

Server-side Web Development

Unit 10. Composer. PHP Frameworks. Symfony.

Fidel Oltra, Ricardo Sánchez



IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2024-25

Index

1	Composer	2
1.1	Installing Composer on Linux	2
1.2	Installing Composer on Windows	3
1.3	Packagist	3
1.4	Composer first steps	4
1.4.1	require	4
1.4.2	install	4
1.4.3	update	5
1.4.4	Another composer useful commands	5
1.5	Composer and version control	5
1.6	Autoloading	6
1.7	Creating projects	6
2	PHP Frameworks	7
2.1	Benefits of using a PHP Framework	7
2.2	Frameworks and micro-frameworks	7
2.3	Some PHP frameworks	8
3	Symfony	8
3.1	Installing Symfony	9
3.1.1	Work environment	9
3.1.2	Windows installation	9
3.1.3	Linux installation	9
3.2	Our first Symfony project	11
3.3	.env file	14
3.4	Customizing our first web application	15

1 Composer

Composer is a dependency manager for PHP that allows you to install different modules or libraries in a project. It contains an online database, [Packagist.com](https://packagist.org), with many centralized available libraries, with which we can indicate which one(s) we want for each specific project, and Composer downloads and installs them for us. It is something very similar to other managers such as NPM (Node Package Manager), for Node or Javascript in general.

Composer can be installed locally for each web project, or globally for the entire system. This last option is recommended if we want to manage several projects on our computer, so as not to have to install it on all of them.

1.1 Installing Composer on Linux

To install Composer on Ubuntu, first of all install the required dependencies:

```
sudo apt update
sudo apt install php-cli unzip
```

Next, install Composer with these commands:

```
php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');"
php -r "if (hash_file('sha384', 'composer-setup.php') ===
→ 'dac665fdc30fdd8ec78b38b9800061b4150413ff2e3b6f88543c636f7cd84f6db9189d43a81e5503cda
→ { echo 'Installer verified'; } else { echo 'Installer corrupt';
→ unlink('composer-setup.php'); } echo PHP_EOL;"
php composer-setup.php
php -r "unlink('composer-setup.php');"
```

This installer script will simply check some `php.ini` settings, warn you if they are set incorrectly, and then download the latest `composer.phar` in the current directory. The 4 lines above will, in order:

- Download the installer to the current directory
- Verify the installer SHA-384, which you can also cross-check [here](https://getcomposer.org/installer)
- Run the installer
- Remove the installer

Next, move the `composer.phar` file into a directory on your `PATH`, so you can call `composer` from any directory:

```
sudo mv composer.phar /usr/local/bin/composer
```

You can check your installation with:

```
composer -V
```

The complete installation guide:

- <https://getcomposer.org/download/>

1.2 Installing Composer on Windows

Using the Installer is the easiest way to get Composer set up on your machine.

Download and run [Composer-Setup.exe](#). It will install the latest Composer version and set up your PATH so that you can call composer from any directory in your command line.

1.3 Packagist

Packagist is Composer's default package repository, from which Composer pulls libraries and their dependencies when you ask it to install a specific library. There are hundreds of libraries available on Packagist. If you need a feature in your PHP projects that you think should already be available as a 3rd party library, Packagist is the first place to look.

You can search directly in the Packagist web page, packagist.org, or search in the command line. For instance:

```
composer search monolog
```

returns the result in the format **vendor/package**:

```
monolog/monolog           Sends your logs to files, sockets...
symfony/monolog-bundle     Symfony MonologBundle
symfony/monolog-bridge     Provides integration for Monolog ...
symfony/debug-bundle       Provides a tight integration of the
↳ Symfony VarDumper component...
maxbanton/cwh              AWS CloudWatch Handler for Monolog
↳ library
...
```

1.4 Composer first steps

The common procedure is to create a directory for your project. The next composer commands must be written inside the main folder of your project.

1.4.1 require

The composer **require** command is a shortcut for the process of creating the dependency files for your project. `require` will create a **composer.json** file (if it does not exist) and add the required package to it automatically. It also creates and/or updates the **composer.lock** file, which is used to write the package's information with the exact versions. And finally, it installs the required package and dependencies in the **vendor** directory.

