CONTENTS

Prerequisites

Step 1 — Obtaining the Demo Application

Step 2 — Setting Up the Application's .env File

Step 3 — Setting Up the Application's Dockerfile

Step 4 — Setting Up Nginx Configuration and Database Dump Files

Step 5 — Creating a Multi-Container Environment with Docker Compose

Step 6 — Running the Application with Docker Compose

Conclusion

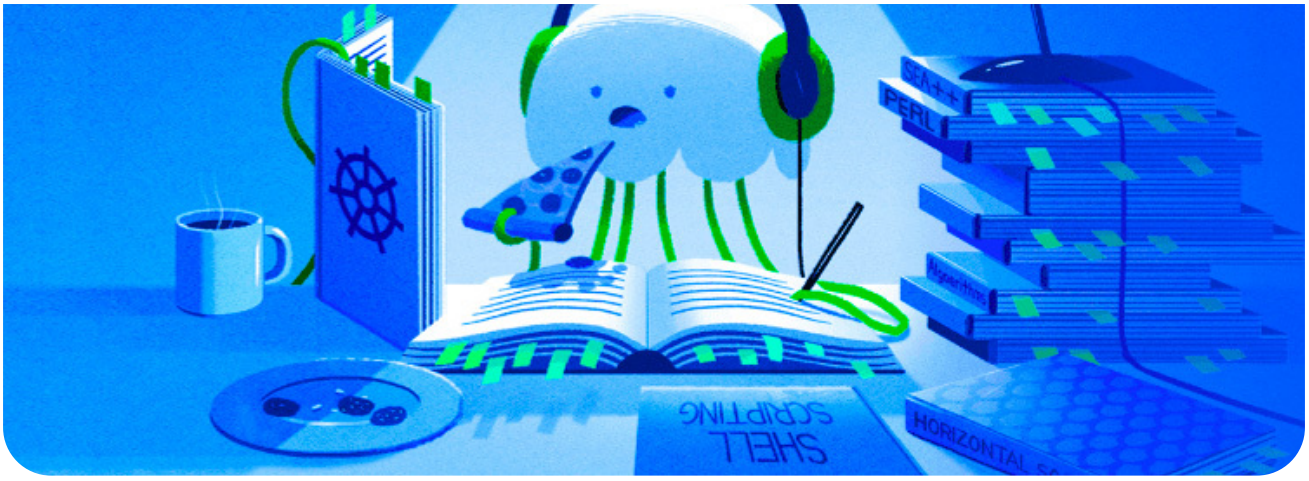
// Tutorial //

How To Install and Set Up Laravel with Docker Compose on Ubuntu 22.04

Published on April 25, 2022

Container Laravel LEMP PHP PHP Frameworks Docker Ubuntu 22.04 Ubuntu

By Erika Heidi and Jamon Camisso



Not using Ubuntu 22.04?
Choose a different version or distribution.

Ubuntu 22.04

Introduction

To *containerize* an application refers to the process of adapting an application and its components in order to be able to run it in lightweight environments known as containers. Such environments are isolated and disposable, and can be leveraged for developing, testing, and deploying applications to production.

In this guide, we'll use Docker Compose to containerize a Laravel application for development. When you're finished, you'll have a demo Laravel application running on three separate service containers:

- An `app` service running PHP7.4-FPM;
- A `db` service running MySQL 5.7;
- An `nginx` service that uses the `app` service to parse PHP code before serving the Laravel application to the final user.

To allow for a streamlined development process and facilitate application debugging, we'll keep application files in sync by using shared volumes. We'll also see how to use `docker-compose exec` commands to run Composer and Artisan on the `app` container.

Prerequisites

s to an Ubuntu 22.04 local machine or development server as a non-root ith sudo privileges. If you're using a remote server, it's advisable to have an active firewall installed. To set these up, please refer to our Initial Server

Setup Guide for Ubuntu 22.04.
- Docker installed on your server, following Steps 1 and 2 of How To Install and Use Docker on Ubuntu 22.04.
- Docker Compose installed on your server, following Step 1 of How To Install and Use Docker Compose on Ubuntu 22.04.

Step 1 – Obtaining the Demo Application

To get started, we'll fetch the demo Laravel application from its Github repository. We're interested in the `tutorial-01` branch, which contains the basic Laravel application we've created in the first guide of this series.

