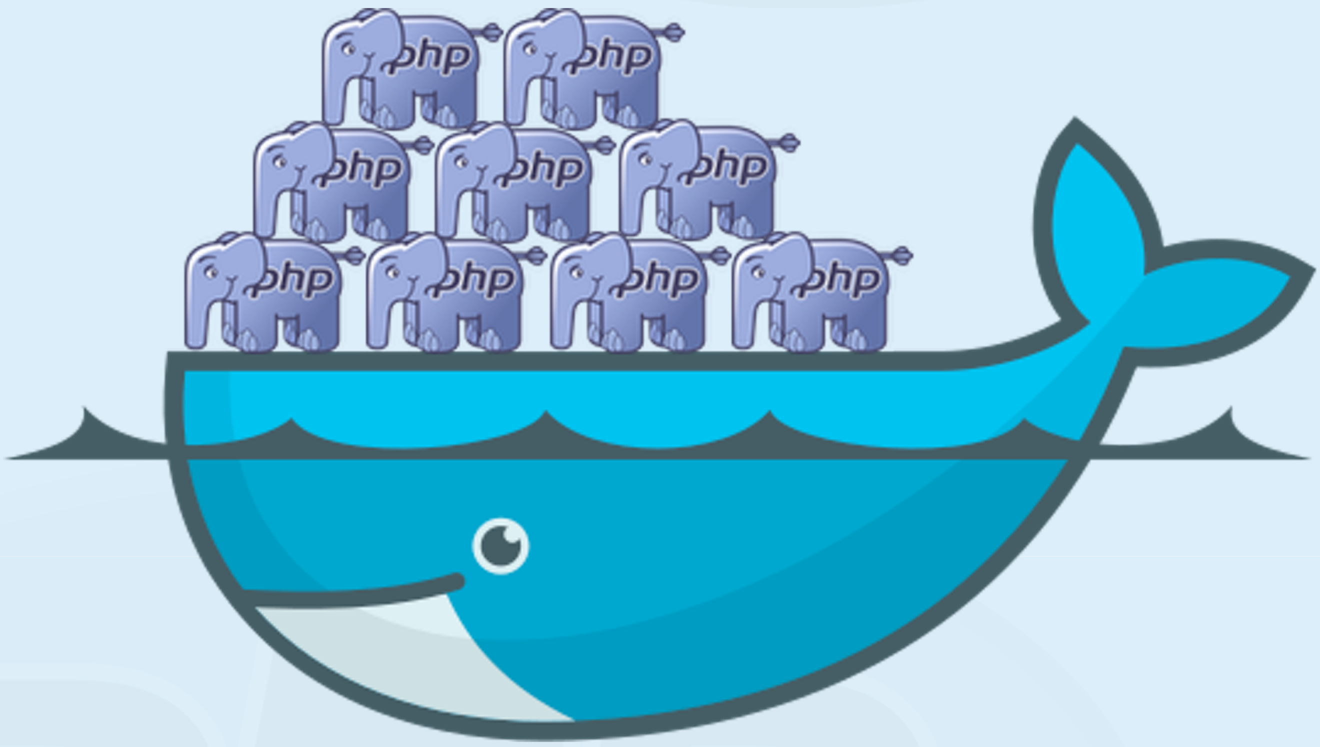
DOCKER: From Development to Deployed!

A short guide for **PHP** developers



By Matthew Setter

For Docker Engine 19.03.13

Developing With Docker

Matthew Setter

Version , December 02, 2020

Table of Contents

How To Build a Local Development Environment Using Docker

Building a local development environment which mirrors production hasn't, historically, been an easy task. But with Docker, it's become, virtually, trivial. Come learn how to setup a local development environment using Docker for developing Zend Expressive (and other PHP) applications.

Developers, Developers, Developers. It's not easy to be one; am I right?

While it's a challenge which we love, it's still a tough gig. There are so many things to know; from frameworks and software design patterns to deployment and scaling techniques. Our time's stretched thin staying abreast of everything.

But somehow we manage. Somehow, we keep our heads above water and survive. Given all this, the last thing we want to do is waste our precious time on anything which isn't productive.

And something which isn't productive is setting up a development environment. Why in this modern day and age is setting up a development environment still such a complicated process?

Creating Local Development Environments Is Challenging

Why is it still so hard to get one setup that works, that does what you need, and that matches the deployment environment's of testing, staging, production and so on?

Here's what I mean. We start off using the native tools available on our operating system of choice. We next likely start using LAMP, MAMP, and WAMP stacks.

After we've exhausted these, we usually progress to Vagrant & VirtualBox VM's—after learning one (or more) provisioning tools, such as Chef, Puppet, or Ansible.

By now our development environments have grown quite sophisticated. But the overhead of both building and maintaining them has increased significantly also.

Wouldn't it be easy if we could set them up, but with only a small investment of time and effort? I think you know where I might be heading with this.

You can. Yes, that's right, you can, with Docker.

Enter Docker

That's right - Docker. Docker was initially released back in March of 2013 by a hosting company called dotCloud. dotCloud had been using the tool internally to make their lives easier managing their hosting business.

To quote Wikipedia:

> Docker is an open-source project that automates the deployment of Linux applications inside software containers.

Here's a longer description:

> Docker containers wrap up a piece of software in a complete filesystem that contains everything it needs to run: code, runtime, system tools, system libraries—anything you can install on a server. This guarantees that it will always execute the same, regardless of the environment it is running in.

Docker Is Simpler

Now you might be thinking that this all sounds similar to everything else you've used, such as a LAMP stack or a Vagrant/VirtualBox VM.

In a way it is. But it's also a lot less resource and time intensive. As the quote above summarizes, Docker contains—and uses—**only** what it needs to run your application—nothing more.

You're not building a big virtual machine which will consume a good chunk of your development machine's resources. You don't have to learn—and write—massive configuration setups to build a basic, working, setup.

You don't need to do much at all to get your up and running. Docker allows you to build your application infrastructure, as you would your code. You determine the parts and services you need and stack them together like Lego blocks.

If you need to change your web server or database server, then switch the current one out for another. Need to add a caching, logging, or queueing server? Add it into the mix and keep on going. It is that simple.

Sound enticing? I hope so.

How to Build a Development Environment with Docker

If you're keen to find out the latest, and best, way to create a development environment, one which you can be up and running within less than 20 minutes, let's get started.

In this tutorial I'm going to show you how to build a local development setup, using Docker, to run a Zend Expressive app, based on the Zend Expressive Skeleton Installer.