The following command shows how to install the monolog package (a library used to create logs) with `require`. Write that inside your project directory:

```
composer require monolog/monolog
```

The content of the `composer.json` file is:

```
{
  "require": {
    "monolog/monolog": "^3.7"
  }
}
```

1.4.2 install

This command installs the project dependencies from the `composer.lock` file if present, or falls back on the `composer.json`.

To use `install`, you first need to create the `composer.json` file in your project. Alternatively to the initialization with `require`, you can write your own `composer.json` file and then run the `composer install` command to download the package and all its dependencies:

```
composer install
```

For instance, try to delete your `vendor` dir and then run `composer install`. You'll see how the `vendor` directory will be created again.

1.4.3 update

The `composer.lock` file prevents you from automatically getting the latest versions of your dependencies. It helps to maintain the integrity of the project so forces all the members of the developers team to have the same versions.

To update to the latest versions, use the `update` command. This will fetch the latest matching versions (according to your `composer.json` file) and update the lock file with the new versions.

The command:

```
composer update
```

updates all the packages.

If you want to update only a package, write:

```
composer update vendor/package
```

1.4.4 Another composer useful commands

- **composer list:** List the Composer commands.
- **composer self-update:** Updates Composer to the latest version.
- **composer show:** Shows information about packages (`composer show` inside a project or `composer show vendor/package`)
- **composer status:** Shows a list of locally modified packages.
- **composer remove:** Removes a package from the project.

1.5 Composer and version control

If you use a version control, like **Git**, you need to add your `composer.json` and `composer.lock` files to the repo, but you don't have to add the `vendor` folder, because it stores a large amount of data that can be recreated again with `composer install`.

Committing `composer.lock` to version control is important because it will cause anyone who sets up the project to use the exact same versions of the dependencies that you are using. That mitigates the potential for bugs affecting only some parts of the deployments. Even if you develop alone, in six months when reinstalling the project you can feel confident the dependencies installed are still working even if your dependencies released many new versions since then.

The `.gitignore` file of your project should have the next line:

vendor/

To clone a PHP project with composer, first clone the project and then, enter to the project's folder and run composer install:

```
git clone project-repo-url
cd project-dir
composer install
```

1.6 Autoloading

For libraries that specify **autoload** information, Composer generates a `vendor/autoload.php` file. You can include this file and start using the classes that those libraries provide without any extra work:

```
require __DIR__ . '/../vendor/autoload.php';

$log = new Monolog\Logger('name');
$log->pushHandler( new Monolog\Handler\StreamHandler('app.log',
    ↪ Monolog\Logger::WARNING));
$log->warning('warning message');
```

1.7 Creating projects

You can use Composer to create new projects from an existing package. This is the equivalent of doing a `git clone` followed by a `composer install` of the vendors.

To create a new project using Composer you can use the `create-project` command. Pass it a package name, and the directory to create the project in. You can also provide a version as a third argument, otherwise the latest version is used.

If the directory does not currently exist, it will be created during installation.

Some examples:

```
# Creates a Doctrine project, with the 2.2 version
composer create-project doctrine/orm my-doctrine-project "2.2.*"
```

```
# Creates a Laravel project
composer create-project laravel/laravel my-laravel-project

# Creates a Symfony micro-service project
composer create-project symfony/skeleton my-symfony-microservice
```

Composer documentation: <https://getcomposer.org/doc/>

2 PHP Frameworks

At the moment we can write a PHP application from scratch, creating our own classes, views, validators and functions to access the database. Therefore, we have tools to help us in the process: the **frameworks**. What is a framework?

A **framework** is a tool that provides the basic structure and the main components to develop an application.

PHP frameworks provide tools, commands, built-in functions, libraries and other reusable components to make the web development easier. Most PHP frameworks help us to create projects with the **MVC** (*Model-View-Controller*) software design pattern. Because of its modular structure, the resulting project is very easy to maintain.

2.1 Benefits of using a PHP Framework

A PHP framework speeds up the development and helps us increase productivity. These are some of the benefits of using a PHP framework:

- Using a framework saves development time
- Makes easier to cooperate with other developers
- Helps to keep the code more organized
- Most PHP frameworks provide a lot of extensive and clear documentation
- Frameworks usually provide safe code in order to minimize risks like cross-site scripting or SQL injection

2.2 Frameworks and micro-frameworks

A full-stack PHP framework helps developers with the entire developing process, from the user interface to the access to the database. However, we can use micro-frameworks to develop small applications

with low complexity and a specific use.