To obtain the application code that is compatible with this tutorial, download release `tutorial-1.0.1` to your home directory with:

```
$ cd ~
$ curl -L https://github.com/do-community/travellist-laravel-demo/archive/t
```

Copy

We'll need the `unzip` command to unpack the application code. In case you haven't installed this package before, do so now with:

```
$ sudo apt update
$ sudo apt install unzip
```

Copy

Now, unzip the contents of the application and rename the unpacked directory for easier access:

```
$ unzip travellist.zip
$ mv travellist-laravel-demo-tutorial-1.0.1 travellist-demo
```

Copy

Navigate to the `travellist-demo` directory:

```
$ cd travellist-demo
```

Copy

In the next step, we'll create a `.env` configuration file to set up the application.

Step 2 – Setting Up the Application's `.env` File

Th configuration files are located in a directory called `config`, inside the ap s root directory. Additionally, a `.env` file is used to set up environment-dependent configuration, such as credentials and any information that might vary

between deploys. This file is not included in revision control.

> **Warning**: The environment configuration file contains sensitive information about your server, including database credentials and security keys. For that reason, you should never share this file publicly.

The values contained in the `.env` file will take precedence over the values set in regular configuration files located at the `config` directory. Each installation on a new environment requires a tailored environment file to define things such as database connection settings, debug options, application URL, among other items that may vary depending on which environment the application is running.

We'll now create a new `.env` file to customize the configuration options for the development environment we're setting up. Laravel comes with an example `.env` file that we can copy to create our own:

```
$ cp .env.example .env
```
Copy

Open this file using `nano` or your text editor of choice:

```
$ nano .env
```
Copy

The current `.env` file from the `travellist` demo application contains settings to use a local MySQL database, with `127.0.0.1` as database host. We need to update the `DB_HOST` variable so that it points to the database service we will create in our Docker environment. In this guide, we'll call our database service `db`. Go ahead and replace the listed value of `DB_HOST` with the database service name:

.env

```
APP_NAME=Travellist
APP_ENV=dev
APP_KEY=
APP_DEBUG=true
APP_URL=http://localhost:8000

LOG_CHANNEL=stack

DB_CONNECTION=mysql
D[redacted]db
[redacted]306
[redacted]SE=travellist
D[redacted]AME=travellist_user
DB_PASSWORD=password
```

```
...
```

Feel free to also change the database name, username, and password, if you wish. These variables will be leveraged in a later step where we'll set up the `docker-compose.yml` file to configure our services.

Save the file when you're done editing. If you used `nano`, you can do that by pressing `Ctrl+x`, then `Y` and `Enter` to confirm.

Step 3 – Setting Up the Application's Dockerfile

Although both our MySQL and Nginx services will be based on default images obtained from the Docker Hub, we still need to build a custom image for the application container. We'll create a new Dockerfile for that.

Our **travellist** image will be based on the `php:7.4-fpm` official PHP image from Docker Hub. On top of that basic PHP-FPM environment, we'll install a few extra PHP modules and the Composer dependency management tool.

We'll also create a new system user; this is necessary to execute `artisan` and `composer` commands while developing the application. The `uid` setting ensures that the user inside the container has the same uid as your system user on your host machine, where you're running Docker. This way, any files created by these commands are replicated in the host with the correct permissions. This also means that you'll be able to use your code editor of choice in the host machine to develop the application that is running inside containers.

Create a new Dockerfile with:

```
$ nano Dockerfile
```

Copy

Copy the following contents to your Dockerfile:

Dockerfile

```
FROM php:7.4-fpm

# Arguments defined in docker-compose.yml
ARG user
ARG uid

# Install system dependencies
RUN apt-get update && apt-get install -y \
    git \
    curl \
```

```
    libpng-dev \
    libonig-dev \
    libxml2-dev \
    zip \
    unzip

# Clear cache
RUN apt-get clean && rm -rf /var/lib/apt/lists/*

# Install PHP extensions
RUN docker-php-ext-install pdo_mysql mbstring exif pcntl bcmath gd

# Get latest Composer
COPY --from=composer:latest /usr/bin/composer /usr/bin/composer

# Create system user to run Composer and Artisan Commands
RUN useradd -G www-data,root -u $uid -d /home/$user $user
RUN mkdir -p /home/$user/.composer && \
    chown -R $user:$user /home/$user

# Set working directory
WORKDIR /var/www

USER $user
```

Don't forget to save the file when you're done.

Our Dockerfile starts by defining the base image we're using: `php:7.4-fpm`.

