How to integrate Docker into a Symfony-based project

Description of the steps to follow to dockerize a Symfony project that uses Nginx, PHP-FPM and MySQL

Gerardo Fernández
Dec 10, 2019 · 8 min read ★

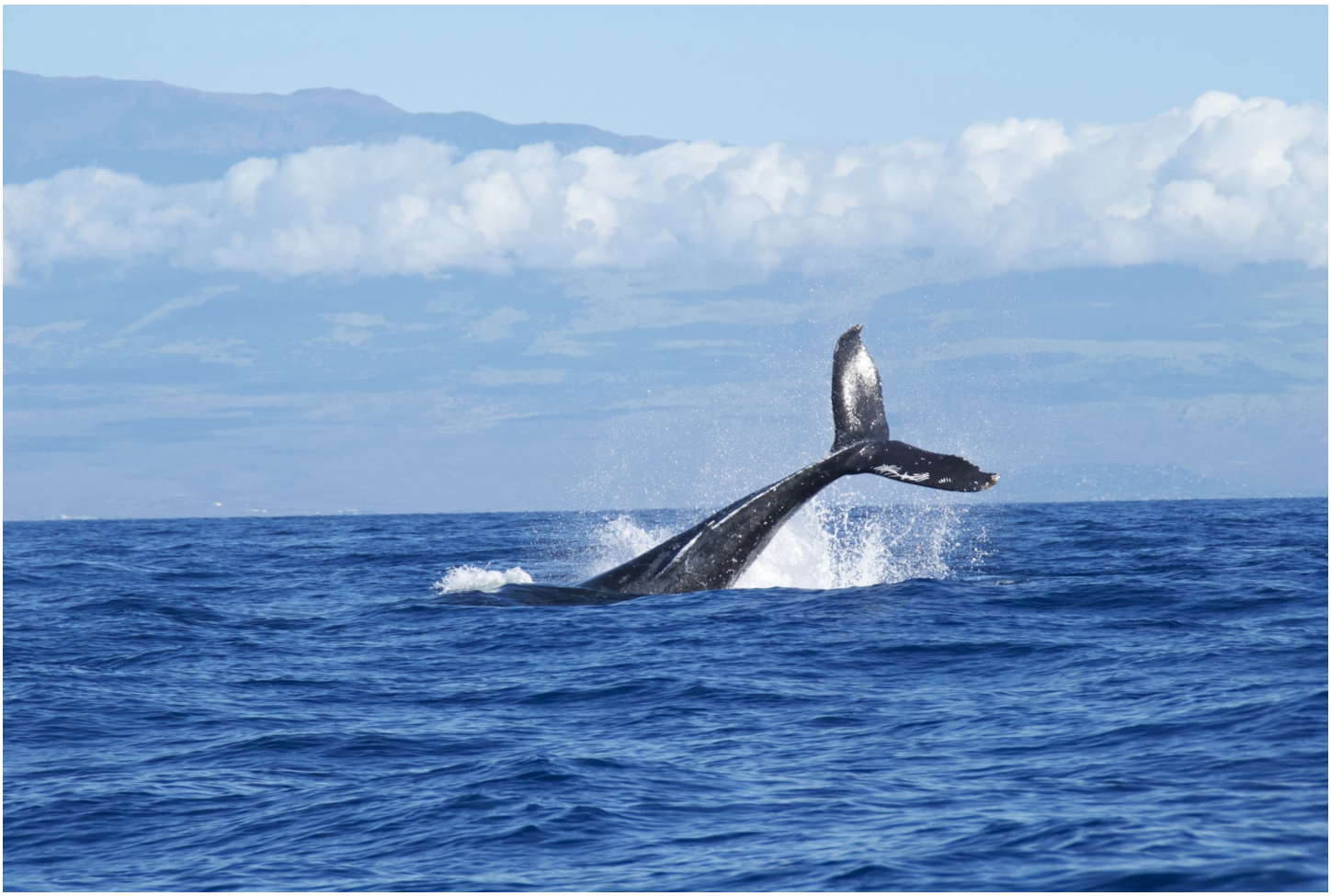


Photo by Abigail Lynn on Unsplash

If you have worked with projects with a certain level of complexity, you have probably heard of **Docker**. In short, this tool allows us to standardize the software used in each project so that it works the same regardless of the machine that hosts it.

