How To Set Up Laravel, Nginx, and MySQL with Docker Compose

*The author selected*[*The FreeBSD Foundation*](https://www.brightfunds.org/organizations/the-freebsd-foundation)*to receive a donation as part of the*[*Write for DOnations*](https://do.co/w4do-cta)*program.*

Introduction

Over the past few years, [Docker](https://docs.docker.com/) has become a frequently used solution for deploying applications thanks to how it simplifies running and deploying applications in ephemeral [containers](https://www.docker.com/resources/what-container). When using a LEMP application stack, for example, with [PHP](https://php.net/docs.php), [Nginx](https://nginx.org/en/), [MySQL](https://dev.mysql.com/doc/) and the [Laravel](https://laravel.com/docs/5.6) framework, Docker can significantly streamline the setup process.

[Docker Compose](https://docs.docker.com/compose/) has further simplified the development process by allowing developers to define their infrastructure, including application services, networks, and volumes, in a single file. Docker Compose offers an efficient alternative to running multiple docker container create and docker container run commands.

In this tutorial, you will build a web application using the Laravel framework, with Nginx as the web server and MySQL as the database, all inside Docker containers. You will define the entire stack configuration in a docker-compose file, along with configuration files for PHP, MySQL, and Nginx.

Prerequisites

Before you start, you will need:

* One Ubuntu 18.04 server, and a non-root user with sudo privileges. Follow the [Initial Server Setup with Ubuntu 18.04](https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-18-04) tutorial to set this up.
* Docker installed, following Steps 1 and 2 of [How To Install and Use Docker on Ubuntu 18.04](https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-on-ubuntu-18-04).
* Docker Compose installed, following Step 1 of [How To Install Docker Compose on Ubuntu 18.04](https://www.digitalocean.com/community/tutorials/how-to-install-docker-compose-on-ubuntu-18-04).

Step 1 — Downloading Laravel and Installing Dependencies

As a first step, we will get the latest version of Laravel and install the dependencies for the project, including [Composer](https://github.com/composer/docker), the application-level package manager for PHP. We will install these dependencies with Docker to avoid having to install Composer globally.

First, check that you are in your home directory and clone the latest Laravel release to a directory called laravel-app:

* cd ~
* git clone https://github.com/laravel/laravel.git laravel-app

Move into the laravel-app directory:

* cd ~/laravel-app

Next, use Docker's [composer image](https://hub.docker.com/r/library/composer/) to mount the directories that you will need for your Laravel project and avoid the overhead of installing Composer globally:

* docker run --rm -v $(pwd):/app composer install

Using the -v and --rm flags with docker run creates an ephemeral container that will be bind-mounted to your current directory before being removed. This will copy the contents of your ~/laravel-app directory to the container and also ensure that the vendor folder Composer creates inside the container is copied to your current directory.

As a final step, set permissions on the project directory so that it is owned by your non-root user:

* sudo chown -R $USER:$USER ~/laravel-app

This will be important when you write the Dockerfile for your application image in Step 4, as it will allow you to work with your application code and run processes in your container as a non-rootuser.

With your application code in place, you can move on to defining your services with Docker Compose.

Step 2 — Creating the Docker Compose File

Building your applications with Docker Compose simplifies the process of setting up and versioning your infrastructure. To set up our Laravel application, we will write a docker-compose file that defines our web server, database, and application services.

Open the file:

* nano ~/laravel-app/docker-compose.yml

In the docker-compose file, you will define three services: app, webserver, and db. Add the following code to the file, being sure to replace the root password for MYSQL\_ROOT\_PASSWORD, defined as an [environment variable](https://docs.docker.com/compose/compose-file/#environment) under the db service, with a strong password of your choice:

~/laravel-app/docker-compose.yml

version: '3'

services:

#PHP Service

app:

build:

context: .

dockerfile: Dockerfile

image: digitalocean.com/php

container\_name: app

restart: unless-stopped

tty: true

environment:

SERVICE\_NAME: app

SERVICE\_TAGS: dev

working\_dir: /var/www

networks:

- app-network

#Nginx Service

webserver:

image: nginx:alpine

container\_name: webserver

restart: unless-stopped

tty: true

ports:

- "80:80"

- "443:443"

networks:

- app-network

#MySQL Service

db:

image: mysql:5.7.22

container\_name: db

restart: unless-stopped

tty: true

ports:

- "3306:3306"

environment:

MYSQL\_DATABASE: laravel

MYSQL\_ROOT\_PASSWORD: your\_mysql\_root\_password

SERVICE\_TAGS: dev

SERVICE\_NAME: mysql

networks:

- app-network

#Docker Networks

networks:

app-network:

driver: bridge

The services defined here include:

* app: This service definition contains the Laravel application and runs a custom Docker image, digitalocean.com/php, that you will define in Step 4. It also sets the working\_dir in the container to /var/www.
* webserver: This service definition pulls the [nginx:alpine image](https://hub.docker.com/_/nginx/) from Docker and exposes ports 80 and 443.
* db: This service definition pulls the [mysql:5.7.22 image](https://hub.docker.com/_/mysql/) from Docker and defines a few environmental variables, including a database called laravel for your application and the root password for the database. You are free to name the database whatever you would like, and you should replace your\_mysql\_root\_password with your own strong password. This service definition also maps port 3306 on the host to port 3306 on the container.

Each container\_name property defines a name for the container, which corresponds to the name of the service. If you don't define this property, Docker will assign a name to each container by combining a historically famous person's name and a random word separated by an underscore.

To facilitate communication between containers, the services are connected to a bridge network called app-network. A bridge network uses a software bridge that allows containers connected to the same bridge network to communicate with each other. The bridge driver automatically installs rules in the host machine so that containers on different bridge networks cannot communicate directly with each other. This creates a greater level of security for applications, ensuring that only related services can communicate with one another. It also means that you can define multiple networks and services connecting to related functions: front-end application services can use a frontend network, for example, and back-end services can use a backend network.

Let's look at how to add volumes and bind mounts to your service definitions to persist your application data.

Step 3 — Persisting Data

Docker has powerful and convenient features for persisting data. In our application, we will make use of [*volumes*](https://docs.docker.com/storage/volumes/) and [*bind mounts*](https://docs.docker.com/storage/bind-mounts/) for persisting the database, and application and configuration files. Volumes offer flexibility for backups and persistence beyond a container's lifecycle, while bind mounts facilitate code changes during development, making changes to your host files or directories immediately available in your containers. Our setup will make use of both.

Warning: By using bind mounts, you make it possible to change the host filesystem through processes running in a container, including creating, modifying, or deleting important system files or directories. This is a powerful ability with security implications, and could impact non-Docker processes on the host system. Use bind mounts with care.

In the docker-compose file, define a volume called dbdata under the db service definition to persist the MySQL database:

~/laravel-app/docker-compose.yml

...

#MySQL Service

db:

...

volumes:

- dbdata:/var/lib/mysql

networks:

- app-network

...

The named volume dbdata persists the contents of the /var/lib/mysql folder present inside the container. This allows you to stop and restart the db service without losing data.

At the bottom of the file, add the definition for the dbdata volume:

~/laravel-app/docker-compose.yml

...

#Volumes

volumes:

dbdata:

driver: local

With this definition in place, you will be able to use this volume across services.

Next, add a bind mount to the db service for the MySQL configuration files you will create in Step 7:

~/laravel-app/docker-compose.yml

...

#MySQL Service

db:

...

volumes:

- dbdata:/var/lib/mysql

- ./mysql/my.cnf:/etc/mysql/my.cnf

...

This bind mount binds ~/laravel-app/mysql/my.cnf to /etc/mysql/my.cnf in the container.

Next, add bind mounts to the webserver service. There will be two: one for your application code and another for the Nginx configuration definition that you will create in Step 6:

~/laravel-app/docker-compose.yml

#Nginx Service

webserver:

...

volumes:

- ./:/var/www

- ./nginx/conf.d/:/etc/nginx/conf.d/

networks:

- app-network

The first bind mount binds the application code in the ~/laravel-app directory to the /var/wwwdirectory inside the container. The configuration file that you will add to ~/laravel-app/nginx/conf.d/ will also be mounted to /etc/nginx/conf.d/ in the container, allowing you to add or modify the configuration directory's contents as needed.