After installing system packages and PHP extensions, we install Composer by copying the `composer` executable from its latest official image to our own application image.

A new system user is then created and set up using the `user` and `uid` arguments that were declared at the beginning of the Dockerfile. These values will be injected by Docker Compose at build time.

Finally, we set the default working dir as `/var/www` and change to the newly created user. This will make sure you're connecting as a regular user, and that you're on the right directory, when running `composer` and `artisan` commands on the application container.

Step 4 – Setting Up Nginx Configuration and Database Dump Files

When building development environments with Docker Compose, it is often necessary to share configuration or initialization files with service containers, in order

to set up or bootstrap those services. This practice facilitates making changes to configuration files to fine-tune your environment while you're developing the application.

We'll now set up a folder with files that will be used to configure and initialize our service containers.

To set up Nginx, we'll share a `travellist.conf` file that will configure how the application is served. Create the `docker-compose/nginx` folder with:

```
$ mkdir -p docker-compose/nginx
```

Copy

Open a new file named `travellist.conf` within that directory:

```
$ nano docker-compose/nginx/travellist.conf
```

Copy

Copy the following Nginx configuration to that file:

docker-compose/nginx/travellist.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;
    }
}
```

Th configure Nginx to listen on port `80` and use `index.php` as default index pa set the document root to `/var/www/public`, and then configure Nginx to us service on port `9000` to process `*.php` files.

Save and close the file when you're done editing.

To set up the MySQL database, we'll share a database dump that will be imported when the container is initialized. This is a feature provided by the MySQL 5.7 image we'll be using on that container.

Create a new folder for your MySQL initialization files inside the `docker-compose` folder:

```
$ mkdir docker-compose/mysql
```
Copy

Open a new `.sql` file:

```
$ nano docker-compose/mysql/init_db.sql
```
Copy

The following MySQL dump is based on the database we've set up in our Laravel on LEMP guide. It will create a new table named `places`. Then, it will populate the table with a set of sample places.

Add the following code to the file:

docker-compose/mysql/db_init.sql

```
DROP TABLE IF EXISTS `places`;

CREATE TABLE `places` (
  `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(255) COLLATE utf8mb4_unicode_ci NOT NULL,
  `visited` tinyint(1) NOT NULL DEFAULT '0',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=12 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_un

INSERT INTO `places` (name, visited) VALUES ('Berlin',0),('Budapest',0),('Ci
```
Copy

The `places` table contains three fields: `id`, `name`, and `visited`. The `visited` field is a flag used to identify the places that are still *to go*. Feel free to change the sample places or include new ones. Save and close the file when you're done.

We've finished setting up the application's Dockerfile and the service configuration files. Next, we'll set up Docker Compose to use these files when creating our services.

Step 5 – Creating a Multi-Container Environment with Docker Compose

Docker Compose enables you to create multi-container environments for applications

running on Docker. It uses *service definitions* to build fully customizable environments with multiple containers that can share networks and data volumes. This allows for a seamless integration between application components.

To set up our service definitions, we'll create a new file called `docker-compose.yml`. Typically, this file is located at the root of the application folder, and it defines your containerized environment, including the base images you will use to build your containers, and how your services will interact.

We'll define three different services in our `docker-compose.yml` file: `app`, `db`, and `nginx`.

The `app` service will build an image called `travellist`, based on the Dockerfile we've previously created. The container defined by this service will run a `php-fpm` server to parse PHP code and send the results back to the `nginx` service, which will be running on a separate container. The `mysql` service defines a container running a MySQL 5.7 server. Our services will share a bridge network named `travellist`.

The application files will be synchronized on both the `app` and the `nginx` services via **bind mounts**. Bind mounts are useful in development environments because they allow for a performant two-way sync between host machine and containers.

Create a new `docker-compose.yml` file at the root of the application folder:

```
$ nano docker-compose.yml
```

Copy

A typical `docker-compose.yml` file starts with a version definition, followed by a `services` node, under which all services are defined. Shared networks are usually defined at the bottom of that file.

To get started, copy this boilerplate code into your `docker-compose.yml` file:

docker-compose.yml

```
version: "3.7"
services:


networks:
  travellist:
    driver: bridge
```

We'll edit the `services` node to include the `app`, `db` and `nginx` services.

The app Service

The `app` service will set up a container named `travellist-app`. It builds a new Docker image based on a Dockerfile located in the same path as the `docker-compose.yml` file. The new image will be saved locally under the name `travellist`.

Even though the document root being served as the application is located in the `nginx` container, we need the application files somewhere inside the `app` container as well, so we're able to execute command line tasks with the Laravel Artisan tool.

Copy the following service definition under your `services` node, inside the `docker-compose.yml` file:

docker-compose.yml