That is, what Docker provides us with is a layer of abstraction on the software used through the concept of "container" which simplifies the creation, deployment and execution of our applications. Thanks to these containers we can forget about one of the most recurrent programming problems:

> "It works on my computer"

Since they are packaged the dependencies of our application avoiding having to install them manually every time we change machines.

What we will do in this article will be to learn how to integrate **Docker** into a **Symfony** project, so that we can run it on any computer or server. To do this, it will be enough to have Docker installed where we want to execute the project, something you can do from the following link:

Get Started with Docker | Docker

Learn about the complete container solution provided by Docker. Find information for developers, IT operations, and...

www.docker.com

Done this, let's go for the interesting part!

Folder structure

Before we begin, the first thing we will do will be to propose the folder structure that our project will have. Since we are going to use **Docker**, our project will have the following structure:

```
📦symfony-docker
 ┣ 📂build
 ┣ 📂mysql
 ┣ 📂symfony
 ┗ 📜docker-compose.yml
```

- In the **build** folder you will find the files that we will use to configure each of the Docker containers of our application.
- The database of our project will be stored in the **mysql** folder.
- Inside the **symfony** folder will reside the application that we will write using our favorite PHP framework.
- Finally, the `docker-compose.yml` file will be where we configure the 3 containers that our application will have: that of Nginx, that of PHP and that of MySQL.

For those who do not sound like **Docker Compose**, this utility allows us to define in a file all the containers of which our application is composed and run them later. This configuration is done within a Yaml file located at the root of the project (our already mentioned `docker-compose.yml`). If you want to read more about this tool, I suggest you go to the official documentation since it is perfectly explained there:

Overview of Docker Compose

Looking for Compose file reference? Find the latest version here. Compose is a tool for defining and running...

docs.docker.com

Configuring the container for Nginx

The first thing we will do is configure the first container in which a Nginx-based server will be found.

To do this, we will open our `docker-compose.yml` file and add the following:

```
1 version: '3'
2
3 services:
4     nginx:
5         build:
6             context: .
```

```
7             dockerfile: Dockerfile-nginx
8         volumes:
9             - ./symfony/:/var/www/symfony/
10        ports:
11            - 8001:80
12        networks:
13            - symfony
14
15 networks:
16     symfony:
```

From line 4 is where we will define the Nginx container:

- `dockerfile` indicates the configuration file that will be used to build the container and that we will create next.
- `volumes` will establish that our local folder called `symfony` will be linked to the `/var/www/symfony` folder of the container.
- `ports` performs mapping between port 80 of the container and port **8001** that we will use to access through the browser.
- And finally `networks` establish a network for the entire project so that all containers can communicate with each other.

Next, we will create the **Dockerfile-nginx** file also at the root of our project with the following content:

```
1 FROM nginx:latest
2 COPY ./build/nginx/default.conf /etc/nginx/conf.d/
```

In line 1, we will define the Nginx image we want to use using the FROM directive.

In line 2 we will use the `COPY` method to copy the contents of the `default.conf` file of our project (where the default server configuration that Nginx will use) is located in the `/etc/nginx/conf.d` folder of the container.