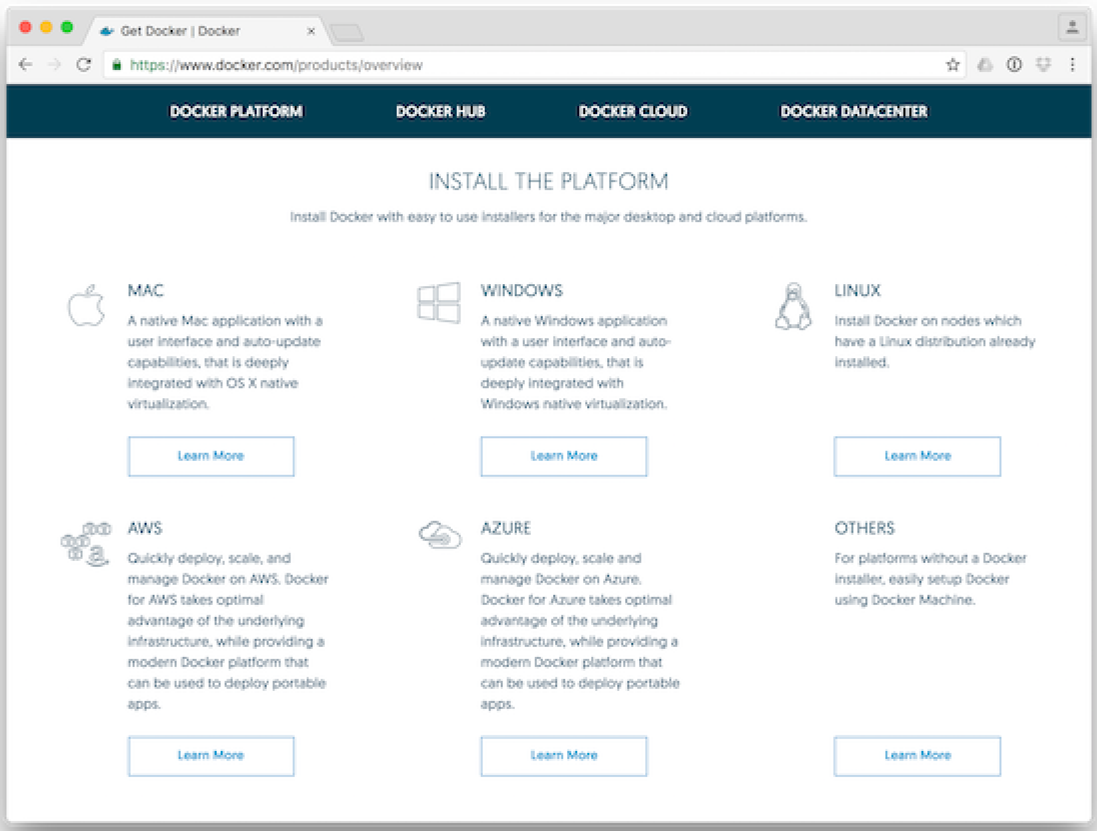
> It would likely work for any Zend Expressive, or PHP, application for that matter.

Here's how it will work; we'll have one container for PHP, one container for Nginx, and one container for MySQL. Our configuration will bind them altogether so that, when finished, we can run one command line script to build it, boot it, and view it locally on port 8080.

Let's begin.

> This is not an extensive tutorial on Docker. If you want to learn all there is to know, I strongly encourage you to check out either Learning Containers, by Chris Tankersley, Shipping Docker by Chris Fidao, or buy Docker for Developers. They're all excellent resources!

Installing Docker



INSTALL THE PLATFORM

Install Docker with easy to use installers for the major desktop and cloud platforms.

MAC

A native Mac application with a user interface and auto-update capabilities, that is deeply integrated with OS X native virtualization.

WINDOWS

A native Windows application with a user interface and auto-update capabilities, that is deeply integrated with Windows native virtualization.

LINUX

Install Docker on nodes which have a Linux distribution already installed.

AWS

Quickly deploy, scale, and manage Docker on AWS. Docker for AWS takes optimal advantage of the underlying infrastructure, while providing a modern Docker platform that can be used to deploy portable apps.

AZURE

Quickly deploy, scale and manage Docker on Azure. Docker for Azure takes optimal advantage of the underlying infrastructure, while providing a modern Docker platform that can be used to deploy portable apps.

OTHERS

For platforms without a Docker installer, easily setup Docker using Docker Machine.

I'll assume that you don't have Docker installed on your local machine. If not, and you're using a Linux distribution, then use its package manager to install Docker.

If you're not using Linux, then grab a copy of Docker for Mac or Docker for Windows, depending which platform you're using. The installers do an excellent job of making the setup pretty painless.

With Docker installed, we're now able to start building our setup.

The Docker Setup

Most PHP applications, at their most basic, are composed of three parts:

- A web server (commonly Nginx or Apache)
- A PHP runtime (most often using PHP-FPM these days)
- A database server (usually MySQL, PostgreSQL, or SQLite)

This can be visualized in the illustration below. Sure, there are a host of other components, such as ElasticSearch, caching, and logging servers, but I’m sticking to the basics.

[Visualisation of a location development environment using Docker] | *docker-local-*

development-environment-simple-illustration.jpg

Our setup's going to mirror that, having a container for each component which we've listed above. Let's start with the web server configuration.

The Web Server Container

In the root directory of your project, create a new file, called docker-compose.yml. In there, add the following configuration:

```
version: '2'

volumes:
  database_data:
    driver: local

services:
  nginx:
    image: nginx:latest
    ports:
      - 8080:80
    volumes:
      - ./docker/nginx/default.conf:/etc/nginx/conf.d/default.conf
    volumes_from:
      - php
```

The configuration starts off by specifying that we're using version 2 of the Docker Compose file format. This is important as using version 2 requires less work on our part, in comparison with version 1.

It next sets up a persistable filesystem volume, which will be used later in the MySQL container. This is important to be aware of as, by default, filesystems in a Docker container are setup to be read-only.

Given that, any changes made aren't permanent. When a container restarts, the original files will be restored and any new files will be removed. Not a great thing when working with databases, or other storage mechanisms.

We next define an element called services. This element lists the definitions of the three containers which will make up our build, and start defining the Nginx container.

What this does is to create a container called nginx, which can be referred to by the other containers using the hostname nginx. It will use the latest, official, Docker Nginx container image as the base for the container. After that, we map port 80 in the container to port 8080 on our host machine. This way, when we're finished, we'll be able to access our application by navigating to http://localhost:8080.

It next copies ./docker/nginx/default.conf from the local filesystem to /etc/nginx/conf.d/default.conf in the container's filesystem. default.conf provides the core configuration for Nginx. To save space, I've not included it here. However, you can find it in the repository for this tutorial.

Finally, the container gets access to a filesystem volume in the PHP container, which we'll see next. This will let us develop locally on our host machine, yet use the code in the Nginx server.