```
app:
  build:
    args:
      user: sammy
      uid: 1000
    context: ./
    dockerfile: Dockerfile
  image: travellist
  container_name: travellist-app
  restart: unless-stopped
  working_dir: /var/www/
  volumes:
    - ./:/var/www
  networks:
    - travellist
```

These settings do the following:

- `build`: This configuration tells Docker Compose to build a local image for the `app` service, using the specified path (context) and Dockerfile for instructions. The arguments `user` and `uid` are injected into the Dockerfile to customize user creation commands at build time.
- `image`: The name that will be used for the image being built.
- `container_name`: Sets up the container name for this service.
- `restart`: Always restart, unless the service is stopped.
- `working_dir`: Sets the default directory for this service as `/var/www`.
- `volumes`: Creates a shared volume that will synchronize contents from the current directory to `/var/www` inside the container. Notice that this is not your document root, since that will live in the `nginx` container.
- `networks`: Sets up this service to use a network named `travellist`.

The db Service

The `db` service uses a pre-built MySQL 8.0 image from Docker Hub. Because Docker Compose automatically loads `.env` variable files located in the same directory as the `docker-compose.yml` file, we can obtain our database settings from the Laravel `.env` file we created in a previous step.

Include the following service definition in your `services` node, right after the `app` service:

docker-compose.yml

```
db:
  image: mysql:8.0
  container_name: travellist-db
  restart: unless-stopped
  environment:
    MYSQL_DATABASE: ${DB_DATABASE}
    MYSQL_ROOT_PASSWORD: ${DB_PASSWORD}
    MYSQL_PASSWORD: ${DB_PASSWORD}
    MYSQL_USER: ${DB_USERNAME}
    SERVICE_TAGS: dev
    SERVICE_NAME: mysql
  volumes:
    - ./docker-compose/mysql:/docker-entrypoint-initdb.d
  networks:
    - travellist
```

These settings do the following:

- `image`: Defines the Docker image that should be used for this container. In this case, we're using a MySQL 5.7 image from Docker Hub.
- `container_name`: Sets up the container name for this service: `travellist-db`.
- `restart`: Always restart this service, unless it is explicitly stopped.
- `environment`: Defines environment variables in the new container. We're using values obtained from the Laravel `.env` file to set up our MySQL service, which will automatically create a new database and user based on the provided environment variables.
- `volumes`: Creates a volume to share a `.sql` database dump that will be used to initialize the application database. The MySQL image will automatically import `.sql` files placed in the `/docker-entrypoint-initdb.d` directory inside the container.
- `networks`: Sets up this service to use a network named `travellist`.

Th Service

The `nginx` service uses a pre-built Nginx image on top of Alpine, a lightweight Linux distribution. It creates a container named `travellist-nginx`, and it uses the `ports`

definition to create a redirection from port `8000` on the host system to port `80` inside the container.

Include the following service definition in your `services` node, right after the `db` service:

docker-compose.yml

```
nginx:
  image: nginx:1.17-alpine
  container_name: travellist-nginx
  restart: unless-stopped
  ports:
    - 8000:80
  volumes:
    - ./:/var/www
    - ./docker-compose/nginx:/etc/nginx/conf.d
  networks:
    - travellist
```

These settings do the following:

- `image`: Defines the Docker image that should be used for this container. In this case, we're using the Alpine Nginx 1.17 image.
- `container_name`: Sets up the container name for this service: **travellist-nginx**.
- `restart`: Always restart this service, unless it is explicitly stopped.
- `ports`: Sets up a port redirection that will allow external access via port `8000` to the web server running on port `80` inside the container.
- `volumes`: Creates **two** shared volumes. The first one will synchronize contents from the current directory to `/var/www` inside the container. This way, when you make local changes to the application files, they will be quickly reflected in the application being served by Nginx inside the container. The second volume will make sure our Nginx configuration file, located at `docker-compose/nginx/travellist.conf`, is copied to the container's Nginx configuration folder.
- `networks`: Sets up this service to use a network named `travellist`.

Finished `docker-compose.yml` File

This is how our finished `docker-compose.yml` file looks like:

docker-compose.yml