This `default.conf` file will be placed in the `/build/nginx` folder and will look like this:

```
server {
    listen 80;
    root /var/www/symfony/public;

    location / {
        try_files $uri /index.php$is_args$args;
    }

    location ~ ^/index\.php(/|$) {
        # Connect to the Docker service using fpm
        fastcgi_pass php:9000;
        fastcgi_split_path_info ^(.+\.php)(/.*)$;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
        fastcgi_param DOCUMENT_ROOT $realpath_root;
        internal;
    }
    location ~ \.php$ {
        return 404;
    }

    error_log /dev/stdout info;
    access_log /var/log/nginx/project_access.log;
}
```

As you can see, on line 3 we have the `root` configuration that points to the `public` folder of our project

Note. Remember that in a Symfony project the server must point not to the root of the project but to the `public` folder of the same.

With all this, we can already test our first container by executing the following command from the root of the project:

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

which will build or lift the only container we have.

Configuring the container for PHP

The next step will be to create the container where the **PHP FPM** service will run, for which we will add the following to our `docker-compose.yml` file:

```yaml
version: '3'

services:
    nginx:
        build:
            context: .
            dockerfile: Dockerfile-nginx
        volumes:
            - ./symfony/:/var/www/symfony/
        ports:
            - 8001:80
        networks:
            - symfony
    php:
        build:
            context: .
            dockerfile: Dockerfile-php
        environment:
            APP_ENV: dev
        volumes:
            - ./symfony/:/var/www/symfony/
        networks:
            - symfony

networks:
    symfony:
```

As you can see, the configuration is similar to the one we use for Nginx except for the following details:

- Within `environment` we can declare the environment variables that we want our Symfony project to read and load later.
- `networks` are also made up of a single element: `symfony` which allows this container and that of Nginx to communicate.

- Finally, `volumes` have the same value as the Nginx container, so that the `symfony` folder of our project is linked to the `/var/www/symfony` folder of the container.

As for the `Dockerfile-php` file it will look like this:

```
FROM php:fpm-stretch

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

RUN apt-get update && apt-get install -y --no-install-recommends \
        git \
        zlib1g-dev \
        libxml2-dev \
        libzip-dev \
    && docker-php-ext-install \
        zip \
        intl \
        mysqli \
        pdo pdo_mysql

RUN curl -sS https://getcomposer.org/installer | php && mv composer.phar /usr/local/bin/composer
COPY symfony/ /var/www/symfony
WORKDIR /var/www/symfony/
```

Through the `docker-php-ext-install` command we can request all the PHP extensions we need. In our case I am only using `zip`, `intl` and those associated with MySQL. The complete list can be seen here:

https://gist.github.com/giansalex/2776a4206666d940d014792ab4700d80

Again you can verify that everything has gone correctly using the following command:

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

Now we will see how two containers are deployed, the Nginx and the new PHP.

Configuring the container for MySQL

Finally, the container for MySQL can be configured by adding the following to our `docker-compose.yml` file:

```yaml
version: '3'

services:
    nginx:
        build:
            context: .
            dockerfile: Dockerfile-nginx
        volumes:
            - ./symfony/:/var/www/symfony/
        ports:
            - 8001:80
        networks:
            - symfony
    php:
        build:
            context: .
            dockerfile: Dockerfile-php
        environment:
            APP_ENV: dev
            DATABASE_URL: mysql://root:root@mysql:3306/symfony_db?serverVersion=5.7
        volumes:
            - ./symfony/:/var/www/symfony/
        networks:
            - symfony
        depends_on:
            - mysql
    mysql:
        image: mysql
        command: ['--character-set-server=utf8mb4', '--collation-server=utf8mb4_unicode_ci','--default-authentication-plugin=mysql_native_password']
        environment:
            MYSQL_ROOT_PASSWORD: root
        ports:
            - 3306:3306
        volumes:
            - ./mysql:/var/lib/mysql
        networks:
            - symfony
networks:
    symfony:

```

In this case we do not need a dockerfile since all the configuration needed by our container for MySQL can be specified within the YAML file. However, there are a few interesting details to comment.

