

Laravel Handbook

Flavio Copes

Updated July 31, 2026

Contents

Preface	2
1. Introduction to Laravel	2
2. Getting started	2
3. Blade	13
4. Dynamic routes	15
5. Adding a database	16
6. How to use migrations to create and modify the database schema	17
7. Using forms to accept user input and store it into the database	21
8. Adding a better layout	29
9. Adding the form at the bottom of the list	35
10. Allow users to delete dogs from the list	38
11. Adding authentication using Laravel Breeze	40
12. Only authenticated users can add items to the database	49
13. Push the app code to GitHub	59
14. Deployment	60
15. Dynamic routes	83
16. Non-web routes	84
17. Creating commands	85
18. Where to go from here	89
Conclusion	89

Preface

The Laravel Handbook follows the 80/20 rule: learn in 20% of the time the 80% of a topic.

In particular, the goal is to get you up to speed quickly with Laravel.

This book is written by Flavio. I **publish programming tutorials** on my blog flaviocopes.com and I organize a yearly bootcamp at bootcamp.dev.

You can reach me on Twitter [@flaviocopes](https://twitter.com/flaviocopes).

Enjoy!

1. Introduction to Laravel

Laravel is one of those legendary frameworks that everyone using it loves.

To me, it's in the same level of Rails and Django.

If you know/prefer Ruby you use Rails.

If you know/prefer Python you use Django.

If you know/prefer PHP you use Laravel.

Generally speaking I mean, because each of those languages has a ton of alternatives.

I would say Laravel, together with WordPress, is the “PHP killer app”.

PHP is often disregarded by developers, but it has some unique features that make it a great language for Web Development and Laravel figured out how to take advantage of the best features of PHP.

On Twitter I can only see love for Laravel.

Much like how it happens for Rails.

This is not a “new framework of the month” kind of thing. Laravel has been around since 2011, well before modern frontend tools like React and Vue.js existed.

It stood the test of time. And it evolved over the years to a serious and complete solution for building Web Applications, which comes out of the box complete with everything you need.

Something like Laravel does not exist in pure JavaScript tooling.

Things like Next.js or Remix appear very primitive in some aspects, while in some other aspects they seem more modern.

It's just a different tool.

And I think as Web Developers we must know in which scenario one tool is more optimal than others. So we can make the best technical choice depending on the requirements.

In this handbook I am going to give a introduction to Laravel to get you up and running.

2. Getting started

To get started with Laravel, you need to set up your PHP environment on your computer.

You can do this in various ways.

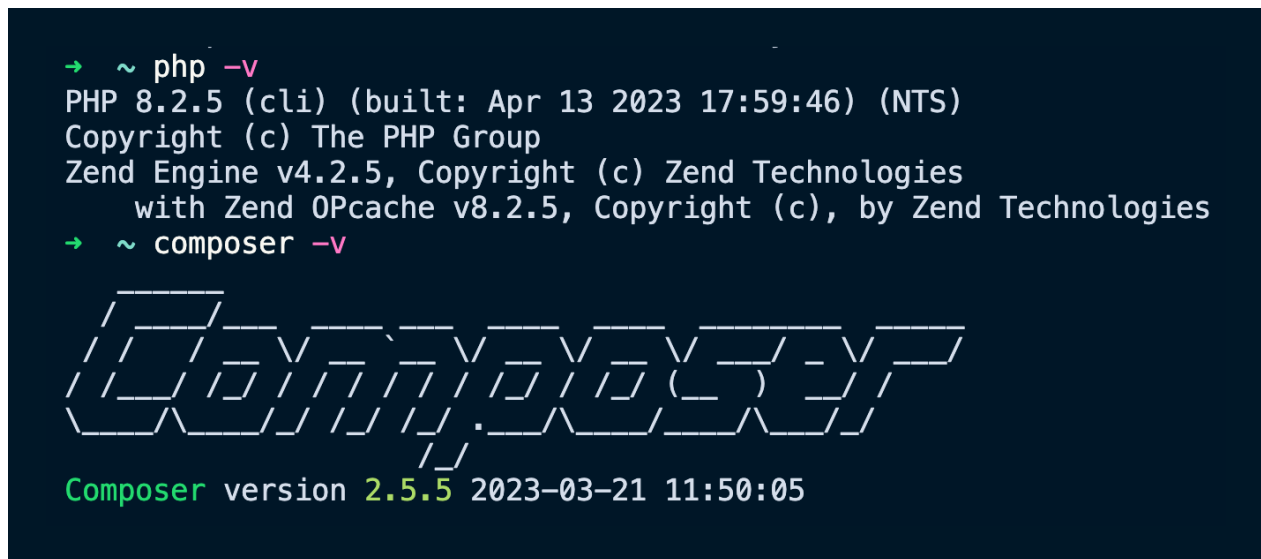
Before going on, remove any older PHP installations you might have done in the past. How exactly depends on how you installed PHP on your machine. Hopefully you haven't any and we can go on.

On macOS, use Homebrew ([install Homebrew](#) first if you haven't already) and install both PHP and [Composer](#) using

```
brew install php composer
```

(might take a while)

Once installed you should be able to run the `php -v` command to get the version of PHP installed (same for `composer -v`):

A terminal window with a dark blue background. It shows the output of the commands 'php -v' and 'composer -v'. The PHP output shows version 8.2.5 (cli) built on April 13, 2023, at 17:59:46, NTS, with copyright for The PHP Group and Zend Engine v4.2.5. The Composer output shows version 2.5.5, dated 2023-03-21 at 11:50:05. The Laravel logo is displayed in a stylized, outlined font in the center of the terminal output.

```
→ ~ php -v
PHP 8.2.5 (cli) (built: Apr 13 2023 17:59:46) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.2.5, Copyright (c) Zend Technologies
    with Zend OPcache v8.2.5, Copyright (c), by Zend Technologies
→ ~ composer -v

Laravel

Composer version 2.5.5 2023-03-21 11:50:05
```

Now you can go into folder on your computer that you reserve for development. I have a `dev` folder in my home directory, for example.

In there, run:

```
composer create-project laravel/laravel first
```

To create a new project in a folder called `first`.

Now go into that folder and run `php artisan serve`:

```
cd first
php artisan serve
```

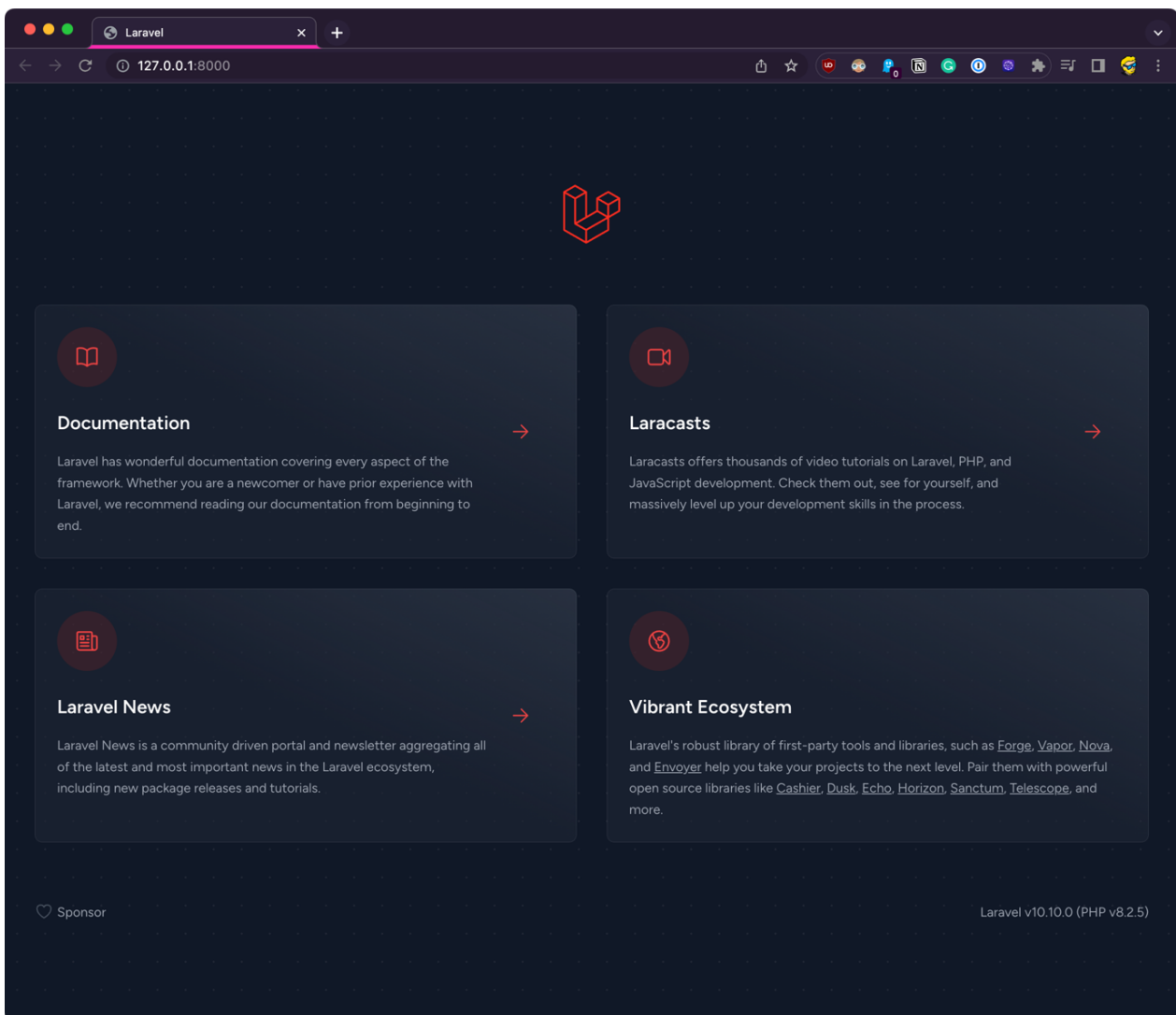
```
→ dev cd first
→ first php artisan serve
```

INFO Server running on [http://127.0.0.1:8000].