2.3 Some PHP frameworks

Below you can see a list of some PHP frameworks. All of them are open-source:

- Laravel
- CodeIgniter
- Symfony
- Laminas (continuation of Zend)
- Phalcon
- CakePHP
- FuelPHP

Some PHP microframeworks:

- Slim
- Fat-free Framework (F3)

In this course we will use **Symfony**.

3 Symfony

Symfony has a lot of features that make it a very attractive option. Some of them are:

- modular component system
- a [large developers community](#)
- very oriented to MVC design pattern
- a fast template engine (*Twig*)
- a built-in debugging tool
- interaction with third-party libraries (*bundles*)
- SymfonyCloud support (optional)
- many helpful learning resources and extensive documentation

As of today, the last Symfony version is the 7.1, but the next version, 7.2, is planned to be released in november 2025.

For this course we will use the **Symfony version 7.1**, which requires at least **PHP 8.2**.

[Symfony official page](#)

[Official Symfony book \(downloadable by paying\)](#)

3.1 Installing Symfony

3.1.1 Work environment

Symfony runs on any popular OS (macOS, Windows, Linux).

In **Windows**, as work environment, we can keep using XAMPP. Our DBMS will be MariaDB/MySQL, and our web server will be Apache. For Symfony 6.1 to 6.4, we will need to check that our PHP version is 8.1 or higher. And for Symfony 7.1 we need PHP 8.2 or higher.

In Linux we will continue with the existent configuration: **PHP 8.2** (recommended version), **Apache** and **MySQL** or **MariaDB**.

Let's assume you've already installed **Composer**.

3.1.2 Windows installation

Let's assume we have **Xampp** installed. We will work with Apache later. Check that our Xampp version includes the PHP version that we need.

We can get Symfony CLI from [the official web](#):

We need Scoop (a command-line installer for Windows) ([install Scoop](#)). In a PowerShell terminal, run:

```
> Set-ExecutionPolicy RemoteSigned -Scope CurrentUser  
> Invoke-RestMethod -Uri https://get.scoop.sh | Invoke-Expression
```

And then, in a terminal:

```
scoop install symfony-cli
```

3.1.3 Linux installation

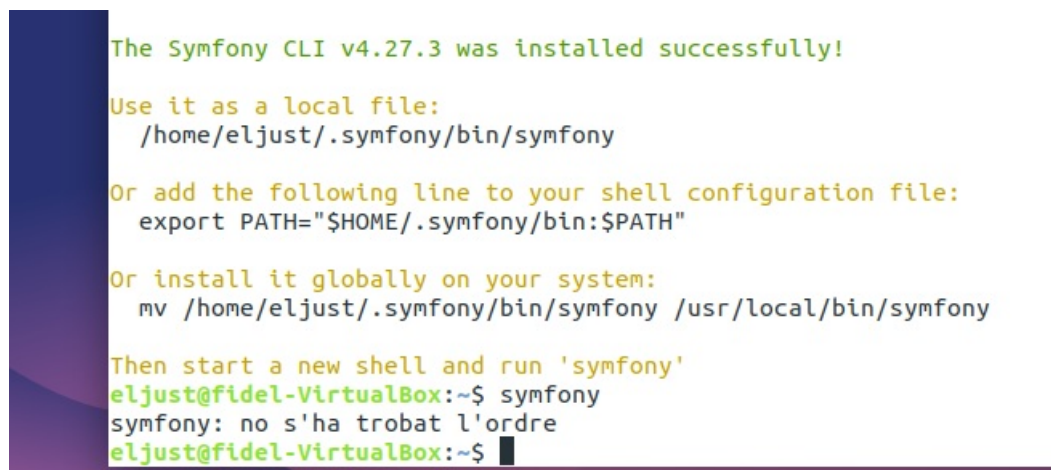
We can also install Symfony CLI in Linux from our command line by doing:

```
wget https://get.symfony.com/cli/installer -O - | bash
```