Authentication Plugin

On **line 29** I have added the configuration that allows us to establish which authentication plugin will be used when connecting to MySQL. This is because until PHP 7.4 there is no compatibility with the plugin that MySQL uses to authenticate users, which causes the following failure:

PDO::__construct(): The server requested authentication method unknown to the client [caching_sha2_password]

Reference: https://github.com/laradock/laradock/issues/1390

Database Users

On the other hand, on **line 31** I set the `root` user's password so that we can then use it to authenticate from Doctrine. In the case that we would like to generate a database at the time of creating the container together with a user with all the privileges on it we can opt for the following configuration:

```
    php:
        build:
            context: .
            dockerfile: Dockerfile-php
        environment:
            APP_ENV: dev
            DATABASE_URL: mysql://symfony_user:symfony_password@mysql:3306/symfony_db?serverVersion=5.7
        volumes:
            - ./symfony/:/var/www/symfony/
        networks:
            - symfony
        depends_on:
            - mysql
    mysql:
        image: mysql
        command: ['--character-set-server=utf8mb4', '--collation-server=utf8mb4_unicode_ci','--default-
authentication-plugin=mysql_native_password']
        environment:
            MYSQL_DATABASE: symfony_db
            MYSQL_USER: symfony_user
            MYSQL_PASSWORD: symfony_password
            MYSQL_ROOT_PASSWORD: root
        ports:
            - 3306:3306
        volumes:
            - ./mysql:/var/lib/mysql
        networks:
            - symfony
```

Dependencies between containers

Finally on **line 25** I have added the dependency with the MySQL container to the PHP container so that they are lifted in order.

Deploy the 3 containers

Once we have finished configuring the MySQL container, we will have everything ready, so we will execute the command to build and deploy the 3 containers:

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

And, if everything went well, you should get an output similar to the following:

```
symfony-docker_nginx_1 is up-to-date

symfony-docker_php_1 is up-to-date

Creating symfony-docker_mysql_1 ... done
```

Installing Symfony

Since from now on we will work with our containers, the installation of Symfony will be carried out within them and not from our operating system.

Therefore, the first thing we will have to do is access our container where PHP is located (remember to have picked it up) using the command:

```
docker exec -it  symfony-docker_php_1 bash
```

Where **`symfony-docker_php_1`** is the name that the `docker-compose up` command returned to you after you finished launching the containers.

Now inside the container we can install Symfony as the documentation suggests:

```
curl -sS https://get.symfony.com/cli/installer | bash

mv /root/.symfony/bin/symfony /usr/local/bin/symfony

symfony new symfony --dir=/var/www/symfony
```

