configuration files above, or need to install some new packages.

docker-compose.yml

```
version: '2'
    services:
```

Our services in this case are PHP and MySQL.

```
php:
        build: .
```

'build .' is building out our container, that was defined in our Dockerfile.

`command: /usr/bin/supervisord -c /etc/supervisor/conf.d/supervisord.conf` then command runs whatever single command we want to trigger when the container is started.

```
ports:
        - "80:80"
        - "443:443"
```

ports exposes whatever ports we want to make accessible from our host computer inside of the Docker container.

```
links:
        - mysql
```

links sets up—as you may have guessed it—a link between the PHP container and the MySQL container so that one may see the other.

```
volumes:
        - ./www:/var/www
        - ./sites:/etc/nginx/conf.d
        - ./logs:/var/log/supervisor
```

volumes sets up a connection between our local file system and our containers file system. The format is pretty simple: `/local/location : /container/location.`

```
mysql:
        image: mysql
        ports:
            - "3306:3306"
    environment:
        MYSQL_ROOT_PASSWORD: password
```

For the mysql service we are using a predefined image from Docker, exposing the default mysql port, and setting a default root mysql password in the container's environment variables.

Build & Run Containers

The next step for us is to build our container by running `docker-compose build` within the folder where the Dockerfile and docker-compose.yml files are located.

```
➜  docker-compose build
mysql uses an image, skipping
Building php
…
Successfully built fe03206fb1a1
```

Once successfully built, which could take a while, you will need to bring your server up. Run `docker-compose up`

Now head over to the browser and navigate to 0.0.0.0.

You should see a PHP info page.

Docker-Compose Commands

Within Compose there are quite a few commands that can be used from the command line. We will review the most common ones below, but you can find a full listing with a good bit of detail here https://docs.docker.com/compose/reference/

`docker-compose up`
https://docs.docker.com/compose/reference/up/

This is one of the most commonly used commands which 'builds, (re)creates, starts, and attaches to containers for a service.' If you run this command with the detatched flag `docker-compose up -d` it will run the processes in the background.

`docker-compose start/stop/restart`
https://docs.docker.com/compose/reference/stop/

This is a common flow for us once we have used the up command. `docker-compose stop` Stops running containers without removing them. They can be started again with `docker-compose start` , and as you might expect we can always reboot/restart when needed with `docker-compose restart`

`docker-compose down`
https://docs.docker.com/compose/reference/down/

This command 'Stops containers and removes containers, networks, volumes, and images created by `docker-compose up` .' We will not always use this command, mostly you will use the start/stop/restart flow from above.

`docker-compose run`
https://docs.docker.com/compose/reference/run/

Run is a very helpful command—it runs a one-time command against a service. A very common use is to run bash on the remote service container. For our service 'php', which is the name we defined in the docker-compose.yml file, we would run the following.

`docker-compose run php bash` which is the run command followed by the container name followed by the command, which in this case is bash or /bin/bash. The run command has quite a few option flags that you can use, so make sure to read up on them at the link above.

`docker-compose exec`
https://docs.docker.com/compose/reference/exec/

If you want to run a command on an existing and running container, you can use the exec command.

As an example, let's get the php version on our container.

`docker-compose exec php php -v` Breaking down this command is very similar to the run command—we are using the php service container, followed by the php command with the argument -v to check the version.

Now that we have a good understanding of the docker-compose suite of commands, let's talk about workflow.

Adding a New Site

We will for sure need to add a new site at some point, so let's add an example site called 'hello'.

www folder

We will want to contain the site without killing our default site, which is the php info page. This will allow us to have a separate SCM repository for each of our sites.

We will be keeping our project files in the www folder, so let's start with adding a new folder named hello in the www folder and add an index.php file in that hello folder.

For the php file, index.php, let's add <?php echo "Hello World"; ?> just to make sure it's working. Right now, it won't work—until we tell nginx that we have a new site.

sites folder

Our sites folder is where we will define each site on the container. The best option is to create a duplicate of default.vhost and rename it hello.vhost.

The name of this file should match the name of your site.

There are quite a few options we can set in this file, and it would be a good idea to review all of the options that you can place into a .vhost file. You can find all of that info on the nginx site.

For our example, we will just be focusing on a few of the options defined here. The first being server_name:

The nginx documentation explains the usage of server_name, saying: "If there are several servers that match the IP address and port of the request, NGINX Plus tests the request's Host header field against the server_namedirectives in the server blocks."

So in our case, we will set server_name to hello.

The next two options are 'root' and 'index', where root tells nginx where to look for the site files and the index sets up the name of initial file to serve.

Here are our settings:

```
server_name hello.dev;
root        /var/www/hello;
index       index.php;
```

editing the hosts file

The final step in this setup is to edit our computer's hosts file. The hosts file will allow us to set up static URLs and map them to an IP address. We will want to create a site URL for hello.dev and set the IP address to 127.0.0.1, or our local host.

Each operating system is different in how it handles hosts files and how to modify them. I will be using OS X as the example here. If you need a different operating system, rackspace.com has a nice guide. [https://support.rackspace.com/how-to/modify-your-hosts-file/]

In OS X, open a terminal and run the following command: `sudo nano /etc/hosts`.

You will need to supply your password since you are modifying a protected system file. You can use any editor you like, but you will need to elevate your permissions beforehand. If you are using atom.io, for instance, you can run the same command replacing 'nano' with 'atom'.

Your file should look a bit like this.

```
##
# Host Database
#
# localhost is used to configure the loopback interface
# when the system is booting.  Do not change this entry.
##
127.0.0.1                localhost`
255.255.255.255  broadcasthost
::1                           localhost`
```

Underneath the localhost entry, add the following:

```
127.0.0.1 hello.dev
```

Then save the hosts file by pressing Control+x and answering y, or yes, to the save location, and press enter/return.

You may need to run `dscacheutil -flushcache` to refresh the changes.

The final step will be to run `docker-compose restart` to make the changes to nginx configs update.

Now we can browse to http://hello.dev and should see 'Hello World'.

Now that we have the environment set up, it will be easy to set up new sites in the future and develop with near-production products, no matter your operating system. Let us know how it went or ask any questions you may have in the comments below!