Press Ctrl+C to stop the server

`php artisan <some_command>` is something you'll use often in Laravel, as it can do a lot of useful stuff for you. For example we'll use it to "scaffold" models without having to create files by hand.

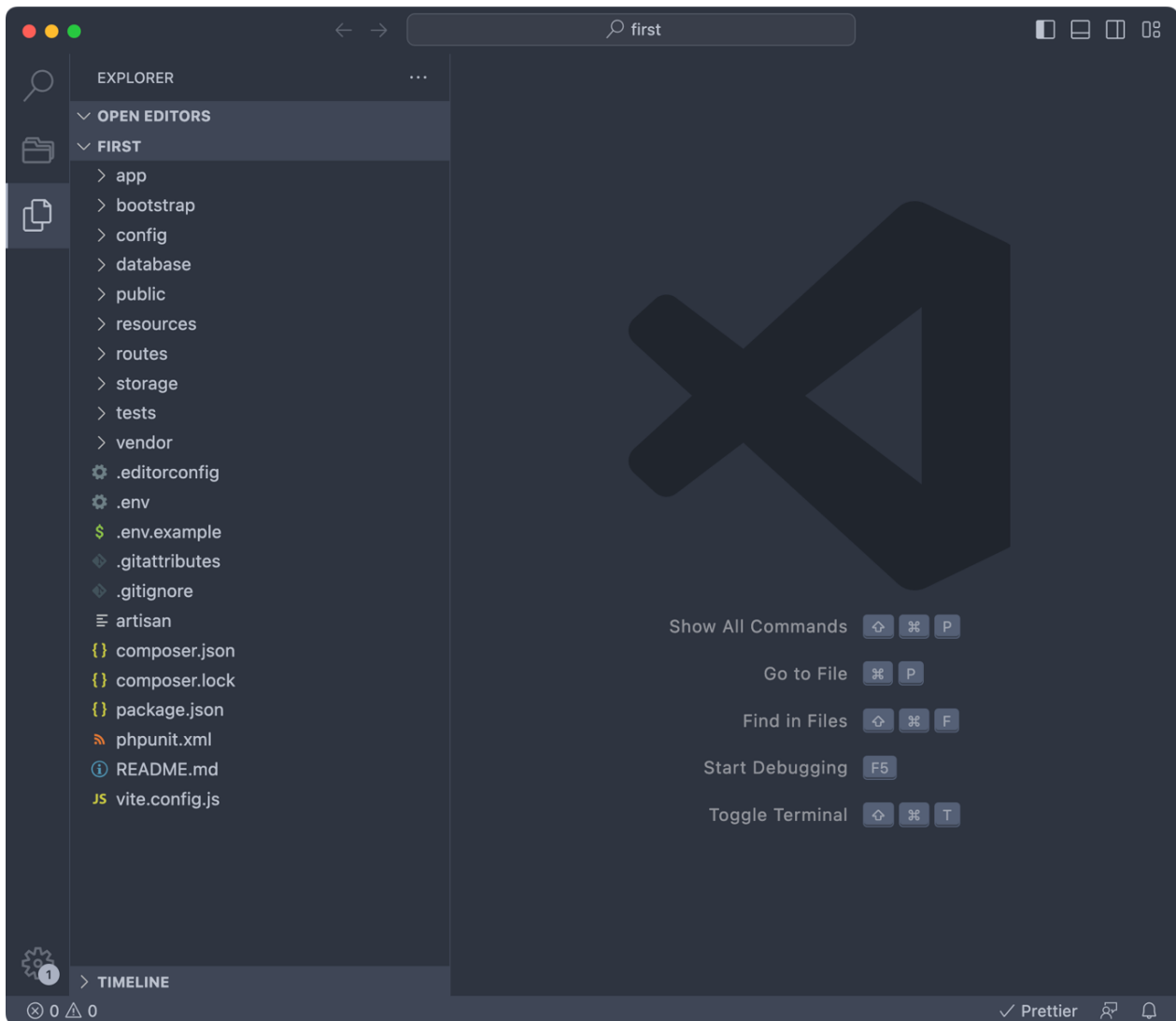
Open your browser and you'll see the default welcome screen of a Laravel app:



If you have troubles reaching this stage, the official documentation has great guides for [macOS](#), [Linux](#) and [Windows](#).

Open the newly created project folder in VS Code.

This should be the file structure:



While you're here I recommend you install the extensions

- [Laravel Extra Intellisense](#)
- [Laravel Artisan](#)
- [Laravel Blade Snippets](#)
- [PHP tools for VS Code](#)

We have a bunch of folders and a bunch of files.

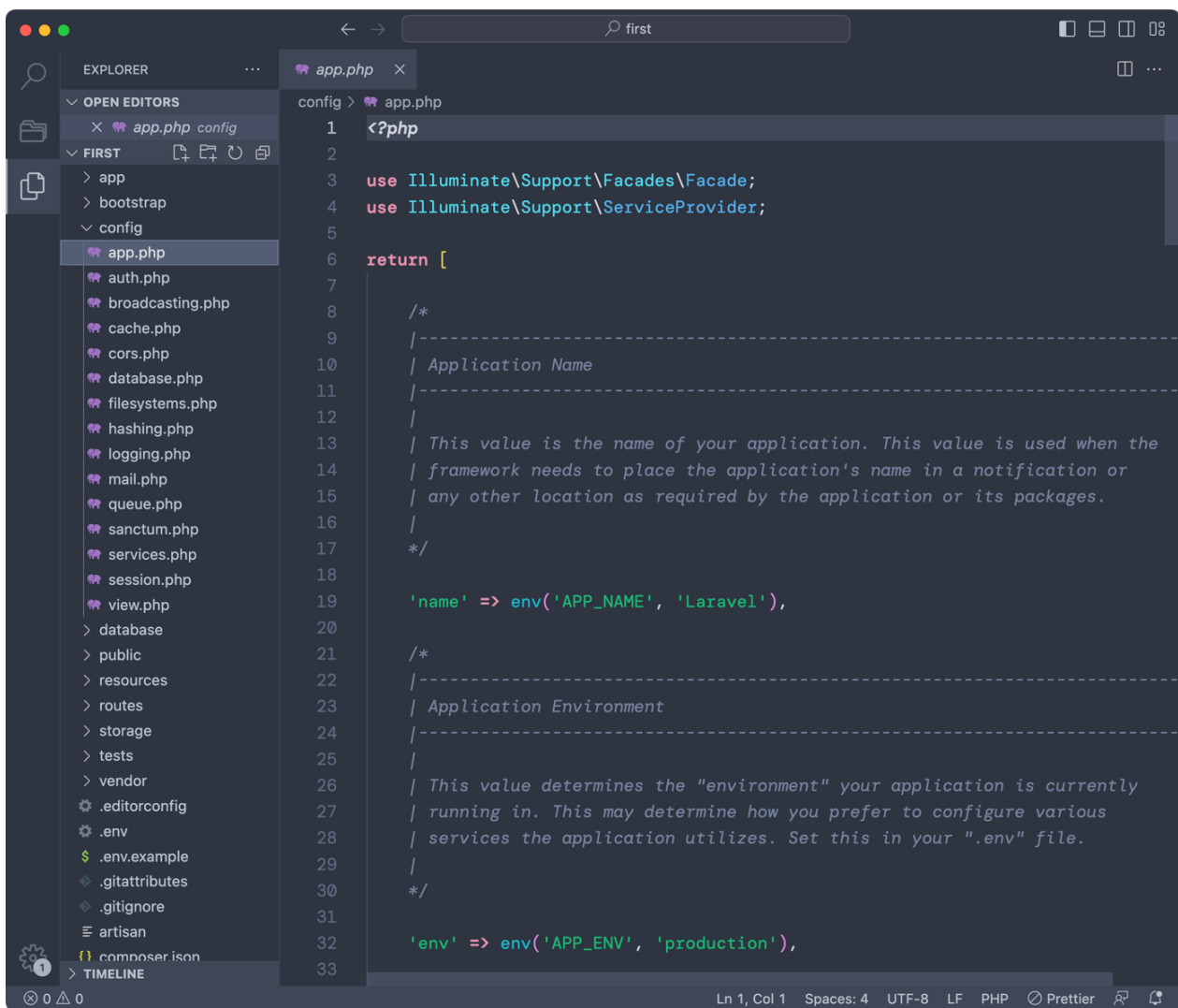
The first thing you're going to look at is the `.env` file.

It contains a lot of configuration options, called environment variables, for your app:

```
1 APP_NAME=Laravel
2 APP_ENV=local
3 APP_KEY=base64:eS4UWUTQcf3W3OCgd9uX1N0DMwYetotgDb2308VML/k=
4 APP_DEBUG=true
5 APP_URL=http://localhost
6
7 LOG_CHANNEL=stack
8 LOG_DEPRECATIONS_CHANNEL=null
9 LOG_LEVEL=debug
10
11 DB_CONNECTION=mysql
12 DB_HOST=127.0.0.1
13 DB_PORT=3306
14 DB_DATABASE=laravel
15 DB_USERNAME=root
16 DB_PASSWORD=
17
18 BROADCAST_DRIVER=log
19 CACHE_DRIVER=file
20 FILESYSTEM_DISK=local
21 QUEUE_CONNECTION=sync
22 SESSION_DRIVER=file
23 SESSION_LIFETIME=120
24
25 MEMCACHED_HOST=127.0.0.1
26
27 REDIS_HOST=127.0.0.1
28 REDIS_PASSWORD=null
29 REDIS_PORT=6379
30
31 MAIL_MAILER=smtp
32 MAIL_HOST=mailpit
33 MAIL_PORT=1025
```

For example in this portion of the file you can see we set the app name, the debug flag, the URL, settings related to logging, to the database connection, email sending and much more.