```
version: "3.7"
services:
  build:
    args:
```

```
          user: sammy
          uid: 1000
        context: ./
        dockerfile: Dockerfile
      image: travellist
      container_name: travellist-app
      restart: unless-stopped
      working_dir: /var/www/
      volumes:
        - ./:/var/www
      networks:
        - travellist

    db:
      image: mysql:8.0
      container_name: travellist-db
      restart: unless-stopped
      environment:
        MYSQL_DATABASE: ${DB_DATABASE}
        MYSQL_ROOT_PASSWORD: ${DB_PASSWORD}
        MYSQL_PASSWORD: ${DB_PASSWORD}
        MYSQL_USER: ${DB_USERNAME}
        SERVICE_TAGS: dev
        SERVICE_NAME: mysql
      volumes:
        - ./docker-compose/mysql:/docker-entrypoint-initdb.d
      networks:
        - travellist

    nginx:
      image: nginx:alpine
      container_name: travellist-nginx
      restart: unless-stopped
      ports:
        - 8000:80
      volumes:
        - ./:/var/www
        - ./docker-compose/nginx:/etc/nginx/conf.d/
      networks:
        - travellist

networks:
  travellist:
    driver: bridge
```

Make sure you save the file when you're done.

S — Running the Application with Docker Compose

We'll now use `docker-compose` commands to build the application image and run the services we specified in our setup.

Build the `app` image with the following command:

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

Copy

This command might take a few minutes to complete. You'll see output similar to this:

Output

```
Building app
Sending build context to Docker daemon  377.3kB
Step 1/11 : FROM php:7.4-fpm
 ---> 8c08d993542f
Step 2/11 : ARG user
 ---> e3ce3af04d87
Step 3/11 : ARG uid
 ---> 30cb921ef7df
Step 4/11 : RUN apt-get update && apt-get install -y     git     curl     li
. . .
 ---> b6dbc7a02e95
Step 5/11 : RUN apt-get clean && rm -rf /var/lib/apt/lists/*
 ---> 10ef9dde45ad
. . .
Step 6/11 : RUN docker-php-ext-install pdo_mysql mbstring exif pcntl bcmath
. . .
 ---> 920e4f09ec75
Step 7/11 : COPY --from=composer:latest /usr/bin/composer /usr/bin/composer
 ---> dbbcd44e44af
Step 8/11 : RUN useradd -G www-data,root -u $uid -d /home/$user $user
 ---> db98e899a69a
Step 9/11 : RUN mkdir -p /home/$user/.composer &&     chown -R $user:$user /
 ---> 5119e26ebfea
Step 10/11 : WORKDIR /var/www
 ---> 699c491611c0
Step 11/11 : USER $user
 ---> cf250fe8f1af
Successfully built cf250fe8f1af
Successfully tagged travellist:latest
```

When the build is finished, you can run the environment in background mode with:

```
-compose up -d
```

Copy

Output

```
Creating travellist-db    ... done
Creating travellist-app   ... done
Creating travellist-nginx ... done
```

This will run your containers in the background. To show information about the state of your active services, run:

```
$ docker-compose ps
```
Copy

You'll see output like this:

```
Output
      Name                     Command               State                  Po
-------------------------------------------------------------------------------
travellist-app     docker-php-entrypoint php-fpm   Up      9000/tcp
travellist-db      docker-entrypoint.sh mysqld     Up      3306/tcp, 33060/t
travellist-nginx   nginx -g daemon off;            Up      0.0.0.0:8000->80/
```

Your environment is now up and running, but we still need to execute a couple commands to finish setting up the application. You can use the `docker-compose exec` command to execute commands in the service containers, such as an `ls -l` to show detailed information about files in the application directory:

```
$ docker-compose exec app ls -l
```
Copy

```
Output
total 256
-rw-r--r-- 1 sammy sammy    737 Apr 18 14:21 Dockerfile
-rw-r--r-- 1 sammy sammy    101 Jan  7  2020 README.md
drwxr-xr-x 6 sammy sammy   4096 Jan  7  2020 app
-rwxr-xr-x 1 sammy sammy   1686 Jan  7  2020 artisan
drwxr-xr-x 3 sammy sammy   4096 Jan  7  2020 bootstrap
-rw-r--r-- 1 sammy sammy   1501 Jan  7  2020 composer.json
-rw-r--r-- 1 sammy sammy 179071 Jan  7  2020 composer.lock
drwxr-xr-x 2 sammy sammy   4096 Jan  7  2020 config
drwxr-xr-x 5 sammy sammy   4096 Jan  7  2020 database
drwxr-xr-x 4 sammy sammy   4096 Apr 18 14:22 docker-compose
-rw-r--r-- 1 sammy sammy   1017 Apr 18 14:29 docker-compose.yml
-rw-r--r-- 1 sammy sammy   1013 Jan  7  2020 package.json
-rw-r--r-- 1 sammy sammy   1405 Jan  7  2020 phpunit.xml
d       -x 2 sammy sammy   4096 Jan  7  2020 public
        - 1 sammy sammy    273 Jan  7  2020 readme.md
        x 6 sammy sammy   4096 Jan  7  2020 resources
        -x 2 sammy sammy   4096 Jan  7  2020 routes
-rw-r--r-- 1 sammy sammy    563 Jan  7  2020 server.php
```

```
drwxr-xr-x 5 sammy sammy   4096 Jan  7  2020 storage
drwxr-xr-x 4 sammy sammy   4096 Jan  7  2020 tests
-rw-r--r-- 1 sammy sammy    538 Jan  7  2020 webpack.mix.js
```

We'll now run `composer install` to install the application dependencies:

```
$ docker-compose exec app rm -rf vendor composer.lock
$ docker-compose exec app composer install
```

Copy

You'll see output like this:

Output

```
No composer.lock file present. Updating dependencies to latest instead of in
. . .
Lock file operations: 89 installs, 0 updates, 0 removals
  - Locking doctrine/inflector (2.0.4)
  - Locking doctrine/instantiator (1.4.1)
  - Locking doctrine/lexer (1.2.3)
  - Locking dragonmantank/cron-expression (v2.3.1)
  - Locking egulias/email-validator (2.1.25)
  - Locking facade/flare-client-php (1.9.1)
  - Locking facade/ignition (1.18.1)
  - Locking facade/ignition-contracts (1.0.2)
  - Locking fideloper/proxy (4.4.1)
  - Locking filp/whoops (2.14.5)
. . .
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 89 installs, 0 updates, 0 removals
  - Downloading doctrine/inflector (2.0.4)
  - Downloading doctrine/lexer (1.2.3)
  - Downloading dragonmantank/cron-expression (v2.3.1)
  - Downloading symfony/polyfill-php80 (v1.25.0)
  - Downloading symfony/polyfill-php72 (v1.25.0)
  - Downloading symfony/polyfill-mbstring (v1.25.0)
  - Downloading symfony/var-dumper (v4.4.39)
  - Downloading symfony/deprecation-contracts (v2.5.1)
. . .
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postAutoloadDump
> @php artisan package:discover --ansi
Discovered Package: facade/ignition
Discovered Package: fideloper/proxy
Discovered Package: laravel/tinker
[          d Package: nesbot/carbon
[          d Package: nunomaduro/collision
[          anifest generated successfully.
```

The last thing we need to do before testing the application is to generate a unique application key with the `artisan` Laravel command-line tool. This key is used to encrypt user sessions and other sensitive data:

```
$ docker-compose exec app php artisan key:generate
```
Copy

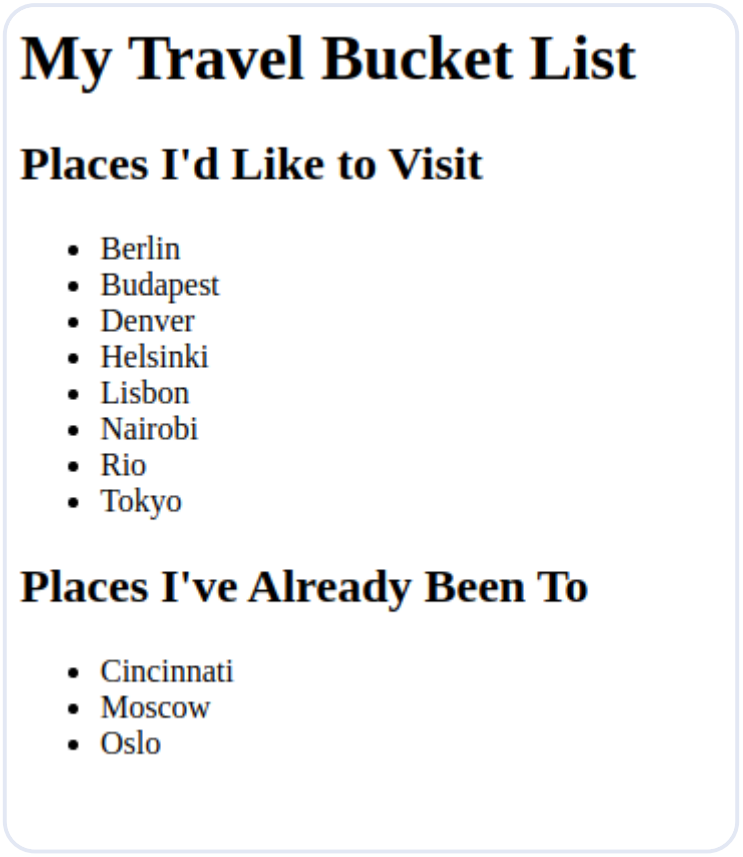
```
Output
Application key set successfully.
```

Now go to your browser and access your server's domain name or IP address on port 8000:

```
http://server_domain_or_IP:8000
```

Note: In case you are running this demo on your local machine, use `http://localhost:8000` to access the application from your browser.

You'll see a page like this:



You can use the `logs` command to check the logs generated by your services:

```
$ docker-compose logs nginx
```

Copy

```
Attaching to travellist-nginx
. . .
travellist-nginx | 172.24.9.1 - - [18/Apr/2022:14:49:16 +0000] "GET / HTTP/1
travellist-nginx | 172.24.9.1 - - [18/Apr/2022:14:51:27 +0000] "GET / HTTP/1
travellist-nginx | 172.24.9.1 - - [18/Apr/2022:14:51:27 +0000] "GET /favicon
```

If you want to pause your Docker Compose environment while keeping the state of all its services, run:

```
$ docker-compose pause
```

Copy

Output

```
Pausing travellist-db    ... done
Pausing travellist-nginx ... done
Pausing travellist-app   ... done
```

You can then resume your services with:

```
$ docker-compose unpause
```

Copy

Output

```
Unpausing travellist-app   ... done
Unpausing travellist-nginx ... done
Unpausing travellist-db    ... done
```

To shut down your Docker Compose environment and remove all of its containers, networks, and volumes, run:

```
$ docker-compose down
```

Copy

Output

```
Stopping travellist-nginx ... done
Stopping travellist-db    ... done
S         travellist-app   ... done
          travellist-nginx ... done
          travellist-db    ... done
R         travellist-app   ... done
Removing network travellist-laravel-demo_travellist
```

For an overview of all Docker Compose commands, please check the Docker Compose command-line reference.

Conclusion

In this guide, we've set up a Docker environment with three containers using Docker Compose to define our infrastructure in a YAML file.

From this point on, you can work on your Laravel application without needing to install and set up a local web server for development and testing. Moreover, you'll be working with a disposable environment that can be easily replicated and distributed, which can be helpful while developing your application and also when moving towards a production environment.

About the authors

Erika Heidi Author

Developer Advocate

Dev/Ops passionate about open source, PHP, and Linux.

Jamon Camisso Author

Comments

6 Comments

Leave a comment...

This textbox defaults to using `Markdown` to format your answer.

You can type `!ref` in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment

raymoncada • January 5, 2023

Hi, great tutorial. I am new to PHP and docker. There is a typo in

INSERT INTO `places` (name, visited) VALUES ('Berlin',0),('Budapest',0), ('Cincinnati',1),('Denver',0),('Helsinki',0),('Lisbon',0),('Moscow',1),('Nairobi',0), ('Oslo',1),('Rio',0),('Tokyo',0);

It should be

INSERT INTO `places` (name, visited) VALUES ('Berlin',0),('Budapest',0), ('Cincinnati',1),('Denver',0),('Helsinki',0),('Lisbon',0),('Kyiv',1),('Nairobi',0), ('Oslo',1),('Rio',0),('Tokyo',0);

Best

Reply

9283cc450f990c5083fdaf5 • November 22, 2022

I have an issue at step to install composer:

docker compose exec app rm -rf vendor composer.lock

ERROR: rm: cannot remove ‘composer.lock’: Permission denied

Any solution for this?

Show replies Reply

aiemsoflow • August 28, 2022

Hi, thanks for putting this together. I have a Laravel 9 app, and I followed your tutorial to the letter however I keep getting SQLSTATE[HY000] [1045] Access denied for user ‘root’@‘172.18.0.4’ (using password: YES) been on this for weeks now, googled everywhere on the internet nothing worked. I have made some minor adjustments since then to the docker-compose and Dockerfile, based on my project requirements. I’d really appreciate pointers in the right direction, at this point I am fed up! Thank you.

docker-compose.yml