The PHP Container

The configuration for the PHP container, below, is rather similar to that of the Nginx container.

```
php:
  build: ./docker/php/
  expose:
    - 9000
  volumes:
    - .:/var/www/html
```

You can see that it starts off by naming the container `php`, which sets the container's hostname. The `build` directive tells it to use a configuration file, called `Dockerfile`, located in `./docker/php` which contains the following instructions:

```
FROM php:7.0-fpm

RUN docker-php-ext-install pdo_mysql \
    && docker-php-ext-install json
```

This states that our container is based on the official PHP 7 image from Docker Hub, which uses PHP-FPM. I'm keeping things as official as possible.

In addition to using the default image, I've also added some PHP extensions, by calling the `docker-php-ext-install` command. Specifically, I'm ensuring that `pdo_mysql` and `json` are available in the container.

This command does not install an extension's dependencies. It only installs the extension, if the dependencies are available.

Going back to docker-compose.yml, it next exposes the container's port 9000. If this is your first time reading about Docker, that might not make a lot of sense. What it's doing is exposing the container's port 9000, a lot like when we allow access to a port through a firewall.

If you've had a look at `./docker/nginx/default.conf` in the source repository, you'll have see that it contains the directive: `fastcgi_pass php:9000;`. This allows the Nginx container to pass off requests to PHP in the PHP container.

Lastly, we're mapping a directory on our development machine to a directory in the container, for use in the container. This will also be available in the Nginx container, thanks to the `volumes_from` directive, which we saw earlier.

This has the effect of sharing your local directory with the container, rather like Vagrant's shared folders, which makes local development quite efficient. When you make a change in your development environment, whether in a text editor, or an IDE such as PhpStorm, the changes will be available in the container as well. There is no need to manually copy or sync files between your development environment and the container.

Thanks to Tom Lobato and Rutvik Prajapati for pointing out that I wasn't as clear here as I should have been.

The MySQL Server

Now, for the final piece, the MySQL container.

```yaml
mysql:
  image: mysql:latest
  expose:
    - 3306
  volumes:
    - database_data:/var/lib/mysql
  environment:
    MYSQL_ROOT_PASSWORD: secret
    MYSQL_DATABASE: project
    MYSQL_USER: project
    MYSQL_PASSWORD: project
```

As with the other containers, we've given it a name (and hostname): `mysql`. We are using the official MySQL container image, from DockerHub as the foundation for it and exposing port 3306, the standard MySQL port, which was referred to in the PHP container.

Next, using the `volumes` directive, we're making any changes in `/var/lib/mysql`, where MySQL will store its data files, permanent. We then finish up setting four environment variables, which the MySQL server needs. These are for the root MySQL password, the name of the database to create, and an application username and password.

Booting the Docker Containers

Now that we've configured the containers let's make use of them. From the terminal, in the root directory of your project, run the following command:

```
docker-compose up -d
```

What this will do, is to look for `docker-compose.yml` in the same directory for the instructions it needs to build the containers, and then start them. After they start, Docker will go into daemon mode.

When you run this, you'll see each container being created and started. If this is the first time that you've created and launched the containers, then the base images will have to be first downloaded, before the containers can be created on top of them.

This may take a few minutes, based on the speed of your connection. However, after the first time, they'll usually be booted in under a minute.

With them created, you’re ready to use them. At this point, in a browser, navigate to localhost:8080, where you’ll see your application running, which renders the standard Zend Expressive Skeleton Project home page.

Conclusion

That’s how to use Docker to build a local development environment for Zend Expressive (or any PHP) application. We have one container which runs PHP, one which runs Nginx, and one which runs MySQL; all able to talk to each other as needed.

You could say that we can now build environments a lot like we can build code—in a modular fashion. It’s a fair way of thinking about it. Why shouldn’t we be able to do so?

I appreciate this has been quite a rapid run-through. But it has covered the basics required to get you started. We haven’t looked too deeply into how Docker works, nor gone too far beyond the basics.

How to Debug a Docker Compose Build

If you're using docker-compose to build a Docker container setup and something's not working, here's a basic process you can follow to find out what happened, and get your containers up and running properly.

To put this into greater context, recently I was trying to create a Docker development environment for building a new Mezzio application. I'd created a basic container configuration, using the `php:7-apache` container and MariaDB and thought that everything should be working as expected.

However, when I attempted to run the installer, the database hostname wasn't able to be resolved, so the installed failed. This seemed rather strange, as I didn't see anything strange in the console output (or so I thought) after I started the containers.

The first thing I thought to do was to access bash on the webserver container to try and ping the database to see if it could be accessed. Sure enough, I wasn't able to.

Review the Docker Compose Configuration

I thought that I must have misconfigured something. But on reviewing the configuration with a known working setup, nothing seemed out of the ordinary. However, the database server clearly wasn't working, for some reason.

Check the State of Each Container

So I decided to have a quick look at its state, by running `docker-compose ps`. On doing so, I saw the following output; note `Exit 1` as the state for the database container.

While not that clear, nor insightful, it means that the container wasn't able to start successfully. As the webserver's state is set to `Up`, it is running properly.

```
      Name                    Command              State        Ports
-----------------------------------------------------------------------------------
core1000_database_1    docker-entrypoint.sh mysqld     Exit 1
core1000_webserver_1   docker-php-entrypoint apac ...  Up      0.0.0.0:80->80/tcp
```

To get the status of a particular container, add its name to the end of the command. For example: `docker-compose ps database`.

What Do the Logs Say?

Given that the database container had exited prematurely, it was time to get more detailed information about the problem that caused this to happen. To do that, I needed to find out what the logs contained.

The reason why is that this is not so different from any other form of debugging. If you want to know what's going on, the best place to look is the logs. They contain information such as ports already being in use, server misconfiguration, and so on.

To have a look at the logs for the database container, I used the command:

```
docker-compose logs --follow database
```

You don't have to include the --follow flag. But I decided to, as I wanted to reload the page a few times and see the output scroll by, instead of having to call the command repeatedly.

On doing so, here's what I saw:

```
database_1   | error: database is uninitialized and password option is not specified
database_1   |   You need to specify one of MYSQL_ROOT_PASSWORD,
MYSQL_ALLOW_EMPTY_PASSWORD and MYSQL_RANDOM_ROOT_PASSWORD
```

Fix the Problem, Rebuild the Containers, and Check Again

Turns out, I'd almost configured the containers correctly, but had forgot to configure a mandatory setting for the MariaDb container. Given that, I added a value for MYSQL_ROOT_PASSWORD, saved the file and ran docker-compose up -d --build database to rebuild and restart the database container.

This time, I paid more careful attention to the console output and definitely saw nothing out of the obvious. However, just to be sure, I ran docker-compose ps again, to ensure that both containers were up and running.