or

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

Once installed, we need to move Symfony under our `$PATH` in order to have it available from our project folders. As you can see in the figure, we can't execute Symfony from everywhere yet.



```
The Symfony CLI v4.27.3 was installed successfully!

Use it as a local file:
/home/eljust/.symfony/bin/symfony

Or add the following line to your shell configuration file:
export PATH="$HOME/.symfony/bin:$PATH"

Or install it globally on your system:
mv /home/eljust/.symfony/bin/symfony /usr/local/bin/symfony

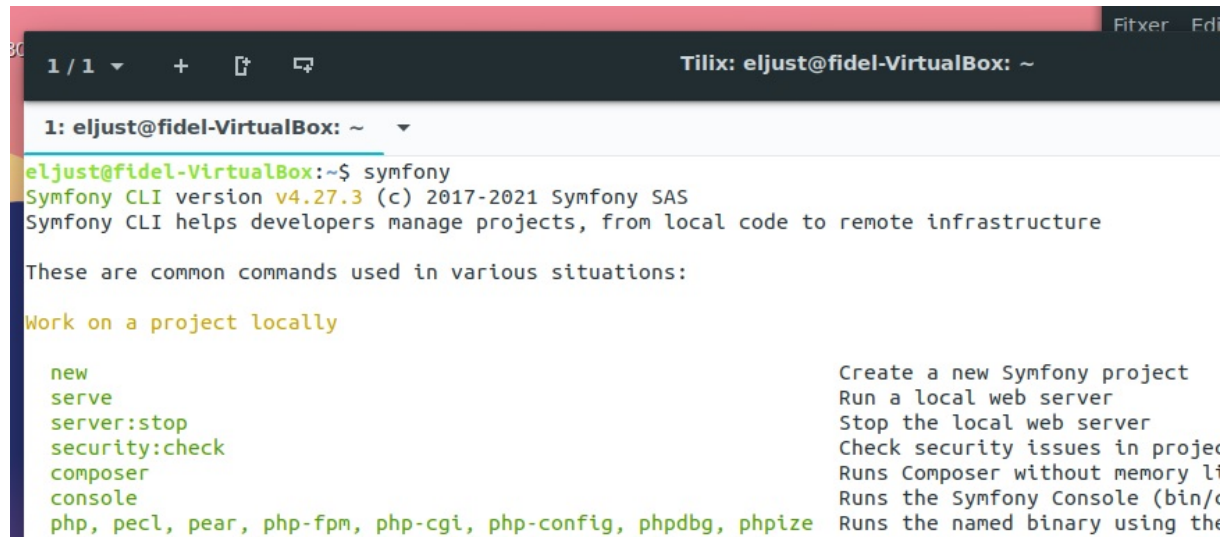
Then start a new shell and run 'symfony'
eljust@fidel-VirtualBox:~$ symfony
symfony: no s'ha trobat l'ordre
eljust@fidel-VirtualBox:~$
```

Figure 1: Symfony installed

So, we move the executable file to a folder included in our `$PATH`, for instance `/usr/local/bin`

```
sudo mv .symfony5/bin/symfony /usr/local/bin/symfony
```

Let's try if it works:



```

Tilix: eljust@fidel-VirtualBox: ~
1: eljust@fidel-VirtualBox: ~
eljust@fidel-VirtualBox:~$ symfony
Symfony CLI version v4.27.3 (c) 2017-2021 Symfony SAS
Symfony CLI helps developers manage projects, from local code to remote infrastructure

These are common commands used in various situations:

Work on a project locally

new          Create a new Symfony project
serve        Run a local web server
server:stop  Stop the local web server
security:check Check security issues in project
composer     Runs Composer without memory limit
console      Runs the Symfony Console (bin/console)
php, pecl, pear, php-fpm, php-cgi, php-config, phpdbg, phpize Runs the named binary using the
  
```

Figure 2: Symfony CLI is working

Once you have installed the *symfony-cli* tool, run the next command to check if you have the required extensions:

```
symfony check:requirements
```

Then, install the missing dependencies, if needed.