One very useful folder is `config`. Here's for example the `config/app.php` file:



Each file in the folder contain a lot of configuration options you can set, very well documented.

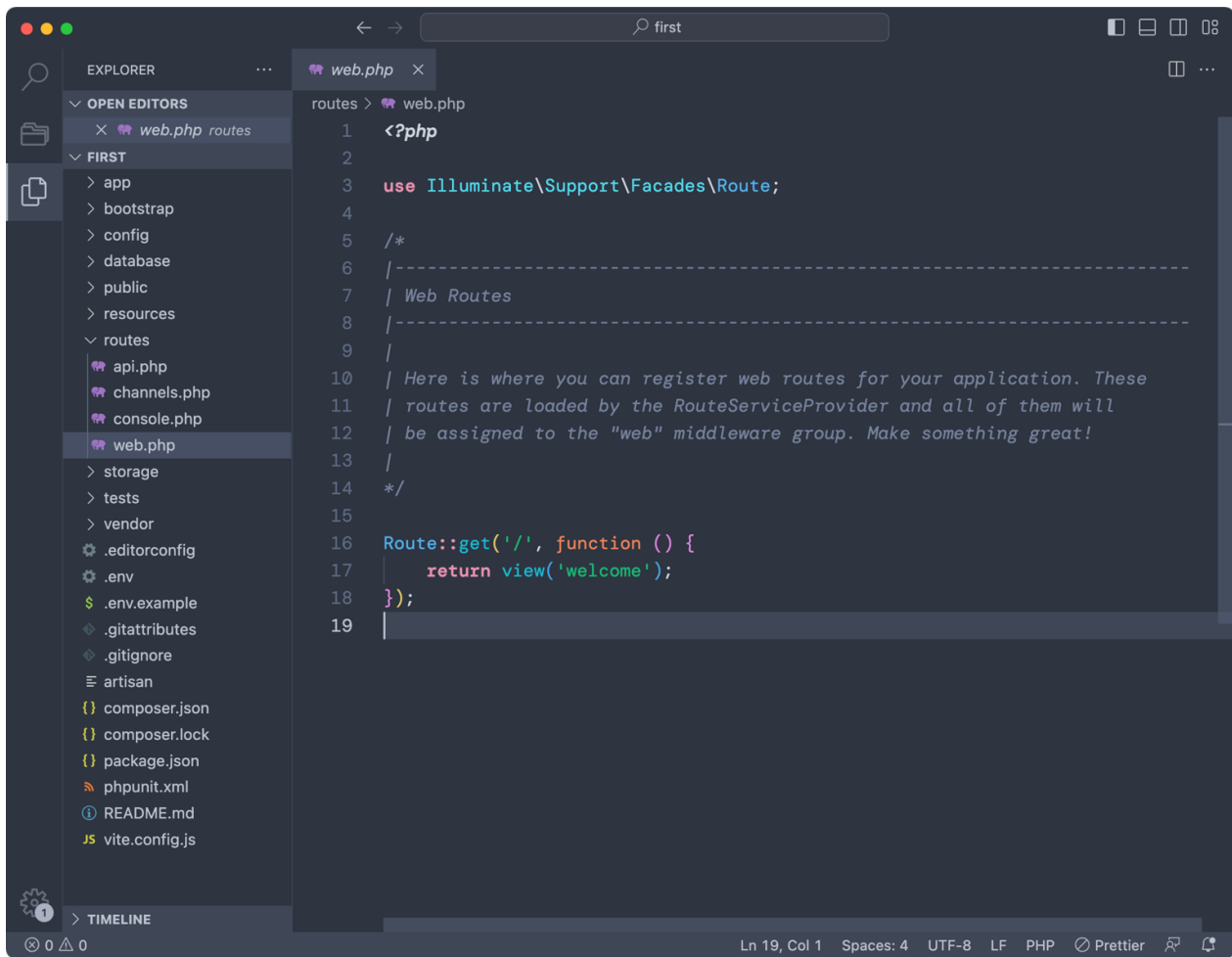
What's the difference between config files and the `.env` file? Environment variables in `.env` can be changed depending on the deployment, for example locally in development you can have debug enabled, while on the production server you don't want that.

Some options in `config` files, like the ones you see above, make use of the `env()` Laravel helper function to get the environment variable.

While options stored directly in the `config` folder hardcoded are "one for all environments".

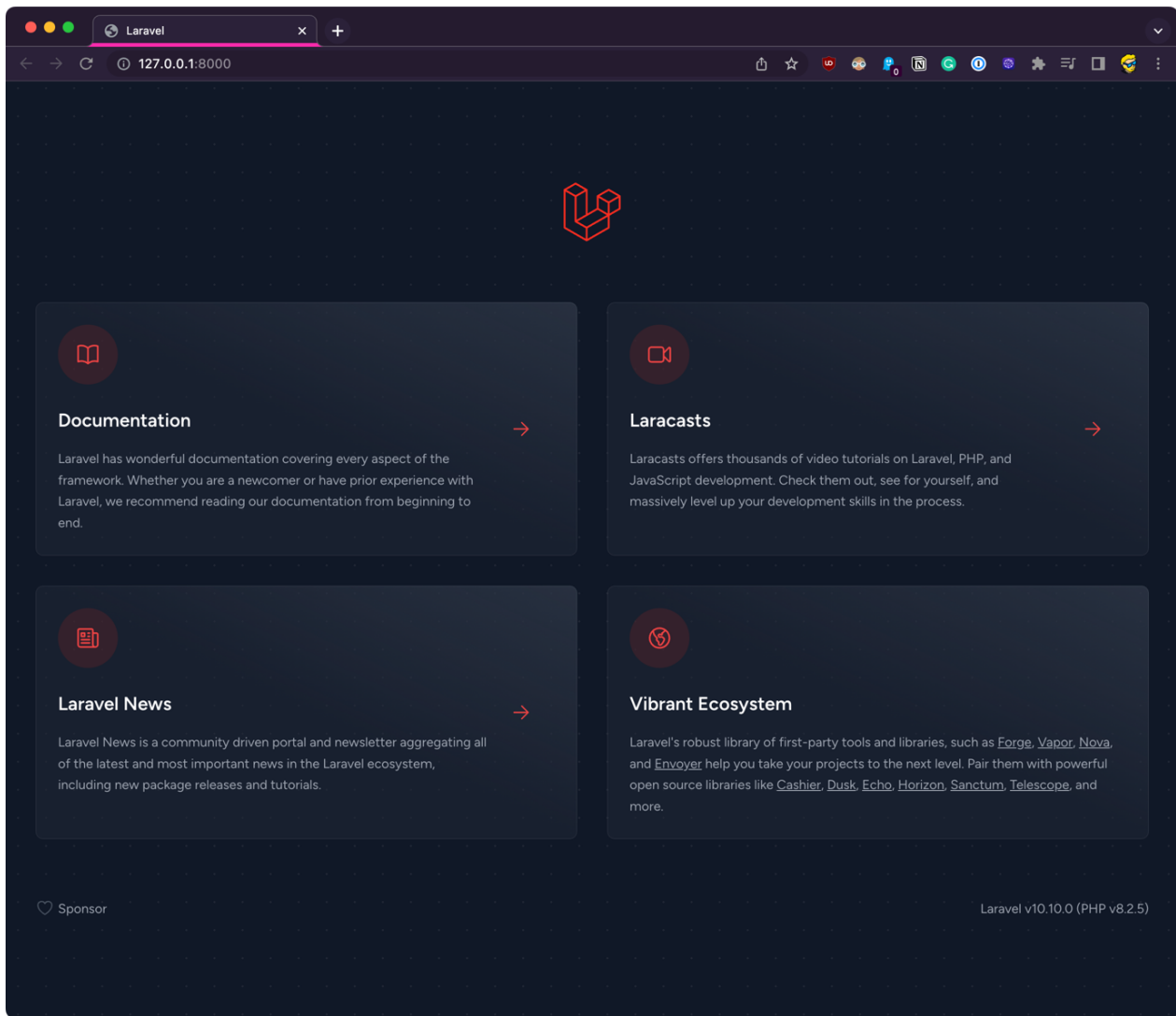
Before looking at changing any of the configuration options, let's modify what you see in the browser.

Open the `routes` folder and you'll 4 files. Open `web.php`:



```
1 <?php
2
3 use Illuminate\Support\Facades\Route;
4
5 /*
6 |-----|
7 | Web Routes
8 |-----|
9 |
10 | Here is where you can register web routes for your application. These
11 | routes are loaded by the RouteServiceProvider and all of them will
12 | be assigned to the "web" middleware group. Make something great!
13 |
14 */
15
16 Route::get('/', function () {
17     return view('welcome');
18 });
19
```

This is the code that displays the sample home page of the Laravel application:



We made a request to the `/` relative URL (`http://127.0.0.1:8000/`), which means the “home page”.

This URL is handled in the `routes/web.php` file, which contains the router dedicated to handling HTTP requests coming from the browser.