On doing so, sure enough, everything was working as expected. Given that, I reloaded the containers and, after a minute or so, had a working installation ready to use. Achievement unlocked!

```
      Name                  Command               State        Ports
-------------------------------------------------------------------------------------
core1000_database_1    docker-entrypoint.sh mysqld     Up      0.0.0.0:3306-
>3306/tcp
core1000_webserver_1   docker-php-entrypoint apac ...   Up      0.0.0.0:80->80/tcp
```

In Conclusion

While this hasn't been the most in-depth of guides on how to debug a Docker configuration, built with Docker Compose, it's covered the essentials.

It's covered how to determine if a container isn't working and how to find out more detailed information so that you can correct problems.

To summarise, follow these steps to debug a Docker Compose-based container configuration:

```
Use `docker-compose ps` to see the state of all the containers
Use `docker-compose logs --follow` to inspect the logs to find out what errors are
occurring
Fix the problem
Rebuild and restart the containers with `docker-compose up -d --build`
Lather, rinse, repeat
```

If you need any further information, check out the official Docker Compose documentation.

How to Build a Docker Test Environment

Docker makes it easy to build local development environments. But, what about being able to build test environments and run acceptance, unit, functional, and other types of tests? In this chapter, I'll show you how to implement testing using PHPUnit and Codeception in the configuration which we've built.

In the first part in this series on developing web applications using Docker, we saw how to create a local development environment using Docker; one ideally suited to creating a Mezzio application (or any other kind of PHP-based web application). But, what we didn't cover was how to handle testing in a Docker-based environment.

At first glance, this might not seem like all that much of a problem. However, the challenge I found, at least when I was first getting up to speed with Docker, was *"how do I run the tests"*?

If I was using a Vagrant-based virtual machine, then I'd run `vagrant ssh` and run the tests as I normally would; whether by calling `phpunit` or `codecept`. Alternatively, I could execute the tests remotely in the virtual machine, instead of ssh'ing in first.

But how do you run tests when working with Docker containers?

After a bit of searching, I found that it's not that difficult. But you have to use the right combination of commands. At first I thought that `docker run` was how you ran them. Unfortunately, that booted up another instance of the container I'd specified, in a separate environment (and network), without any of the other containers being available. As a result the tests failed.

Instead, what you need to do is to use `docker exec`. This command connects to an existing, *running*, container, one which is in a network along with the other containers which we need. As a result it will be able to access the containers and execute the tests successfully.

Adding Test Support

We didn't, explicitly, add test support in part one of this series, but most projects constructed with the Zend Skeleton installer() will have some basic tests in place. If you look in `test/AppTest/Action` you'll see `HomePageActionTest.php`.

This test class performs unit tests on the `HomePageAction` class. To run it, we could call `vendor/bin/PHPUnit --configuration test/Unit/PHPUnit.xml`. Given it's quite elementary in what it's assessing, running it could be done locally, without needing the containers. So, what about acceptance tests instead? These tests need all the containers in our setup.

PHPUnit doesn't have support for acceptance tests. For that we're going to need a tool such as Codeception, which has built-in support for acceptance tests. To make it available, run the following commands to both install it as a dependency and to create its core configuration files.

```
// Add it as a dependency
composer require codeception/codeception

// Create the core configuration files
vendor/bin/codecept bootstrap
```

With the files created, in tests/acceptance.suite.yml, set the value of url: to nginx. This sets the base URI to use when running acceptance tests against our application. Now we have one thing left to do, which is to actually create an acceptance test. Let’s use Codeception to create a skeleton file for us, by running the command:

```
vendor/bin/codecept generate:cest acceptance HomePageTest
```

After this, open up the file tests/acceptance/HomePageCest.php, which will look a lot like the code below:

```php
<?php

class HomePageCest
{
    public function _before(AcceptanceTester $I)
    {
    }

    public function _after(AcceptanceTester $I)
    {
    }

    // tests
    public function tryToTest(AcceptanceTester $I)
    {
    }
}
```

With the file created, we’ll replace the three existing functions with the following one, which will perform elementary tests on the home page.

```php
public function tryToTest(AcceptanceTester $I)
{
  $I->am('Guest User');
  $I->expectTo('Be able to view all journal records listed in reverse date order');
  $I->amOnPage('/');
  $I->seeResponseCodeIs(200);
  $I->see('Welcome to zend-expressive', '//h1');
  $I->seeLink('Middleware', 'https://github.com/zendframework/zend-stratigility');
}
```

This test will do the following:

- Attempt to connect to the test site on the default route, /
- Check that the response code is an HTTP 200 OK
- Check that ‘Welcome to zend-expressive' is set as the contents of the page’s h1 tag
- Check that a link exists with the text of 'Middleware' and a href of 'https://github.com/laminas/laminas-stratigility'.

If all of these assertions pass, then the test will have succeeded.

Running the Tests

Now that the test structure is in place, let’s run it. Gladly, it’s not that different from running tests in either a virtual machine, or locally. To do so, you need run the following command:

```
docker exec -it healthmonitor_php_1 php vendor/bin/codecept run acceptance
```

That’s a little long-winded. So I’ll explain what it does. Using `exec` Docker will execute the command `php vendor/bin/codecept run acceptance` in the container called `healthmonitor_php_1`. If you’re not familiar with the exec command, as the documentation says, it will:

> Run a command in a running container

It’s quite similar to running a command on a remote server with ssh or vagrant ssh using the `-c` or `--command` switches. Don’t make the mistake of using the `run` command, which will run a command in a new container. This will boot up a new instance of the PHP container, on a separate network, without any of the other containers which are needed to perform the tests.

Adding Tooling Support

And that’s how to run tests for PHP-based web applications when you’re developing and running them using a Docker local development environment. But it’s reasonable to expect that you might not remember the command, or make a mistake. So let’s quickly look at a few quick ways to automate the process.

Using Make

The first suggestion I have is to use Make. This is a technique I picked up while working with an excellent group of developers at Refinery29. Create a new file, called Makefile, in the root directory of your project.

In there, add the following:

```
all: test

.PHONY: test unit integration

test: unit functional acceptance

PHPUnit:
    docker exec -it healthmonitor_php_1 php vendor/bin/PHPUnit

unit:
    docker exec -it healthmonitor_php_1 php vendor/bin/codecept run unit

acceptance:
    docker exec -it healthmonitor_php_1 php vendor/bin/codecept run acceptance

functional:
    docker exec -it healthmonitor_php_1 php vendor/bin/codecept run functional
```

What we've done is to create a series of targets, similar to what you do in other tools, such as Phing. The first two, `all` and `.PHONY` setup the default target to run, if we don't request one specifically. Hopefully, the final five should be fairly self-explanatory. But if not, here's how they work, using the PHPUnit command as an example.

```
PHPUnit:
    docker exec -it healthmonitor_php_1 php vendor/bin/PHPUnit
```

The first line is the name of the target. The second line specifies the command to run when the target is called. We can also group commands together, such as in `test: unit functional acceptance`. Here, what we're doing is to create a command called `test` which will run the `unit`, `functional`, and `acceptance` tests.