Alternatively, you can run the **php client** (*php-cli*) command `php -m` to check the PHP extensions currently enabled. If an important extension is missing, we can install it with `apt`. The PHP extensions that we need are: *intl*, *pdo_pgsql*, *xsl*, *amqp*, *gd*, *openssl*, *sodium*, *redis* and *curl*.

Complete installation instructions:

- [Installing & Setting up the Symfony Framework](#)
- [Download Symfony](#)

3.2 Our first Symfony project

To create a Symfony project, move to a selected folder which will contain all our projects, and then open the console terminal and run one of these commands:

To create a complete web application

```
symfony new project_name --webapp
```

To build a microservice, API or console app:

```
symfony new project_name
```

With Composer, to create a full app:

```
composer create-project symfony/skeleton my_project_name  
cd my_project_name  
composer require webapp
```

or, to build a micro-service:

```
composer create-project symfony/skeleton project_name
```

Let's create a new web project called **prova** in our home folder:

```
symfony new prova --webapp
```

It will take from seconds to minutes to create the full project structure. When the process is completed, we can go to the **prova** folder and list the contents to see the whole project structure:

Let's take a look to the more important sections of our project:

- **bin**: this folder contains the main CLI entry point, the console command, among other useful instructions
- **config**: contains some configuration files
- **public**: is the web root directory which will contain our **index.php** file and other static contents (mostly client content like CSS or javascript files)
- **src**: our php code
- **templates**: our views (*Twig* templates)
- **tests**: our tests
- **translations**: for translation services
- **var**: this folder contains caches, logs and files generated at runtime (this directory needs to be writable in production)

```

eljust@fidel-VirtualBox:~$ cd prova
eljust@fidel-VirtualBox:~/prova$ ls -l
total 380
drwxrwxr-x 2 eljust eljust 4096 de des. 30 19:42 bin
-rw-rw-r-- 1 eljust eljust 3101 de des. 30 19:42 composer.json
-rw-rw-r-- 1 eljust eljust 313746 de des. 30 19:42 composer.lock
drwxrwxr-x 4 eljust eljust 4096 de des. 30 19:41 config
-rw-rw-r-- 1 eljust eljust 247 de des. 30 19:42 docker-compose.override.yml
-rw-rw-r-- 1 eljust eljust 717 de des. 30 19:42 docker-compose.yml
drwxrwxr-x 2 eljust eljust 4096 de des. 30 19:42 migrations
-rw-rw-r-- 1 eljust eljust 1367 de des. 30 19:42 phpunit.xml.dist
drwxrwxr-x 2 eljust eljust 4096 de des. 30 19:41 public
drwxrwxr-x 5 eljust eljust 4096 de des. 30 19:42 src
-rw-rw-r-- 1 eljust eljust 12908 de des. 30 19:42 symfony.lock
drwxrwxr-x 2 eljust eljust 4096 de des. 30 19:41 templates
drwxrwxr-x 2 eljust eljust 4096 de des. 30 19:42 tests
drwxrwxr-x 4 eljust eljust 4096 de des. 30 19:42 translations
drwxrwxr-x 2 eljust eljust 4096 de des. 30 19:42 var
drwxrwxr-x 17 eljust eljust 4096 de des. 30 19:41 vendor
eljust@fidel-VirtualBox:~/prova$

```

Figure 3: Our project structure

- **vendor:** this folder contains all packages and libraries installed by **Composer**

By now, we can work with a Symfony local server to check our web applications. To run the server we write this command (the *-d* option is for running the server in the background) from the folder where the project has been created (in this case, **prova**):

```
symfony server:start -d
```