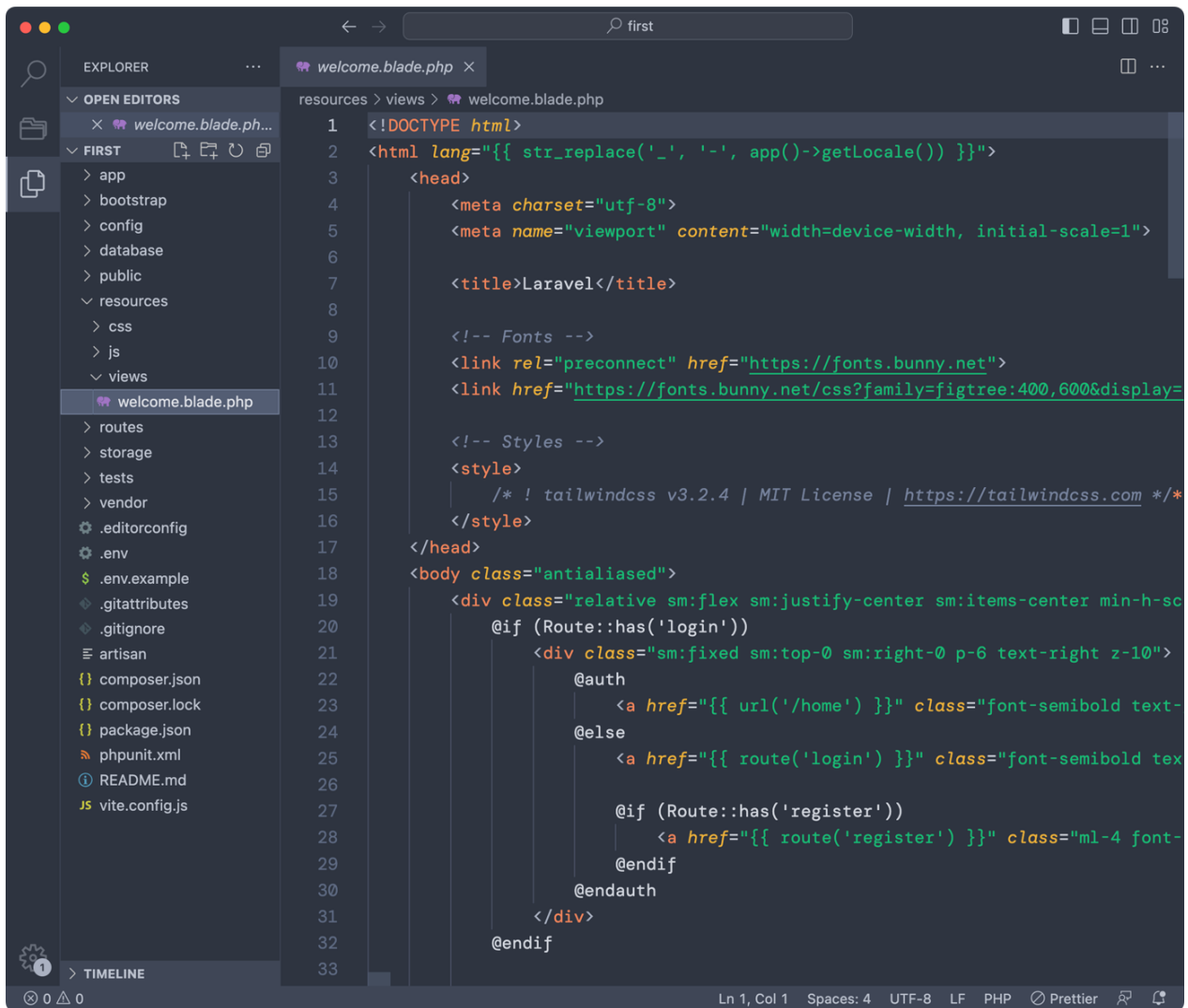
In this file, as shown in the screenshot, we tell Laravel to return the `welcome` view when someone visits the `/` URL using the `GET` HTTP method (the one used when you open the page in the browser):

```
Route::get('/', function () {  
    return view('welcome');  
});
```

To do this we use the `view()` Laravel helper, which knows where to find the `welcome` view because Laravel uses a set of conventions.

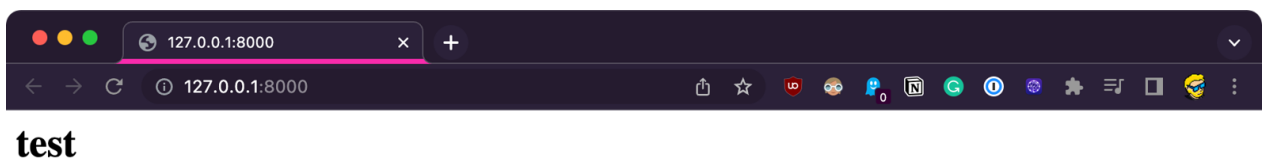
We have folders and files dedicated to holding specific, well-defined parts of our applications.

In this case the `welcome` view is defined in the file `resources/views/welcome.blade.php`:



```
1 <!DOCTYPE html>
2 <html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
3     <head>
4         <meta charset="utf-8">
5         <meta name="viewport" content="width=device-width, initial-scale=1">
6
7         <title>Laravel</title>
8
9         <!-- Fonts -->
10        <link rel="preconnect" href="https://fonts.bunny.net">
11        <link href="https://fonts.bunny.net/css?family=figtree:400,600&display=swap" rel="stylesheet">
12
13        <!-- Styles -->
14        <style>
15            /* ! tailwindcss v3.2.4 | MIT License | https://tailwindcss.com */
16        </style>
17    </head>
18    <body class="antialiased">
19        <div class="relative sm:flex sm:justify-center sm:items-center min-h-screen pt-6 sm:pt-0">
20            @if (Route::has('login'))
21                <div class="sm:fixed sm:top-0 sm:right-0 p-6 text-right z-10">
22                    @auth
23                        <a href="{{ url('/home') }}" class="font-semibold text-gray-500">Home</a>
24                    @else
25                        <a href="{{ route('login') }}" class="font-semibold text-gray-500">Log in</a>
26
27                        @if (Route::has('register'))
28                            <a href="{{ route('register') }}" class="ml-4 font-semibold text-gray-500">Register</a>
29                        @endif
30                    @endauth
31                </div>
32            @endif
33        </div>
```

You can clear all the content of this file, and type `<h1>test</h1>` into it. Save (cmd-s or ctrl-s) and reload in the browser, the homepage content will switch to displaying this string:



So now you know for sure that this file is responsible for what's shown on that URL!

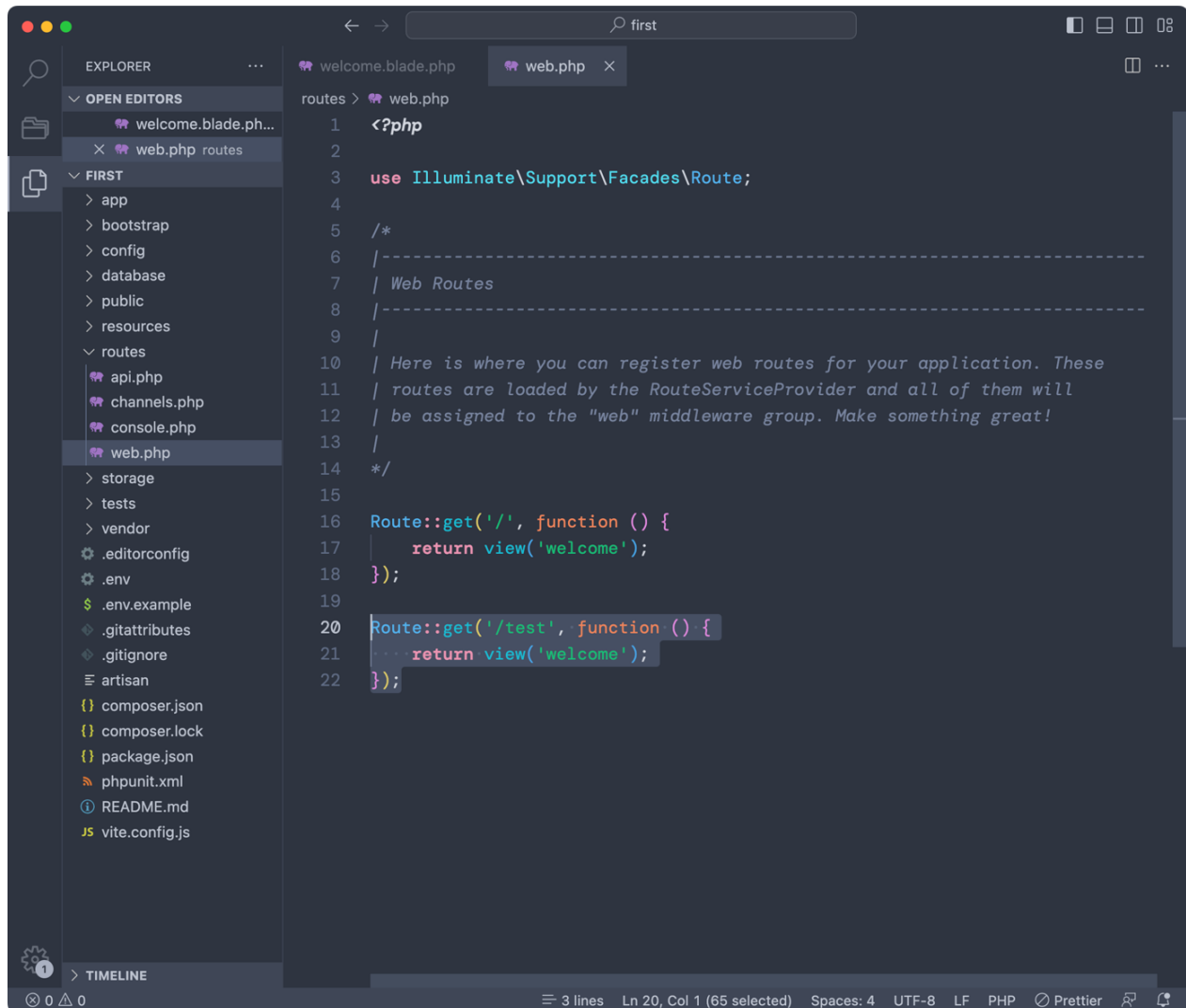
Now let's add a second page.