To run any of them, in the terminal in the root directory of your project, we call `make` along with the target's name. For example, if we wanted to run the unit target, we could then call `make unit`. However, if we wanted to run all the tests, we could call `make` or `make test`.

Using Phing

Now what about something more recent, more PHP-specific? What about Phing? If that's something that you're more comfortable with, then here's a configuration file which will provide sufficient information to get the PHPUnit and Codeception acceptance tests running.

```xml
<?xml version="1.0" encoding="UTF-8"?>

<project name="Health Monitor" default="test">
  <target name="PHPUnit"
        description="Run unit tests using PHPUnit in the Docker container">
    <echo msg="Running PHPUnit tests" />

    <exec command="docker exec -it healthmonitor_php_1 php vendor/bin/PHPUnit"
        logoutput="/dev/stdout"
        checkreturn="true" />
  </target>

  <target name="test" depends="PHPUnit">
    <echo msg="Running acceptance tests using Codeception" />

    <exec command="docker exec -it healthmonitor_php_1 php vendor/bin/codecept
run acceptance"
        logoutput="/dev/stdout"
        checkreturn="true" />
  </target>
</project>
```

Here, you can see that we have a Phing XML file, called build.xml. In it, we've provided a project name and a default target to run, test. Then, we've defined two targets.

We define each target in the target XML element, where it requires a name, and can take an optional description; it's optional, but quite handy when attempting to quickly ascertain what a target does.

Each target makes use of the echo and exec tasks. Echo prints out the string specified in msg. Exec, as you'd likely expect, runs a command, which we define in command and has the option of directing output to either stdout or to another location, as we have here by specifying /dev/stdout as the value of logoutput.

With the file created, we can run it from the command line by using the command vendor/bin/phing which will run all the targets, as test depends on PHPUnit. Alternatively, we can run a target by it's name, by providing the name of the target, such as vendor/bin/phing PHPUnit.

The coverage of Make and Phing were deliberately simplistic, as the intent was to focus on running the test commands. There will be thorough guides on Make and Phing in upcoming tutorials.

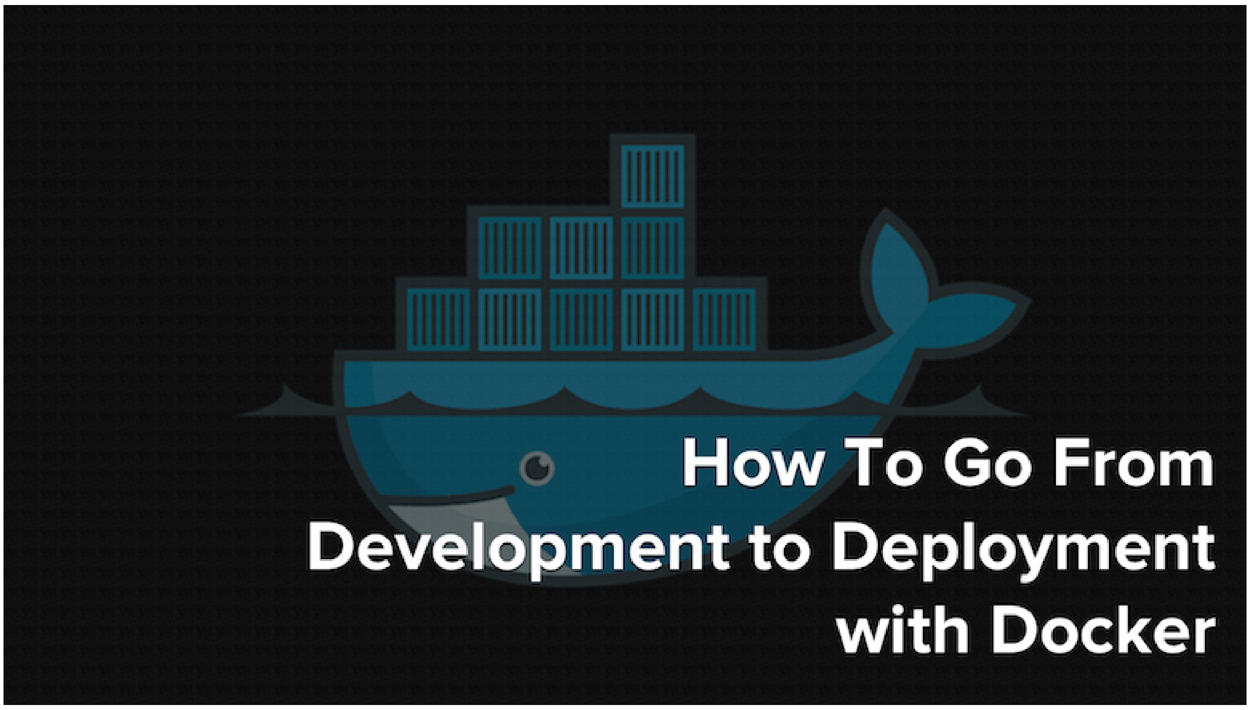
In Conclusion

And that's how to build a test development environment using Docker. While there are many approaches to doing so, this one at least doesn't make things overly complicated.

By making only a slight addition to your local Docker development environment, you are now able to run all your tests, regardless of their type, as easily as you would if you were using a Vagrant-based virtual machine, or one of the MAMP, WAMP, or LAMP stacks.

How to Go From Development to Deployment with Docker

Want to know how to both containerise an application AND deploy it to a production environment? In this mammoth tutorial, I'll show you all the steps involved, and provide background information along the way, so you can build on what you'll learn.



Want to know how to both containerise an application **AND** deploy it to a production environment? In this mammoth tutorial, I'll show you all the steps involved, and provide background information along the way, so you can build on what you'll learn.

In my search to learn how to use Docker as a complete development solution, I've found a range of tutorials which discuss or walk through some part of the process.

Sadly, no one tutorial contains all the steps necessary to step you through containerizing an (existing) application through to deploying the said application in a production environment. Given that, my aim in writing this tutorial is to show you how to do this.

The challenge in doing so, unfortunately, is that there's a lot to learn and absorb. Here's why:

1. So that you know the steps involved.
2. So that those steps *make sense*.
3. Perhaps most importantly—so that you can *continue to educate yourself* about the tools, different tool combinations, tool options, configuration settings, and so on.

So this is going to be a lengthy post. However, I've aimed to provide the most direct path to your first production deployment, as well as to structure it so that it's easy to work through or navigate to the specific part you need.