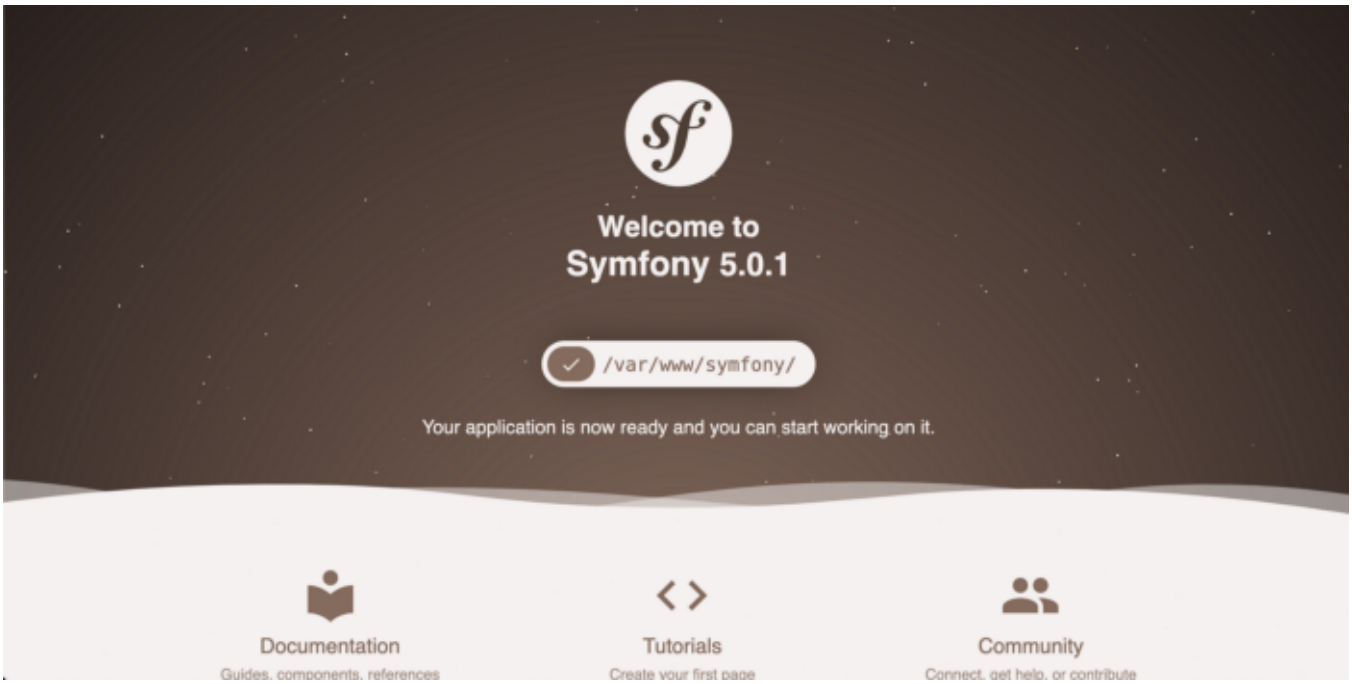
Note. It is possible that the creation of the project of the following fault:

```
*** Please tell me who you are.

Run

git config --global user.email "you@example.com"

git config --global user.name "Your Name"
```

```
to set your account's default identity
```

Adding the Git configuration requested, it will be resolved without further complications

After installing Symfony we can verify that everything has worked correctly by accessing localhost: 8001 (the port is the one we map in Nginx within the `docker-compose.yml` file) which will show us the Symfony welcome page:



If we now want to use Doctrine to work with a database, it would be enough to install its corresponding bundle using composer / flex in the usual way (always from within our container):

```
composer require symfony/orm-pack
composer require --dev symfony/maker-bundle
```

And add the `DATABASE_URL` environment variable into the PHP container configuration:

```
DATABASE_URL:
mysql://symfony_user:symfony_password@mysql:3306/symfony_db?
serverVersion=5.7
```

If you realize, the host of the database is the name of our container (`mysql:3306`) and not the IP 127.0.0.1. That is, if for example our container is named as a `database` within our `docker-compose.yml` file, the environment variable to connect to that database would have the value:

```
DATABASE_URL:
mysql://symfony_user:symfony_password@database:3306/symfony_db?
serverVersion=5.7
```

Once this is done, we will restart the container to pick up this change in the PHP environment variables:

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

And we can work with our database normally. For example, within our PHP container we can launch the command:

```
bin/console doctrine:database:create
```

if we want to verify that the project connects correctly to the database that we have defined

Repository

If you want to access the full code of this article you can do it from here:

ger86/symfony-docker

You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or...

github.com

Final thoughts

After having dockerized the project, we can work with it on any machine without worrying about library dependencies or the operating system used.

This for medium / large-sized projects where several developers are also involved is very useful since all of them will work on the same versions which will prevent us from having failures to use different versions in the libraries that we require.

In addition, it will be enough to install Docker on the machine where we want to work to be able to continue developing the project without forcing us to install the necessary libraries one by one.

Do you want to read more articles like this?

If you liked this article I encourage you to subscribe to the newsletter that I send every Sunday with similar publications to this and more recommended content: 👇👇👇

Latte and Code

Estás a punto de suscribirte a la newsletter de Latte and Code que recibirás cada domingo. En ella podrás encontrar: ...

eepurl.us20.list-manage.com

Docker Docker Compose Symfony Symfony4 Symfony5