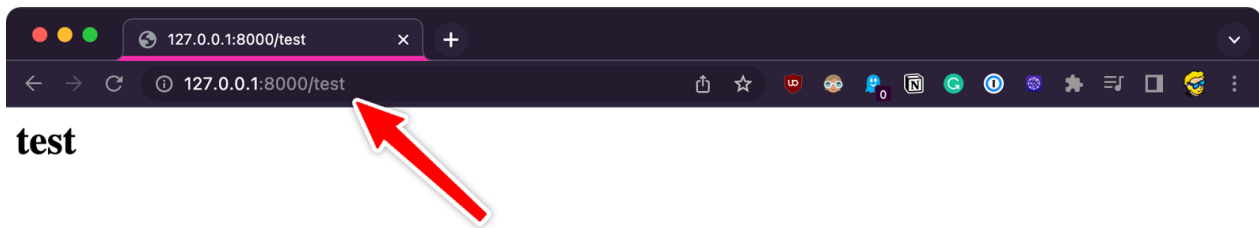
In `routes/web.php`, add:

```
//...
```

```
Route::get('/test', function () {  
    return view('welcome');  
});
```

This will render the `welcome` view also when the `/test` route is called:





You can show a different content by creating a new view in `resources/views` and using that view in the route, for example create a new view `resources/views/test.blade.php`

```
resources/views/test.blade.php
```

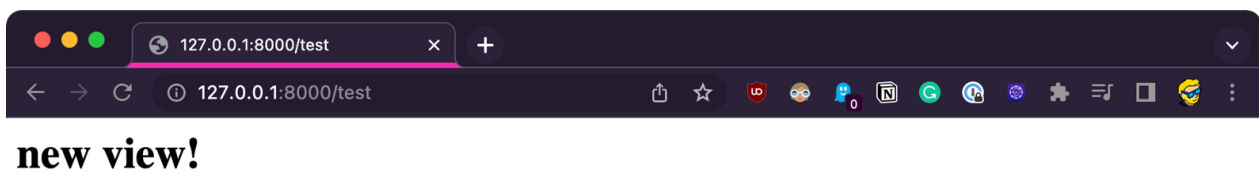
```
<h1>new view!</h1>
```

```
routes/web.php
```

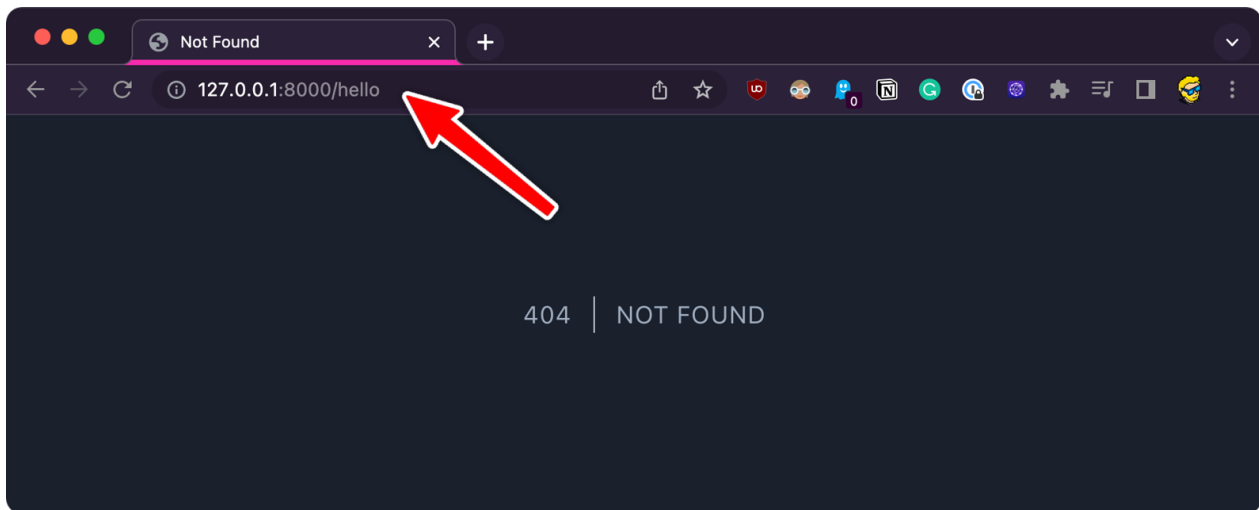
```
//...
```

```
Route::get('/test', function () {  
    return view('test');  
});
```

Here is the result:



Notice that any URL that does not have a specific entry in `routes/web.php` renders a “404 not found” page:

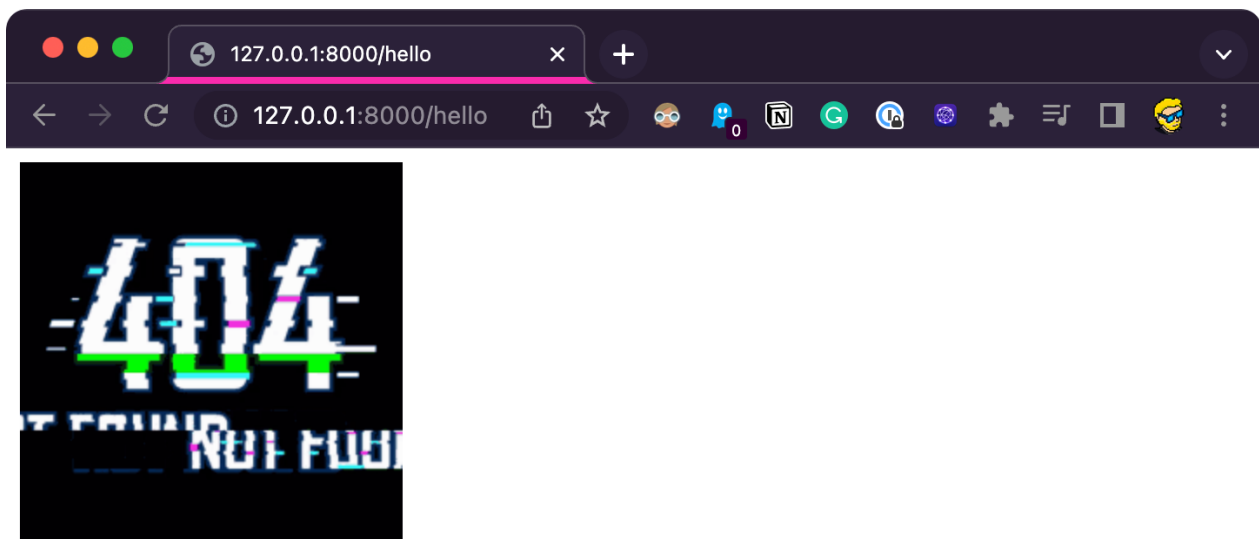


You can customize this error page. Here's how: create an **errors** folder in **resources/views**, and in there create a **404.blade.php** file. Add any content, like

```

```

And this will be rendered for 404 errors:



You didn't have to do anything more than creating the file, because Laravel has this set of conventions, so adding a file in the right place with the right name will do something specific.

3. Blade

The view files that end with **.blade.php** and are **Blade** templates.

Blade is a server-side templating language.

In its basic form it's HTML. As you can see, those templates I used above don't have anything other than HTML.

But you can do lots of interesting stuff in Blade templates: insert data, add conditionals, do loops,

display something if the user is authenticated or not, or show different information depending on the environment variables (e.g. if it's in production or development), and much more.

Here's a 101 on Blade (for more I highly recommend the [official Blade guide](#)).

In the route definition, you can pass data to a Blade template:

```
Route::get('/test', function () {
    return view('test', ['name' => 'Flavio']);
});
```

and use it like this:

```
<h1>{{ $name }}</h1>
```

The `{{ }}` syntax allows you to add any data to the template, escaped.

Inside it you can also run any PHP function you like, and Blade will display the return value of that execution.

You can comment using `{{-- --}}`:

```
{{-- <h1>test</h1> --}}
```

Conditionals are done using `@if @else @endif`:

```
@if ($name === 'Flavio')
    <h1>Yo {{ $name }}</h1>
@else
    <h1>Good morning {{ $name }}</h1>
@endif
```

There's also `@elseif`, `@unless` which let you do more complex conditional structures.

We also have `@switch` to show different things based on the result of a variable.

Then we have shortcuts for common operations, convenient to use:

- `@isset` shows a block if the argument is defined
- `@empty` shows a block if an array does not contain any element
- `@auth` shows a block if the user is authenticated
- `@guest` shows a block if the user is not authenticated
- `@production` shows a block if the environment is a production environment

Using the `@php` directive we can write any PHP:

```
@php
    $cats = array("Fluffy", "Mittens", "Whiskers", "Felix");
@endphp
```

We can do loops using these different directives

- `@for`
- `@foreach`
- `@while`

Like this:

```
@for ($i = 0; $i < 10; $i++)
    Count: {{ $i }}
@endfor

<ul>
    @foreach ($cats as $cat)
        <li>{{ $cat }}</li>
    @endforeach
</ul>
```

Like in most programming languages, we have directives to play with loops like `@continue` and `@break`.

Inside a loop a very convenient `$loop` variable is always available to tell us information about the loop, for example if it's the first iteration or the last, if it's even or odd, how many iterations were done and how many are left.

This is just a basic intro. We'll see more about Blade with components later.

4. Dynamic routes

We've seen how to create static routes with Laravel.

Sometimes you want your routes to be dynamic.

This will be especially useful with databases, but let's do an example without first.