Finally, add the following bind mounts to the app service for the application code and configuration files:

~/laravel-app/docker-compose.yml

#PHP Service

app:

...

volumes:

- ./:/var/www

- ./php/local.ini:/usr/local/etc/php/conf.d/local.ini

networks:

- app-network

The app service is bind-mounting the ~/laravel-app folder, which contains the application code, to the /var/www folder in the container. This will speed up the development process, since any changes made to your local application directory will be instantly reflected inside the container. You are also binding your PHP configuration file, ~/laravel-app/php/local.ini, to /usr/local/etc/php/conf.d/local.ini inside the container. You will create the local PHP configuration file in Step 5.

Your docker-compose file will now look like this:

~/laravel-app/docker-compose.yml

version: '3'

services:

#PHP Service

app:

build:

context: .

dockerfile: Dockerfile

image: digitalocean.com/php

container\_name: app

restart: unless-stopped

tty: true

environment:

SERVICE\_NAME: app

SERVICE\_TAGS: dev

working\_dir: /var/www

volumes:

- ./:/var/www

- ./php/local.ini:/usr/local/etc/php/conf.d/local.ini

networks:

- app-network

#Nginx Service

webserver:

image: nginx:alpine

container\_name: webserver

restart: unless-stopped

tty: true

ports:

- "80:80"

- "443:443"

volumes:

- ./:/var/www

- ./nginx/conf.d/:/etc/nginx/conf.d/

networks:

- app-network

#MySQL Service

db:

image: mysql:5.7.22

container\_name: db

restart: unless-stopped

tty: true

ports:

- "3306:3306"

environment:

MYSQL\_DATABASE: laravel

MYSQL\_ROOT\_PASSWORD: your\_mysql\_root\_password

SERVICE\_TAGS: dev

SERVICE\_NAME: mysql

volumes:

- dbdata:/var/lib/mysql/

- ./mysql/my.cnf:/etc/mysql/my.cnf

networks:

- app-network

#Docker Networks

networks:

app-network:

driver: bridge

#Volumes

volumes:

dbdata:

driver: local

Save the file and exit your editor when you are finished making changes.

With your docker-compose file written, you can now build the custom image for your application.

Step 4 — Creating the Dockerfile

Docker allows you to specify the environment inside of individual containers with a *Dockerfile*. A Dockerfile enables you to create custom images that you can use to install the software required by your application and configure settings based on your requirements. You can push the custom images you create to [Docker Hub](https://hub.docker.com/) or any private registry.

Our Dockerfile will be located in our ~/laravel-app directory. Create the file:

* nano ~/laravel-app/Dockerfile

This Dockerfile will set the base image and specify the necessary commands and instructions to build the Laravel application image. Add the following code to the file:

~/laravel-app/php/Dockerfile

FROM php:7.2-fpm

# Copy composer.lock and composer.json

COPY composer.lock composer.json /var/www/

# Set working directory

WORKDIR /var/www

# Install dependencies

RUN apt-get update && apt-get install -y \

build-essential \

mysql-client \

libpng-dev \

libjpeg62-turbo-dev \

libfreetype6-dev \

locales \

zip \

jpegoptim optipng pngquant gifsicle \

vim \

unzip \

git \

curl

# Clear cache

RUN apt-get clean && rm -rf /var/lib/apt/lists/\*

# Install extensions

RUN docker-php-ext-install pdo\_mysql mbstring zip exif pcntl

RUN docker-php-ext-configure gd --with-gd --with-freetype-dir=/usr/include/ --with-jpeg-dir=/usr/include/ --with-png-dir=/usr/include/

RUN docker-php-ext-install gd

# Install composer

RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer

# Add user for laravel application

RUN groupadd -g 1000 www

RUN useradd -u 1000 -ms /bin/bash -g www www

# Copy existing application directory contents

COPY . /var/www

# Copy existing application directory permissions

COPY --chown=www:www . /var/www

# Change current user to www

USER www

# Expose port 9000 and start php-fpm server

EXPOSE 9000

CMD ["php-fpm"]

First, the Dockerfile creates an image on top of the [php:7.2-fpm Docker image](https://hub.docker.com/_/php/). This is a Debian-based image that has the PHP FastCGI implementation [PHP-FPM](https://php-fpm.org/) installed. The file also installs the prerequisite packages for Laravel: mcrypt, pdo\_mysql, mbstring, and imagick with composer.