Tutorial Prerequisites

To follow along with this tutorial, you're going to need the following, three, things:

- A DigitalOcean account

- A Docker Hub account
- Docker installed on your development machine
- An existing project that you want to deploy using Docker

Make sure you have them before you go any further.

What You Need To Do (Or the tl;dr version)

If you're looking for the quick version of the article, this is it. At its core to containerise and deploy an application, there are only four steps involved. These are:

1. Create & Build the Container
2. Store the Image in an Accessible Registry
3. Build a Deployment Configuration
4. Make the Deployment

With that said, here is some greater context and understanding of the steps. As I've come to understand it, using Docker, in essence, comes down to two essential parts:

1. Create a build image
2. Use that image in a deployed configuration

This way, in contrast to what I suggested in the post on creating a development environment using Docker, an image can be used across multiple projects, and not be tied to only one.

This may be painfully obvious to some, but at first it wasn't to me. For those who thought the same as I did, I hope this helps. It's rather like writing maintainable code if you think about it.

{{< partial "inline-book-promotion" >}}

Create & Build the Container

OK, let's get started with part one: creating and building the container, specifically the creating part.

Create the Container

In the root directory of your project, create a new file called `Dockerfile`, where we'll store the instructions that Docker will use to build our container. In there, add the following code. Then we'll step through what it does.

```
FROM php:7.0-apache
WORKDIR /var/www/html
COPY ./ /var/www/html/
COPY ./docker/default.conf /etc/apache2/sites-enabled/000-default.conf
EXPOSE 80

RUN docker-php-ext-install pdo_mysql \
    && docker-php-ext-install json
```

We start with the FROM statement. This statement specifies the container on which *our* container is built. I've chosen php:7.0-apache. This is so that the PHP run-time and web server are combined into one container. Doing so avoids the need for inter-container communication, along with any unnecessary configuration effort on our part.

Then, we use WORKDIR to set the working directory, or the base path within the container, where other commands operate relative to if relative paths are used.

The two COPY commands are good examples. These commands copy:

- The contents of the current working directory into the container's /var/www/html directory.
- A custom Apache configuration file in place of the container's existing one. The reason for doing so is that the container's default Apache configuration uses /var/www/html as the document root. However, the example code I'm working with needs it to be /var/www/html/public. Set any other directives that you feel the need to use.

For the sakes of complete transparency, here's the configuration that I used. It's merely a copy of the configuration inside the container with the DocumentRoot directive's setting changed.

```
<VirtualHost *:80>
    ServerAdmin webmaster@localhost
    DocumentRoot /var/www/html/public
    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

Next, we set the EXPOSE command, which exposes port 80 in the container. This is done so that the container can be communicated with, no matter which host its contained in (which may be a local development machine or a remote host, such as on DigitalOcean).

Finally, we use the RUN command to install two PHP extensions; pdo_mysql and JSON. We could add any number of other extensions, install any number of packages, and so on. However, for this example, that's all we need.

For the container, we're now done. It has all that it needs to support the application which we're going to place inside of it.

Build the Container

We now need to build it. To do that, we use the docker build command, supplying two arguments to it. These are:

1. The name of the container
2. The contents (or context) of the container

Here's the command that I'll use to build the container:

```
docker build -t basicapp .
```

This gives the container the name basicapp (which is important, as we'll need that in the later sections, and specifies that the local directory is the content for the container. When you run the command, you'll see output similar to the following:

```
Sending build context to Docker daemon  34.3 kB
Step 1/6 : FROM php:7.0-apache
 ---> 23f9c84560a6
Step 2/6 : WORKDIR /var/www/html
 ---> Using cache
 ---> 6fd5d5375996
Step 3/6 : COPY ./ /var/www/html/
 ---> 3f4313a5bb2d
Removing intermediate container cc38a34f844b
Step 4/6 : COPY ./docker/default.conf /etc/apache2/sites-enabled/000-default.conf
 ---> ad8ba9e7bf7f
Removing intermediate container ac39c49311ad
Step 5/6 : EXPOSE 80
 ---> Running in 4c71b935da37
 ---> eb836808c859
Removing intermediate container 4c71b935da37
Step 6/6 : RUN docker-php-ext-install pdo_mysql && docker-php-ext-install json
 ---> Running in 25ffa117cf19
+ cd pdo_mysql
+ phpize
```

There, you can see that it's running through all the commands in Dockerfile, creating our container, which is, in effect, a customised version of the base container: php:7.0-apache. All being well, the last piece of output that you'll see is something similar to:

```
Successfully built 51cc061b52d8
```

We can doubly confirm that the container's ready, by now running the command docker images basicapp. This should result in output similar to the following:

```
REPOSITORY        TAG               IMAGE ID           CREATED            SIZE
basicapp         latest          51cc061b52d8       3 minutes ago       390 MB
```

Note that the size of the container is quite large. I could have chosen to use a smaller base container, such as one based on Alpine Linux. I've deliberately not because the container I've chosen works well for a tutorial.

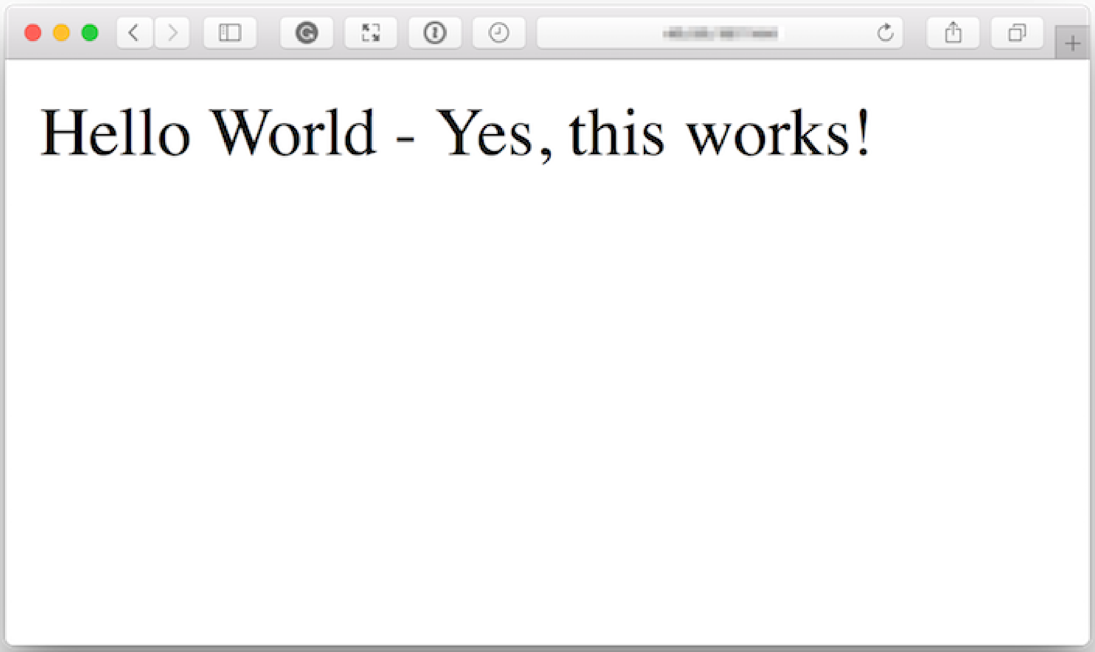
Now that the image is successfully built, we need to test that it works, just like we'd test our code. We can do this by running it. We don't need a complicated setup to do that, just a container and Docker, both of which we have.

To do so, run the following command