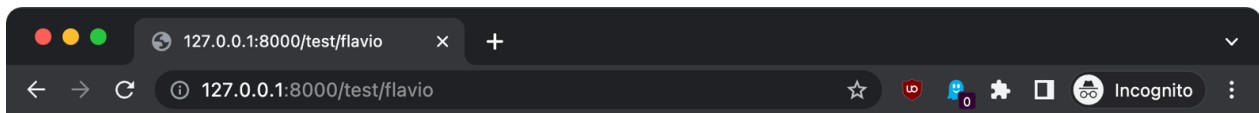
In `routes/web.php` add an entry like this:

```
Route::get('test/{name}', function($name) {
    return view('test', ['name' => $name]);
});
```

and in `resources/views/test.blade.php` write this code:

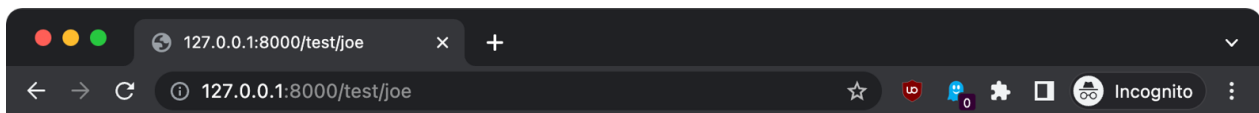
```
@if (isset($name))
    <h1>Hello {{ $name }}</h1>
@else
    <h1>Test</h1>
@endif
```

Now if you navigate with your browser to the `/test/flavio` route, "flavio" is the `$name` parameter in the route, which is passed to the view, so you can print it in the Blade template:



Hello flavio

Change the route parameter, the name in the HTML changes:



Hello joe

5. Adding a database

We're using Laravel in a very basic form, without any database.

Now I want to set up a database and configure Laravel to use it.

After we've configured the database, I'll show you how to use forms to accept user input and store data in the database, and how to visualize this data.

I'll also show you how you can use data from the database with dynamic routes.

The easiest way to use a database is by using SQLite.

SQLite is just a file hosted in your site, no special setup needed.

Open the `.env` file, and instead of the default configuration

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
```

DB_PASSWORD=

add

DB_CONNECTION=sqlite

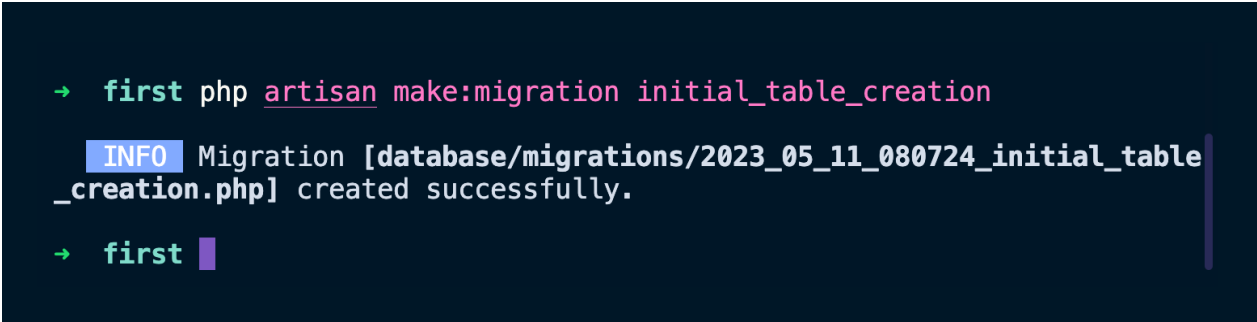
Laravel will automatically create a SQLite database in `database/database.sqlite` the first time you run a migration.

6. How to use migrations to create and modify the database schema

Migrations are excellent for handling changes to the database, so you can apply them, and roll back in case you need to restore to a previous state of your data.

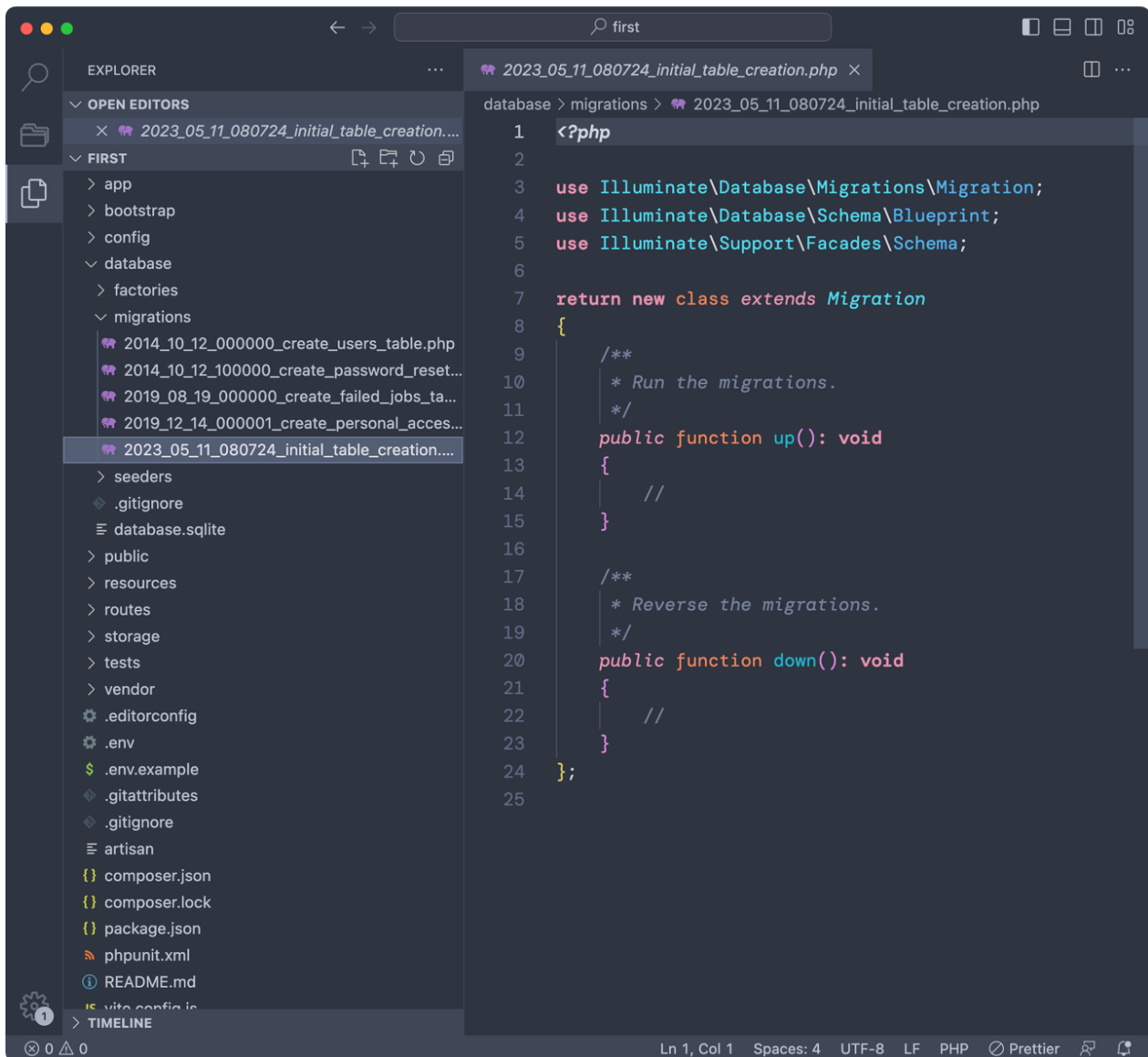
From the terminal, stop the Laravel server and run this command to create a new *migration*, which is what we will use to create the database table(s) we need to use:

```
php artisan make:migration initial_table_creation
```



```
→ first php artisan make:migration initial_table_creation
INFO Migration [database/migrations/2023_05_11_080724_initial_table_creation.php] created successfully.
→ first
```

This command created the `2023_05_11_080724_initial_table_creation.php` file (the date and time will of course change for you) in the `database/migrations` folder.



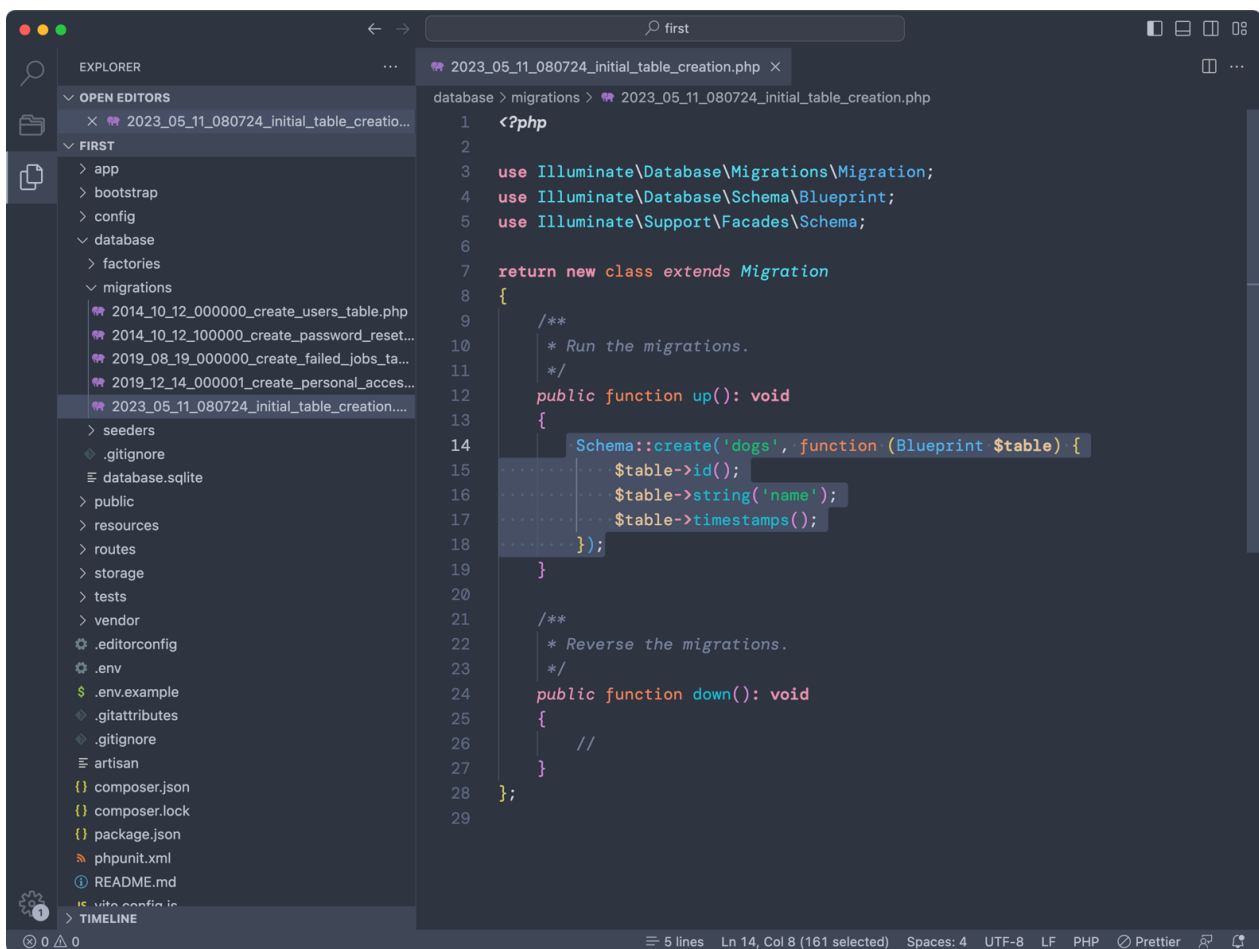
Notice there are other migrations, which are added by the Laravel framework itself for its authentication system.

But let's focus on creating a new table, let's call it **dogs**.

Go in the `up()` function of the migration we created.

Let's create a **dogs** table with 3 columns, an **id**, a **name** string and the timestamp utility columns (**created_at** and **updated_at**, as we'll see).