The RUN directive specifies the commands to update, install, and configure settings inside the container, including creating a dedicated user and group called www. The WORKDIR instruction specifies the /var/www directory as the working directory for the application.

Creating a dedicated user and group with restricted permissions mitigates the inherent vulnerability when running Docker containers, which run by default as root. Instead of running this container as root, we've created the www user, who has read/write access to the /var/www folder thanks to the COPY instruction that we are using with the --chown flag to copy the application folder's permissions.

Finally, the EXPOSE command exposes a port in the container, 9000, for the php-fpm server. CMDspecifies the command that should run once the container is created. Here, CMD specifies "php-fpm", which will start the server.

Save the file and exit your editor when you are finished making changes.

You can now move on to defining your PHP configuration.

Step 5 — Configuring PHP

Now that you have defined your infrastructure in the docker-compose file, you can configure the PHP service to act as a PHP processor for incoming requests from Nginx.

To configure PHP, you will create the local.ini file inside the php folder. This is the file that you bind-mounted to /usr/local/etc/php/conf.d/local.ini inside the container in Step 2. Creating this file will allow you to override the default php.ini file that PHP reads when it starts.

Create the php directory:

* mkdir ~/laravel-app/php

Next, open the local.ini file:

* nano ~/laravel-app/php/local.ini

To demonstrate how to configure PHP, we'll add the following code to set size limitations for uploaded files:

~/laravel-app/php/local.ini

upload\_max\_filesize=40M

post\_max\_size=40M

The upload\_max\_filesize and post\_max\_size directives set the maximum allowed size for uploaded files, and demonstrate how you can set php.ini configurations from your local.ini file. You can put any PHP-specific configuration that you want to override in the local.ini file.

Save the file and exit your editor.

With your PHP local.ini file in place, you can move on to configuring Nginx.

Step 6 — Configuring Nginx

With the PHP service configured, you can modify the Nginx service to use PHP-FPM as the FastCGI server to serve dynamic content. The FastCGI server is based on a binary protocol for interfacing interactive programs with a web server. For more information, please refer to this article on [Understanding and Implementing FastCGI Proxying in Nginx](https://www.digitalocean.com/community/tutorials/understanding-and-implementing-fastcgi-proxying-in-nginx).

To configure Nginx, you will create an app.conf file with the service configuration in the ~/laravel-app/nginx/conf.d/ folder.

First, create the nginx/conf.d/ directory:

* mkdir -p ~/laravel-app/nginx/conf.d

Next, create the app.conf configuration file:

* nano ~/laravel-app/nginx/conf.d/app.conf

Add the following code to the file to specify your Nginx configuration:

~/laravel-app/nginx/conf.d/app.conf

**server** {

**listen** 80;

**index** index.php index.html;

**error\_log** /var/log/nginx/error.log;

**access\_log** /var/log/nginx/access.log;

**root** /var/www/public;

**location** ~ \.php$ {

**try\_files** $uri =404;

**fastcgi\_split\_path\_info** ^(.+\.php)(/.+)$;

**fastcgi\_pass** app:9000;

**fastcgi\_index** index.php;

**include** fastcgi\_params;

**fastcgi\_param** SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

**fastcgi\_param** PATH\_INFO $fastcgi\_path\_info;

}

**location** / {

**try\_files** $uri $uri/ /index.php?$query\_string;

**gzip\_static** **on**;

}

}

The [server block](https://www.digitalocean.com/community/tutorials/understanding-nginx-server-and-location-block-selection-algorithms) defines the configuration for the Nginx web server with the following directives:

* listen: This directive defines the port on which the server will listen to incoming requests.
* error\_log and access\_log: These directives define the files for writing logs.
* root: This directive sets the root folder path, forming the complete path to any requested file on the local file system.

In the php location block, the fastcgi\_pass directive specifies that the app service is listening on a TCP socket on port 9000. This makes the PHP-FPM server listen over the network rather than on a Unix socket. Though a Unix socket has a slight advantage in speed over a TCP socket, it does not have a network protocol and thus skips the network stack. For cases where hosts are located on one machine, a Unix socket may make sense, but in cases where you have services running on different hosts, a TCP socket offers the advantage of allowing you to connect to distributed services. Because our app container is running on a different host from our webserver container, a TCP socket makes the most sense for our configuration.