```
docker run -p 2000:80 basicapp
```

This starts the container, mapping the port 80 in the container to port 2000 on our host, which is our local machine. As the container's not too sophisticated, it should boot quite quickly.

When the console output's stopped scrolling, open your browser to http://localhost:2000, and behold the majesty, *the grandeur*, **the sheer brilliance** that is the output of our app.



Hello World - Yes, this works!

OK, it's a text string. But it works. Given that, use *CTRL+C* to end the process, as we no longer need to run it locally.

Store the Image in an Accessible Registry

It's now time to store the image so that any deployment configuration can use it. To do that we have to store it in a container registry. This is where the Docker Hub account listed in the article’s prerequisites comes in.

To do so, we have first to log in, so that we’re authenticated to use the account. We do that by running `docker login`, providing our Docker Hub username and password when prompted. After successfully logging in, we need to do two things:

1. Tag our new container (which is similar to how you’d tag a release)
2. Push it to our Docker Hub account

Tag an Image

To tag the image, run the following commands:

```
docker tag basicapp settermjd/basicapp:0.0.1
```

Reading through the command from left to right, we pass:

1. The name of the image to tag
2. Our Docker Hub username and the name that we’ll store our image under
3. A tag name

I strongly encourage you to follow semantic versioning when choosing tag names—unless you want to cause pain and heartache for yourself later.

So I’m storing my `basicapp` image, in my account, as `basicapp`, and giving it the tag `0.0.1`. Nothing spectacular, but it’s clean and tidy. It’s also clear that this is the very first version of my container.

Push the Image to Docker Hub

With that done, we now need to push the image. As you *might* expect, we’ll use the `docker push` command to do that. This time, as you can see in the command below, we pass the `<account>/<imagename>:<tagname>` combination to docker push.

```
docker push settermjd/basicapp:0.0.1
```

This will store the image in our account under the name `basicapp` with the tag `0.0.1`. If you want to be sure, login to your account and see that it’s now listed there as a public container in your repository.

Build a Deployment Configuration

Believe it or not, we’re almost done! Now we need to build a deployment configuration so that we can deploy our container. To do that, we’ll create a `docker-compose.yml` file, as you can see below.

```
version: '3'
services:
  web:
    image: settermjd/basicapp:0.0.1
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
    ports:
     —"80:80"
```

If you’re not familiar with the format, here’s what’s happening.

It’s using version 3 of the docker-compose file format and lists one service (or container) in the configuration, called "web". This is also the internal hostname of the container; something we don’t need to think about again in the tutorial.

To the `image:` element, we supply the name of the container which we supplied to `docker push` previously. Here, we are stipulating the image that the service will use, and it’s version. Appreciate the flexibility that this statement represents and how using an image, instead of a direct configuration as we did in the earlier tutorial, gives us many options.

In the `deploy:` element, we specify the deployment options. We’re requesting five replicas of our container to be created in the deployment, which will be transparently used in a round-robin fashion. Then, we’re imposing resource limits on the containers, setting them to use no more than 1 CPU and to have a maximum memory of 50MB.

These limits are somewhat arbitrary, purely there for educational purposes. Make sure you check out the resource limits documentation for more information on what’s available.

Finally—and one of the most critical lines in the configuration, without which the application won’t be accessible—is the `ports:` element. This binds port 80 on the container, to port 80 on the host.

As containers work within a host, when we deploy them, if we don’t do this, they won’t be accessible from the outside world. So this ensures that requests to port 80 to the IP of the host is passed on to port 80 of the container.

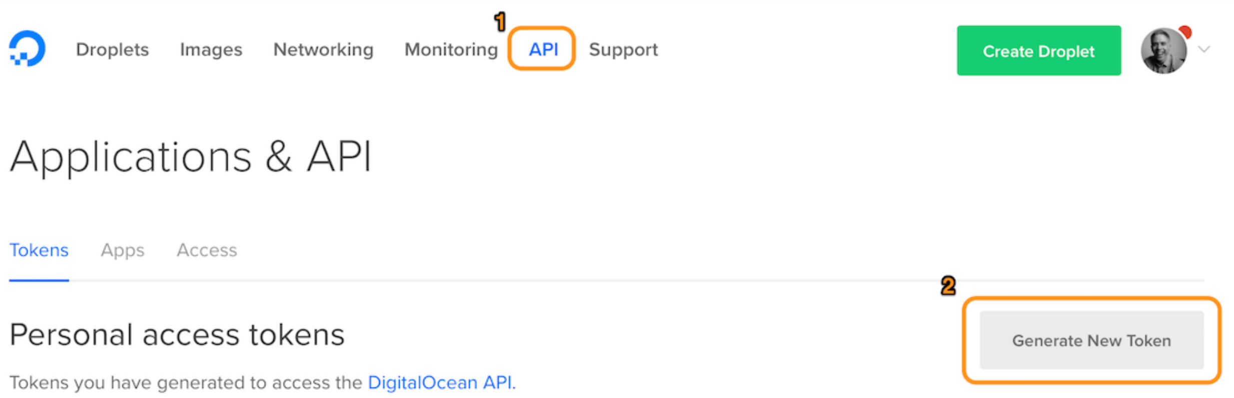
Make the Deployment

All right, the last stage!

Here, we need to do two things:

1. Create the host into which we’ll put our container configuration
2. Deploy the configuration and check that it works

To do this, you’re going to need an API token from your DigitalOcean account. To get this, after logging in to the DigitalOcean dashboard, click on "**API**" (1), and click "**Generate New Token**" (2), as you can see in the image below.



For the sake of simplicity, copy the token and store it as an environment variable in your shell, by running:

```
export DO_TOKEN=<your generated token>
```

With that done, you're ready to create your remote host. For this, we'll need the `docker-machine` command.

Docker-machine creates and manages machines running Docker, in this case, a DigitalOcean droplet. It's not going to be anything fancy, just a standard droplet with 1GB of memory. To create it, run the command below.