```
version: "3.7"
services:
    app:
        build: .
        container_name: cookbookshq
        depends_on:
            - db
        volumes:
            - ./:/var/www
        networks:
            - cookbooks
    db:
        image: mysql:latest
        container_name: db
        environment:
            MYSQL_ALLOW_EMPTY_PASSWORD: false
            MYSQL_DATABASE: test_db
            MYSQL_PASSWORD: pass
        volumes:
            - cookbooks-db:/var/lib/mysql
        networks:
            - cookbooks
    nginx:
        image: nginx:alpine
        container_name: web-server
        restart: unless-stopped
```

```
        ports:
            - "8080:80"
        volumes:
            - ./:/var/www
            - ./docker-compose/nginx:/etc/nginx/conf.d/
        networks:
            - cookbooks

volumes:
    cookbooks-db:

networks:
    cookbooks:
        driver: bridge
```

Dockerfile

```
FROM php:8.0-fpm

# Arguments defined in docker-compose.yml
ARG user
ARG uid

# Install system dependencies
RUN apt-get update && apt-get install -y \
    git \
    curl \
    libpng-dev \
    libonig-dev \
    libxml2-dev \
    zip \
    cron \
    unzip \
    redis-tools

# Clear cache
RUN apt-get clean && rm -rf /var/lib/apt/lists/*

# Install PHP extensions
RUN docker-php-ext-install pdo_mysql mbstring exif pcntl bcmath gd

RUN mkdir -p /var/www
WORKDIR /var/www
COPY ./ /var/www/

: latest Composer
--from=composer:latest /usr/bin/composer /usr/bin/composer

# Create system user to run Composer and Artisan Commands
RUN useradd -G www-data,root -u 1000 -d /home/dev dev
```

```
RUN mkdir -p /home/dev/.composer && \
    chown -R dev:dev /home/dev

# Set working directory
WORKDIR /var/www

USER $user
RUN composer install --no-interaction

RUN echo "memory_limit=1024M" >> /usr/local/etc/php/conf.d/php.ini
RUN echo "allow_url_fopen=on" >> /usr/local/etc/php/conf.d/php.ini
```

Reply

d95341d22c2f4c6bab1a9556b8 • August 21, 2022

I tried the above but keep getting the following error

```
In JsonFile.php line 181:

file_put_contents(./composer.lock): failed to open stream: Permiss
n denied   `
```

and the file permission is still root

Show replies Reply

lordisp • June 21, 2022

What's the point in creating a `non-root` user but adding it to the `root` group?

```
RUN useradd -G www-data,root -u $uid -d /home/$user $user
```

Show replies Reply

aytheta • May 2, 2022

Great quality as usual from DO. You intend MySQL 5.7 but the configurations all refer to MySQL8.

Reply

Popular Topics

Ubuntu

Linux Basics

JavaScript

Python

MySQL

Docker

Kubernetes

All tutorials →

Ta n expert →

Congratulations on unlocking the whale ambience easter egg! Click the whale button in the bottom left of your screen to toggle some ambient whale noises while you read.

Thank you to the Glacier Bay National Park & Preserve and Merrick079 for the sounds behind this easter egg.

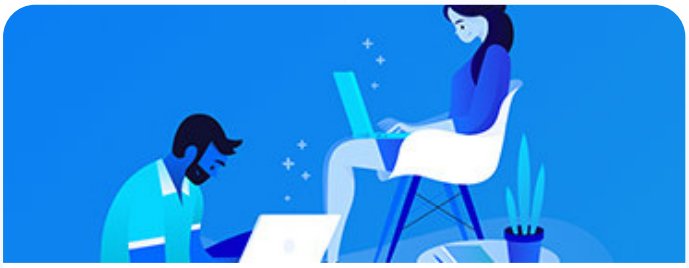
Interested in whales, protecting them, and their connection to helping prevent climate change? We recommend checking out the Whale and Dolphin Conservation.

Reset easter egg to be discovered again / Permanently dismiss and hide easter egg

Hollie's Hub for Good

Working on improving health and education, reducing inequality, and spurring economic growth? We'd like to help.

Learn more →



Become a contributor

You get paid; we donate to tech nonprofits.

Learn more →

Featured on Community

ernetes Course Learn Python 3 Machine Learning in Python

Getting started with Go Intro to Kubernetes