Now let's go to our browser and write the URL *localhost:8000*

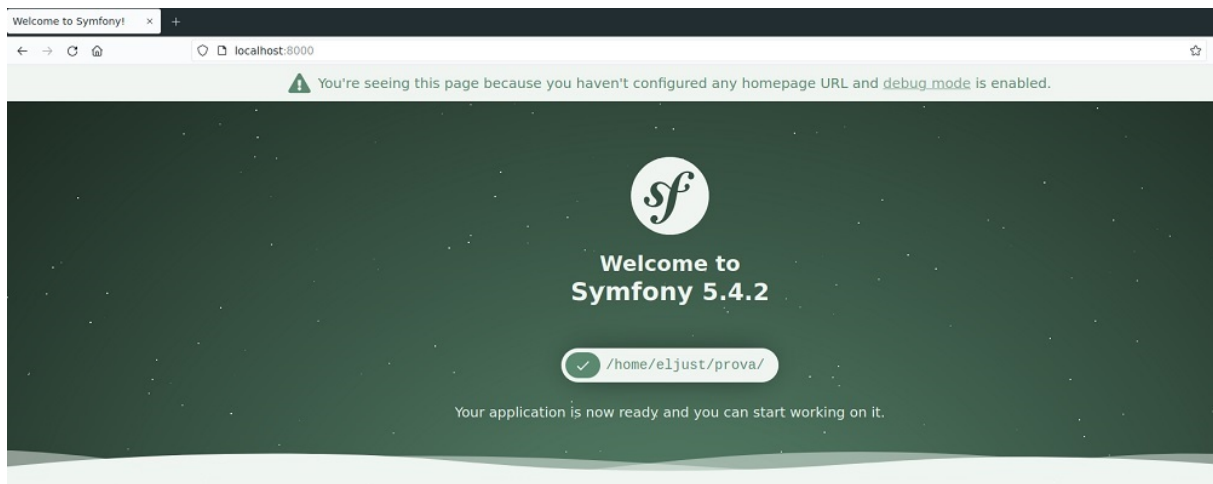


Figure 4: Welcome page

It works!

3.3 .env file

Symfony applications come with a file called **.env** located at the project root directory. This file is used to define the value of environment variables. To apply a particular configuration to our development environment and override environment values, we should make a copy of the file, name it **.env.local**, and then assign or override the values for our local machine.

```
# .env.local
DATABASE_URL="mysql://user:@127.0.0.1:3306/my_database_name"
```

A Symfony application can run in three environments: dev (for local development), prod (for production servers) and test (for automated tests). In the **.env** file, a variable named APP_ENV define the environment we are working on.

```
# .env (or .env.local)
APP_ENV=dev
```

Again, we can create an **.env** file for any environment and override general variables. This file should be named **.env.environment** where **environment** must be **dev**, **prod** or **test**.

3.4 Customizing our first web application

Let's open our **prova** folder with Visual Studio Code. Take a look to the left section that shows our project structure:

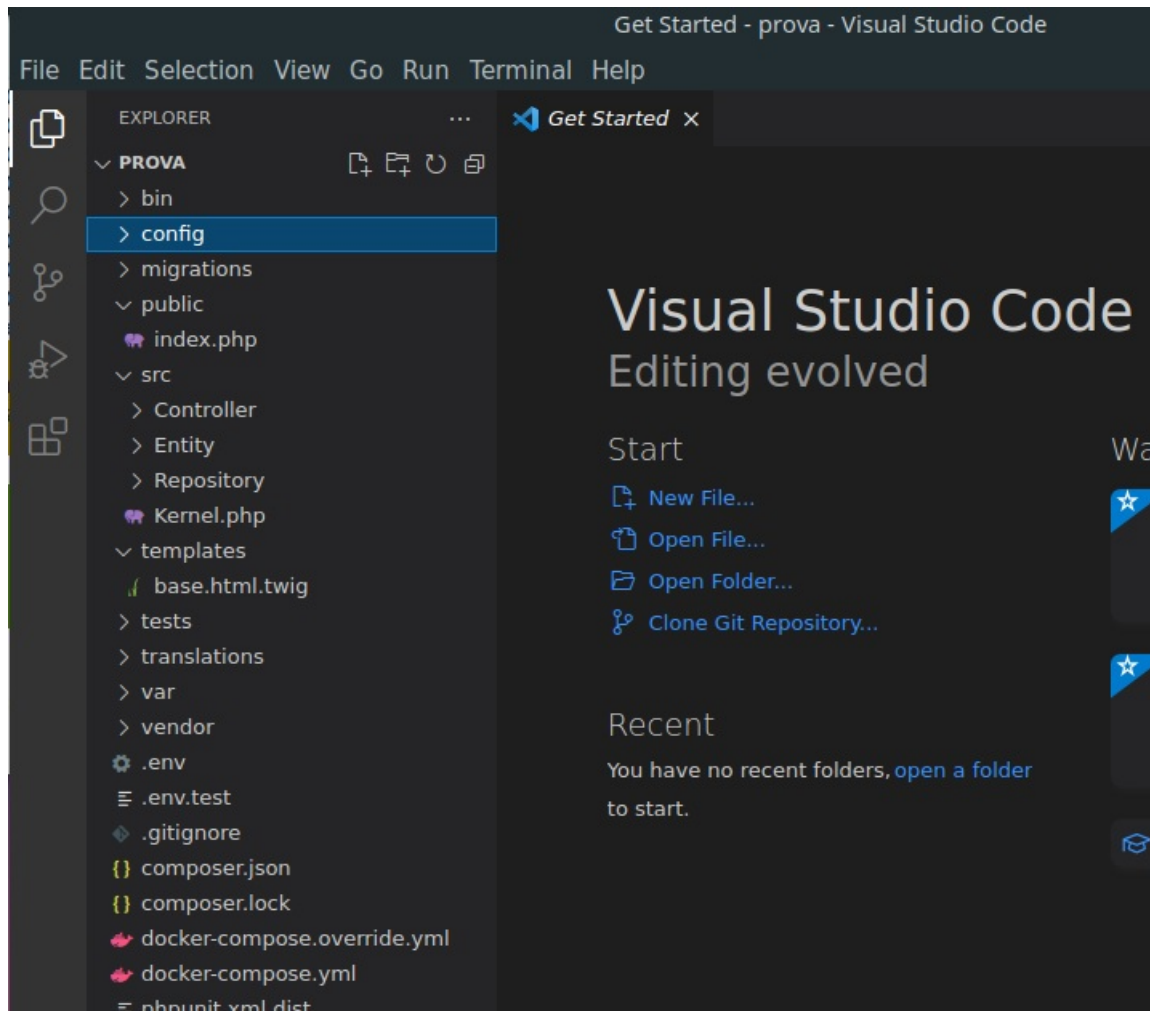


Figure 5: Our project in Visual Studio Code

In the **public** section we only have one file: *index.php*. This is the main entry to our web application. You can see some other folders too, like **src/Controller**, **src/Entity** and **templates**. We will come back to these sections in the next chapters of the unit. The main thing at the moment is to see the content of the *index.php* file. It might be something like this:

```
<?php  
  
use App\Kernel;
```



```
require_once dirname(__DIR__) . '/vendor/autoload_runtime.php';

return function (array $context) {
    return new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);
};
```

We don't need to understand what this file is doing. The only thing we need to know by now is that *index.php* is, at this moment, the main entry of our application. We can change the content of the file to show some kind of welcome message.

For instance, let's copy an *in construction* image to a new folder **public/images**

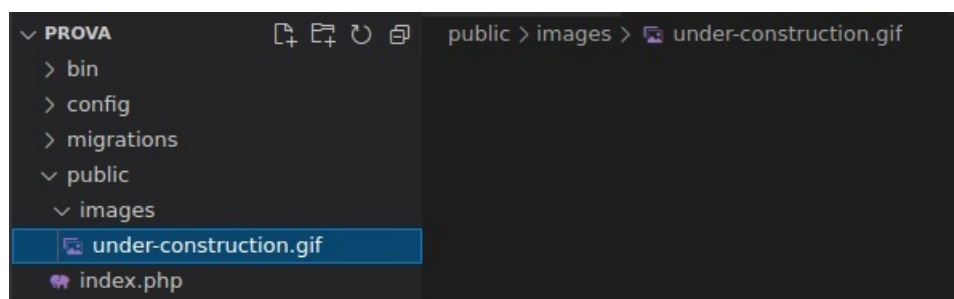


Figure 6: Image in public/images folder

and now let's change the *index.php* this way:

```
<?php

use App\Kernel;

require_once dirname(__DIR__) . '/vendor/autoload_runtime.php';

return function (array $context) {
    echo "<div style='text-align:center;'>";
    echo "<h3>Welcome to my web</h3>";
    echo "<img src='images/under-construction.gif' />";
    echo "</div>";
    //return new Kernel($context['APP_ENV'], (bool) $context['APP_DEBUG']);
};
```

Let's refresh the page in our browser and see what happens:



Figure 7: Our index.php customized page