Save the file and exit your editor when you are finished making changes.

Thanks to the bind mount you created in Step 2, any changes you make inside the nginx/conf.d/folder will be directly reflected inside the webserver container.

Next, let's look at our MySQL settings.

Step 7 — Configuring MySQL

With PHP and Nginx configured, you can enable MySQL to act as the database for your application.

To configure MySQL, you will create the my.cnf file in the mysql folder. This is the file that you bind-mounted to /etc/mysql/my.cnf inside the container in Step 2. This bind mount allows you to override the my.cnf settings as and when required.

To demonstrate how this works, we'll add settings to the my.cnf file that enable the general query log and specify the log file.

First, create the mysql directory:

* mkdir ~/laravel-app/mysql

Next, make the my.cnf file:

* nano ~/laravel-app/mysql/my.cnf

In the file, add the following code to enable the query log and set the log file location:

~/laravel-app/mysql/my.cnf

**[mysqld]**

general\_log = 1

general\_log\_file = /var/lib/mysql/general.log

This my.cnf file enables logs, defining the general\_log setting as 1 to allow general logs. The general\_log\_file setting specifies where the logs will be stored.

Save the file and exit your editor.

Our next step will be to start the containers.

Step 8 — Running the Containers and Modifying Environment Settings

Now that you have defined all of your services in your docker-compose file and created the configuration files for these services, you can start the containers. As a final step, though, we will make a copy of the .env.example file that Laravel includes by default and name the copy .env, which is the file Laravel expects to define its environment:

* cp .env.example .env

We will configure the specific details of our setup in this file once we have started the containers.

With all of your services defined in your docker-compose file, you just need to issue a single command to start all of the containers, create the volumes, and set up and connect the networks:

* docker-compose up -d

When you run docker-compose up for the first time, it will download all of the necessary Docker images, which might take a while. Once the images are downloaded and stored in your local machine, Compose will create your containers. The -d flag daemonizes the process, running your containers in the background.

Once the process is complete, use the following command to list all of the running containers:

* docker ps

You will see the following output with details about your app, webserver, and db containers:

Output

CONTAINER ID NAMES IMAGE STATUS PORTS

c31b7b3251e0 db mysql:5.7.22 Up 2 seconds 0.0.0.0:3306->3306/tcp

ed5a69704580 app digitalocean.com/php Up 2 seconds 9000/tcp

5ce4ee31d7c0 webserver nginx:alpine Up 2 seconds 0.0.0.0:80->80/tcp, 0.0.0.0:443->443/tcp

The CONTAINER ID in this output is a unique identifier for each container, while NAMES lists the service name associated with each. You can use both of these identifiers to access the containers. IMAGE defines the image name for each container, while STATUS provides information about the container's state: whether it's running, restarting, or stopped.

You can now modify the .env file on the app container to include specific details about your setup.

Open the file using docker-compose exec, which allows you to run specific commands in containers. In this case, you are opening the file for editing:

* docker-compose exec app nano .env

Find the block that specifies DB\_CONNECTION and update it to reflect the specifics of your setup. You will modify the following fields:

* DB\_HOST will be your db database container.
* DB\_DATABASE will be the laravel database.
* DB\_USERNAME will be the username you will use for your database. In this case, we will use laraveluser.
* DB\_PASSWORD will be the secure password you would like to use for this user account.

/var/www/.env

DB\_CONNECTION=mysql

DB\_HOST=db

DB\_PORT=3306

DB\_DATABASE=laravel

DB\_USERNAME=laraveluser

DB\_PASSWORD=your\_laravel\_db\_password

Save your changes and exit your editor.

Next, set the application key for the Laravel application with the php artisan key:generatecommand. This command will generate a key and copy it to your .env file, ensuring that your user sessions and encrypted data remain secure:

* docker-compose exec app php artisan key:generate

You now have the environment settings required to run your application. To cache these settings into a file, which will boost your application's load speed, run:

* docker-compose exec app php artisan config:cache

Your configuration settings will be loaded into /var/www/bootstrap/cache/config.php on the container.