```
docker-machine create \
  --driver=digitalocean \
  --digitalocean-access-token=$DO_TOKEN \
  --digitalocean-size=1gb \
  basicapp
```

Here, we're using the DigitalOcean driver, specifying the API token to authenticate against our account, and specifying the disk size, along with a name for the droplet. We could also specify a number of other options, such as *region*, *whether to enable backups*, *the image to use*, and whether to *enable private networking*.

It will take a little while to complete, and you should see output similar to the following, but it shouldn't be more than a few minutes.

```
Running pre-create checks...
Creating machine...
(basicapp) Creating SSH key...
(basicapp) Creating Digital Ocean droplet...
(basicapp) Waiting for IP address to be assigned to the Droplet...
Waiting for machine to be running, this may take a few minutes...
Detecting operating system of created instance...
Waiting for SSH to be available...
Detecting the provisioner...
Provisioning with ubuntu(systemd)...
Installing Docker...
Copying certs to the local machine directory...
Copying certs to the remote machine...
Setting Docker configuration on the remote daemon...
Checking connection to Docker...
Docker is up and running!
To see how to connect your Docker Client to the Docker Engine running on this
virtual machine, run: docker-machine env basicapp
```

When it’s finished, we then have to ensure that any commands we run from now on are run on the remote host, not on our local development machine. To do that, we set several environment variables (four to be specific). These are:

- `DOCKER_TLS_VERIFY`
- `DOCKER_HOST`
- `DOCKER_CERT_PATH`
- `DOCKER_MACHINE_NAME`

We could do all this by hand, but there’s no need to. The script to do that is provided in the last line of the droplet creation process' output, and should be:

```
docker-machine env basicapp
```

Use the `eval` command, as in the sample below, to run it and update your environment settings.

```
eval $(docker-machine env basicapp)
```

With that done, we’re down to the last step: deploying to the remote host. To do that we need, *yet*, another Docker command. Yes, there are a lot of them if you’re thinking that.

The command is `docker swarm`. Docker swarm is Docker’s clustering functionality which, to quote the documentation:

> Turns a pool of Docker hosts into a single, virtual Docker host

However, we only have one host you may be thinking. And right you are. However, if you want to build your deployment into a cluster later, it helps to know about this command. It’s a little outside the scope of this tutorial to discuss it in-depth. So make sure you check out the docs for further information.

To get the swarm ready, we first have to initialise it. We do that by running the command below.

```
docker swarm init --advertise-addr <droplet IP address>
```

You can see that I've passed an IP address to the `--advertise-addr` switch. This was necessary because the droplet exposed two IP addresses, and swarm wasn't sure which one to use.

Now that the swarm is ready, it's time to add a host to it. To do that, we call another command, which you can see below.

```
docker stack deploy --compose-file docker-compose.yml basicapp
```

Docker stack manages Docker stacks. A stack is:

> A collection of services that make up an application in a specific environment.

Are you confused by all the terms yet?

So, to recap just briefly, the swarm is the collection of hosts that will run our application. The stack is the application, made up of a collection of services, that make up our application. There's method in the madness; it just takes a little while to get your head around it.

This command will take a little while to complete building the container on the remote host. It will ensure that there are five containers and that each one has access to no more than 1 CPU and 50MB of memory. You can watch it building if you periodically run `docker stack services basicapp`. This lists the services in the stack.

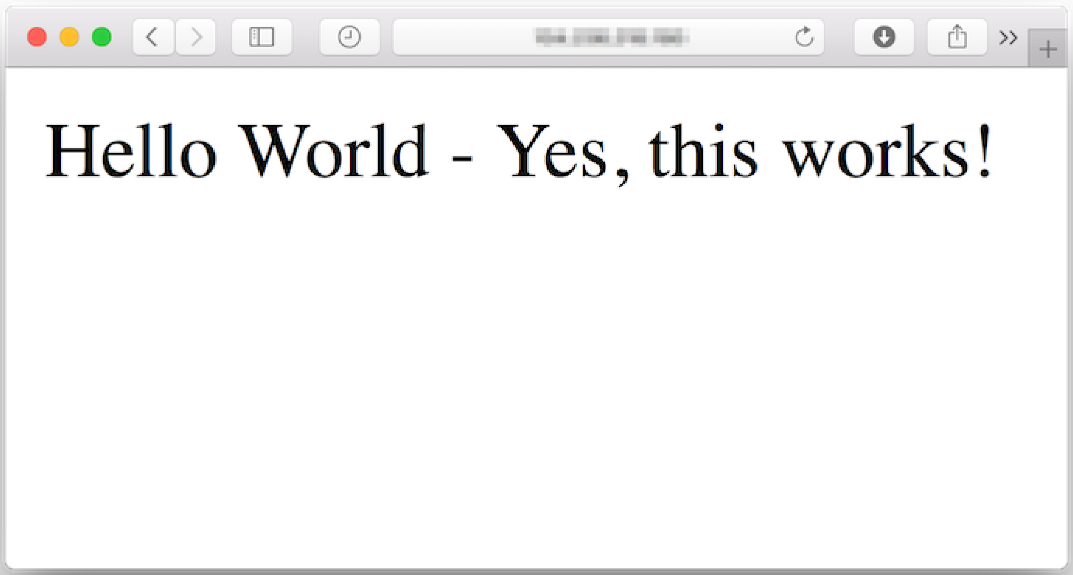
Here's an example output from when I built mine:

```
ID           NAME          MODE        REPLICAS  IMAGE
nvprlz81p2ne  basicapp_web  replicated  3/5       settermjd/basicapp:0.0.1
```

You can see that there's one service, "`basicapp_web`", based on the image that we created earlier, and it has three of the five replicas that we specified ready to go. The name is the service name from the `docker-compose.yml` file, prefixed with the stack name and an underscore.

When it's done, we'll then be able to access our deployed application! If you've not assigned a CNAME record to your new droplet, then grab it's IP address from the Droplets list, and navigate to that IP in your browser of choice.

And here's what mine looks like:


Hello World - Yes, this works!

In Conclusion

And that's the end of the tutorial. We've covered how to containerise an application, how to build a deployment configuration using Docker Swarm, and deploy it to a non-development environment using Docker Stack. Yes, there have been quite a number of steps, and perhaps too many Docker commands—*my pet peeve with Docker*.

But, we're there!

I've taken some shortcuts to keep the post as short as possible. And there are so many things that I've not covered, such as:

- Creating a more sophisticated image or deployment configuration
- Considered the security implications of the container we've deployed
- Considered such requirements such as how to roll back a release
- Seen how to update an existing release
- Seen how to destroy an existing swarm

However, for a simple example, it's sufficient. Ideally, I'd like to expand on this post at some stage. However, I didn't want to overwhelm you today.

I hope that you've been able to follow the instructions here successfully, and in the process learned a lot. If you've had any problems, want to know more, or want to suggest other ways to do it, add your feedback in the comments.