As a final step, visit http://your\_server\_ip in the browser. You will see the following home page for your Laravel application:

With your containers running and your configuration information in place, you can move on to configuring your user information for the laravel database on the db container.

Step 9 — Creating a User for MySQL

The default MySQL installation only creates the root administrative account, which has unlimited privileges on the database server. In general, it's better to avoid using the root administrative account when interacting with the database. Instead, let's create a dedicated database user for our application's Laravel database.

To create a new user, execute an interactive bash shell on the db container with docker-compose exec:

* docker-compose exec db bash

Inside the container, log into the MySQL root administrative account:

* mysql -u root -p

You will be prompted for the password you set for the MySQL root account during installation in your docker-compose file.

Start by checking for the database called laravel, which you defined in your docker-compose file. Run the show databases command to check for existing databases:

* show databases;

You will see the laravel database listed in the output:

Output

+--------------------+

| Database |

+--------------------+

| information\_schema |

| laravel |

| mysql |

| performance\_schema |

| sys |

+--------------------+

5 rows in set (0.00 sec)

Next, create the user account that will be allowed to access this database. Our username will be laraveluser, though you can replace this with another name if you'd prefer. Just be sure that your username and password here match the details you set in your .env file in the previous step:

* GRANT ALL ON laravel.\* TO 'laraveluser'@'%' IDENTIFIED BY 'your\_laravel\_db\_password';

Flush the privileges to notify the MySQL server of the changes:

* FLUSH PRIVILEGES;

Exit MySQL:

* EXIT;

Finally, exit the container:

* exit

You have configured the user account for your Laravel application database and are ready to migrate your data and work with the Tinker console.

Step 10 — Migrating Data and Working with the Tinker Console

With your application running, you can migrate your data and experiment with the tinkercommand, which will initiate a [*PsySH*](http://psysh.org/) console with Laravel preloaded. PsySH is a runtime developer console and interactive debugger for PHP, and Tinker is a REPL specifically for Laravel. Using the tinker command will allow you to interact with your Laravel application from the command line in an interactive shell.

First, test the connection to MySQL by running the Laravel artisan migrate command, which creates a migrations table in the database from inside the container:

* docker-compose exec app php artisan migrate

This command will migrate the default Laravel tables. The output confirming the migration will look like this:

Output

Migration table created successfully.

Migrating: 2014\_10\_12\_000000\_create\_users\_table

Migrated: 2014\_10\_12\_000000\_create\_users\_table

Migrating: 2014\_10\_12\_100000\_create\_password\_resets\_table

Migrated: 2014\_10\_12\_100000\_create\_password\_resets\_table

Once the migration is complete, you can run a query to check if you are properly connected to the database using the tinker command:

* docker-compose exec app php artisan tinker

Test the MySQL connection by getting the data you just migrated:

* \DB::table('migrations')->get();

You will see output that looks like this:

Output

=> Illuminate\Support\Collection {#2856

all: [

{#2862

+"id": 1,

+"migration": "2014\_10\_12\_000000\_create\_users\_table",

+"batch": 1,

},

{#2865

+"id": 2,

+"migration": "2014\_10\_12\_100000\_create\_password\_resets\_table",

+"batch": 1,

},

],

}

You can use tinker to interact with your databases and to experiment with services and models.

With your Laravel application in place, you are ready for further development and experimentation.

Conclusion

You now have a LEMP stack application running on your server, which you've tested by accessing the Laravel welcome page and creating MySQL database migrations.

Key to the simplicity of this installation is Docker Compose, which allows you to create a group of Docker containers, defined in a single file, with a single command. If you would like to learn more about how to do CI with Docker Compose, take a look at [How To Configure a Continuous Integration Testing Environment with Docker and Docker Compose on Ubuntu 16.04](https://www.digitalocean.com/community/tutorials/how-to-configure-a-continuous-integration-testing-environment-with-docker-and-docker-compose-on-ubuntu-16-04). If you want to streamline your Laravel application deployment process then [How to Automatically Deploy Laravel Applications with Deployer on Ubuntu 16.04](https://www.digitalocean.com/community/tutorials/automatically-deploy-laravel-applications-deployer-ubuntu) will be a relevant resource.