```
Schema::create('dogs', function (Blueprint $table) {  
    $table->id();  
    $table->string('name');  
    $table->timestamps();  
});
```



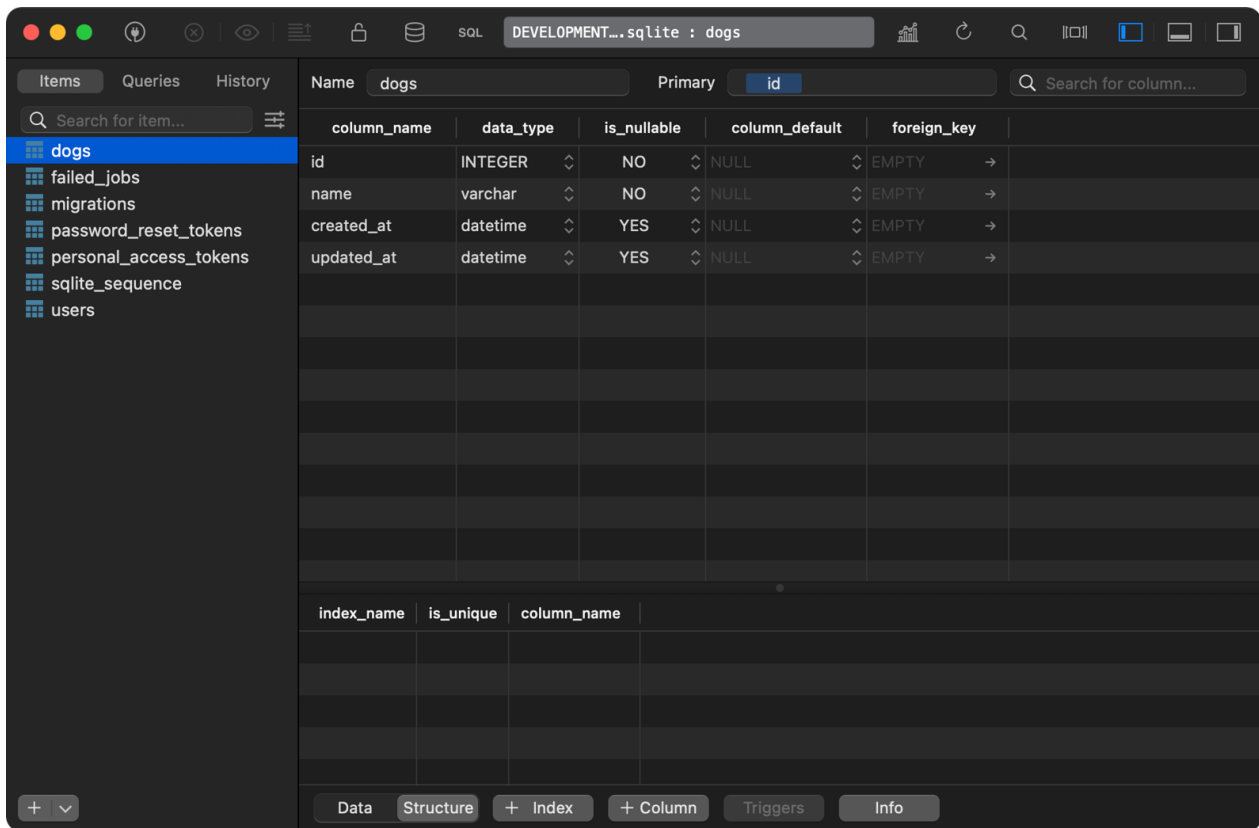
Now from the terminal run the command

```
php artisan migrate
```

And Laravel will apply the migrations that have not been applied yet, which at this point means all the migrations you see in the **migrations** folder:

```
~/d/first  
→ first php artisan migrate  
INFO Preparing database.  
Creating migration table ..... 2ms DONE  
INFO Running migrations.  
2014_10_12_000000_create_users_table ..... 1ms DONE  
2014_10_12_100000_create_password_reset_tokens_table ..... 1ms DONE  
2019_08_19_000000_create_failed_jobs_table ..... 1ms DONE  
2019_12_14_000001_create_personal_access_tokens_table .... 1ms DONE  
2023_05_11_080724_initial_table_creation ..... 0ms DONE  
→ first
```

If you open the database/database.sqlite file using a database visualization tool like [TablePlus](#) (free version, available for all operating systems) you will see the newly created tables, including the one we defined:



If you do a mistake in a migration, you can rollback any change in a migration using
`php artisan migrate:rollback`
and this rolls back the latest changes you did to the database.

Find more about migrations on the [official migrations guide](#).

7. Using forms to accept user input and store it into the database

Now we're going to create a form to add dogs to the table.

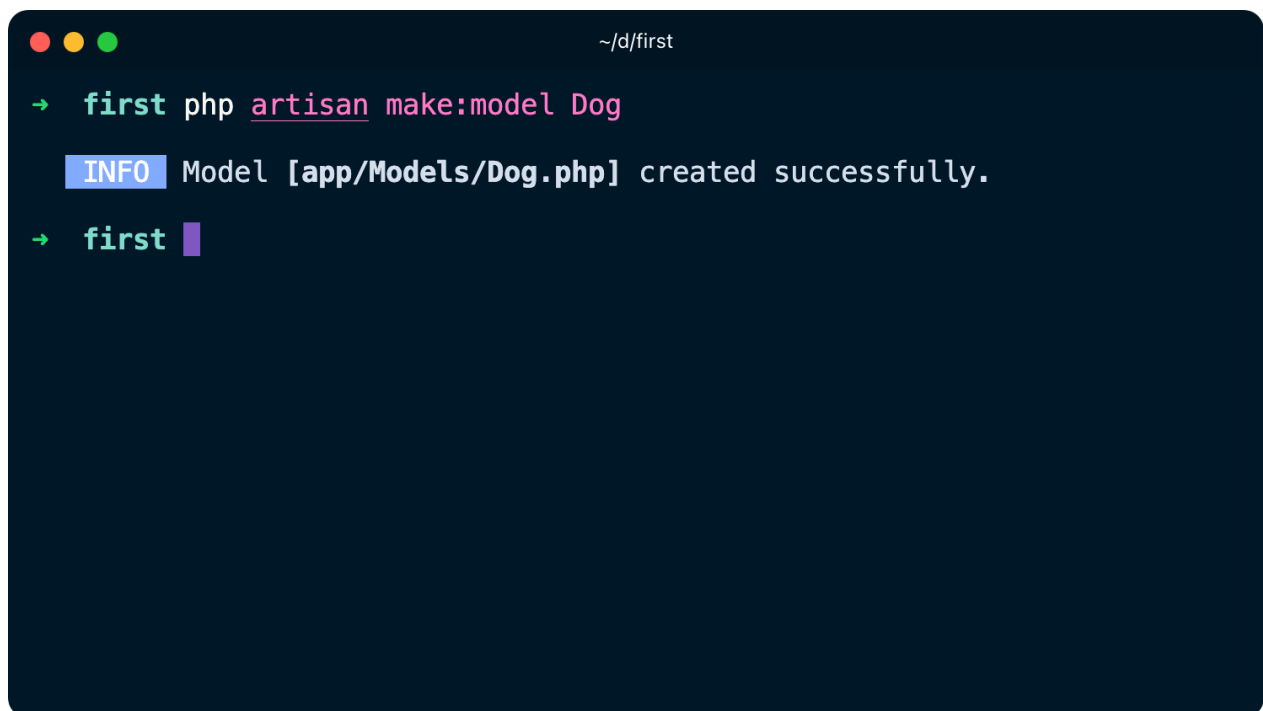
To do so, first we create a **Dog model**.

What's a model? A model is a class that allows us to interact with data stored in the database.

Each model represents a specific table in the database, and we use it to create, read, update and delete records.

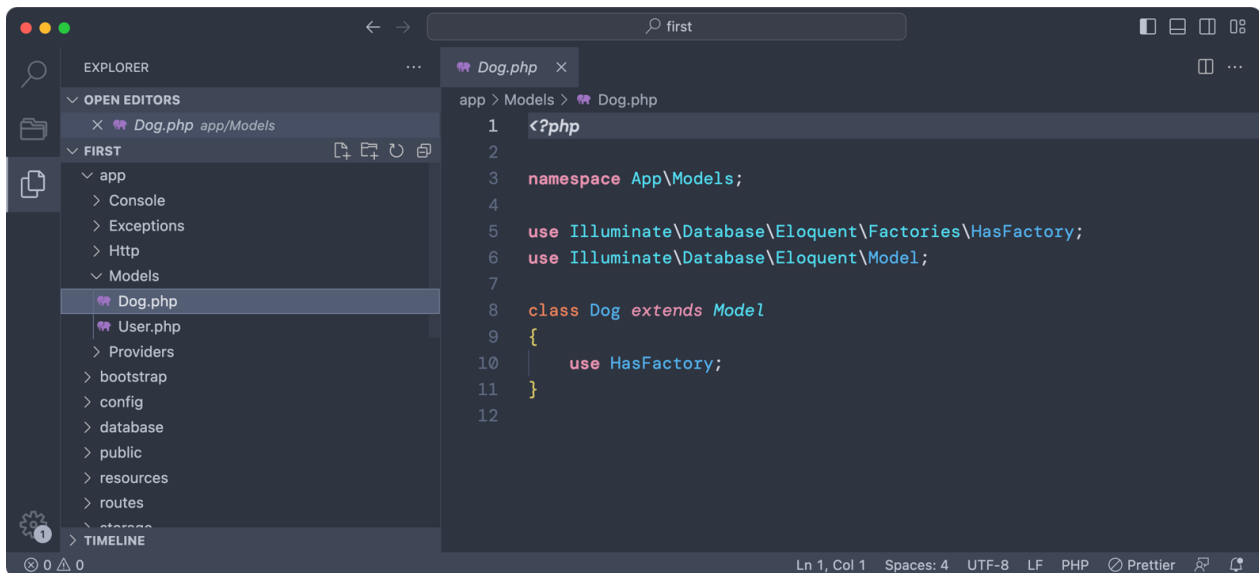
Create the model from the terminal with this command:

```
php artisan make:model Dog
```

A terminal window with a dark blue background and light green text. The window title is '~d/first'. The prompt is 'first'. The command 'php artisan make:model Dog' has been entered. The output is 'INFO Model [app/Models/Dog.php] created successfully.' followed by a new line and the prompt 'first' again.

```
~d/first  
→ first php artisan make:model Dog  
INFO Model [app/Models/Dog.php] created successfully.  
→ first
```

This creates a model in `app/Models/Dog.php`:



Notice the class includes some classes under a “Eloquent” folder.

Eloquent is an **ORM** (object-relational mapper), a tool that basically lets us interact with a database using a (PHP, in this case) class.

The model has a corresponding table, which we do not mention, but it’s the **dogs** table we created beforehand because of the naming convention **dogs** table → **Dog** model.

We’re going to use this model to add an entry to the database.

We’ll show the user a form and they can add a dog name, and click “Add” and the dog will be added to the database.

First we add the **name** field we added to the table to an array named **\$fillable**:

```
protected $fillable = ['name'];
```

Like this:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Dog extends Model
{
    use HasFactory;
    protected $fillable = ['name'];
}
```

A model is a **resource**, and once you define a model you’ll later be able to create a new resource, delete, update it.

Now let’s build a form to add a new dog to the database.

Let’s add a new entry to **routes/web.php**

```
Route::get('/newdog', function () {
    return view('newdog');
});
```

We create a controller named `DogController`:

```
php artisan make:controller DogController
```

Laravel adds a `DogController.php` file into the folder `app/Http/Controllers/`

What is a controller? A controller takes an *action* and determines what to do.

For example we'll create a form that sends a POST request to the `/dogs` route.

The router will say “this controller is in charge” and will tell us which method to use.

Inside the controller we write methods that perform actions, like adding data to the database, or updating it.

If you're unsure what is a POST request, check my [HTTP tutorial](#).

We will start by adding a `create` method to the controller to handle the data coming from the form, so we can store that to the database.

Before doing so, in `routes/web.php` we add the POST `/dogs` route handle controller and we assign it the name `dog.create`

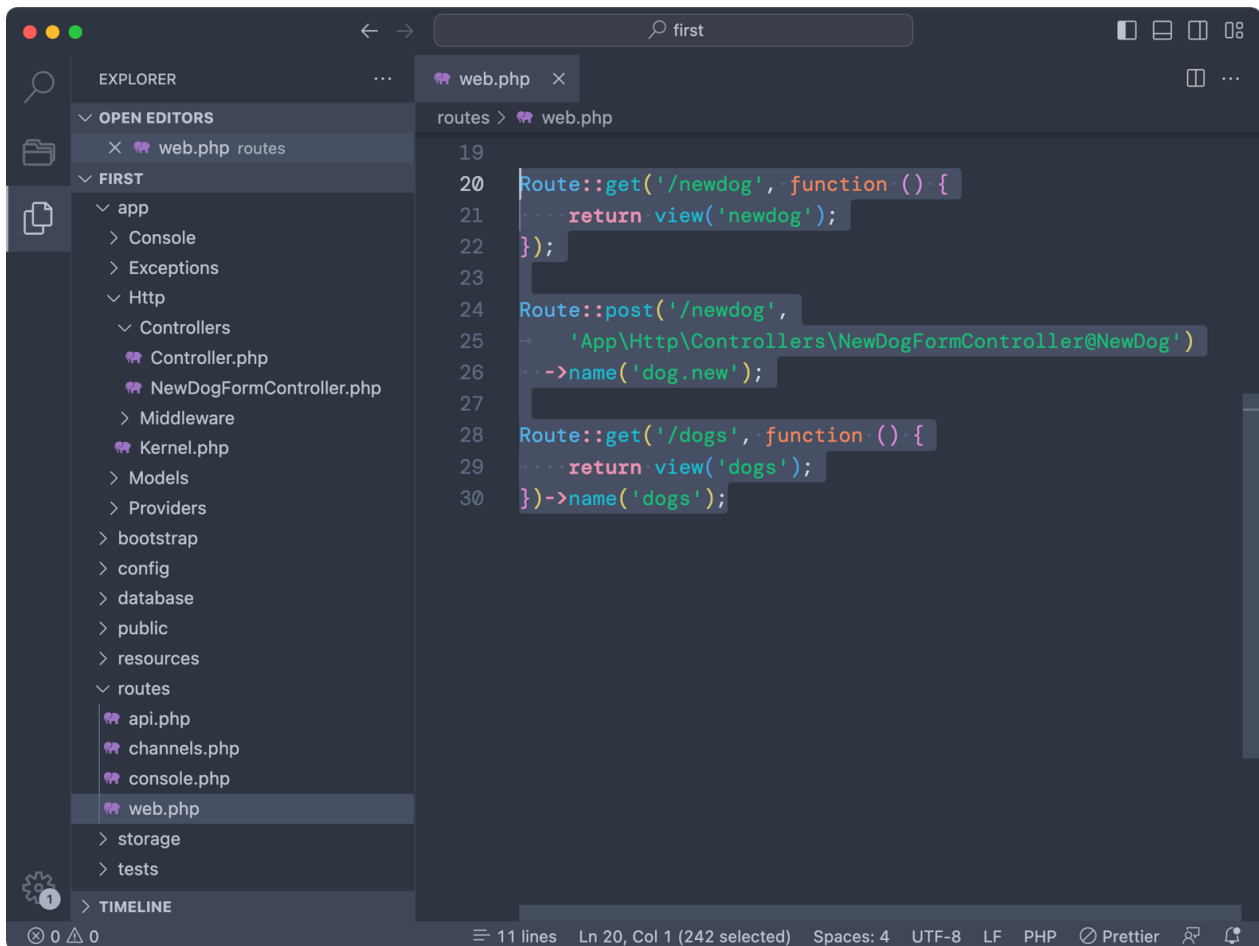
We also add a `/dogs` route which we call `dogs`. We now render the `dogs` view in it, which we have to create yet:

```
use App\Http\Controllers\DogController;
```

```
//...
```

```
Route::post(
    '/dogs',
    [DogController::class, 'create']
)->name('dog.create');
```

```
Route::get('/dogs', function () {
    return view('dogs');
})->name('dogs');
```

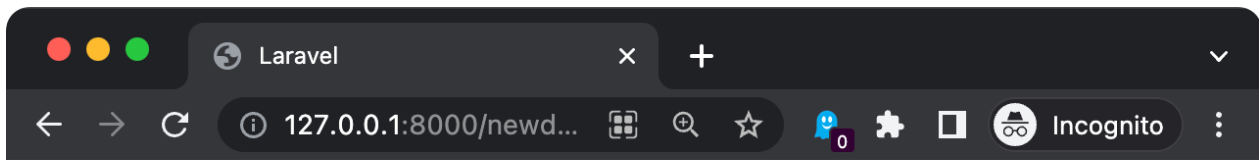


In `resources/views/` create a `newdog.blade.php` file, which contains a form whose **action** attribute points to the `dog.create` route:

```
<form method="post" action="{{ route('dog.create') }}">
    @csrf
    <label>Name</label>
    <input type="text" name="name" id="name">
    <input type="submit" name="send" value="Submit">
</form>
```

Run `php artisan serve` if you stopped the service, and go to <http://127.0.0.1:8000/newdog>

The style is not brilliant, but the form shows up:



Name

Now back to the `app/Http/Controllers/DogController.php` file.

Inside the class we import the Dog model, and we add the `create` method which will first validate the form, then store the dog into the database.

Finally we redirect to the `index` route:

```
<?php

namespace App\Http\Controllers;

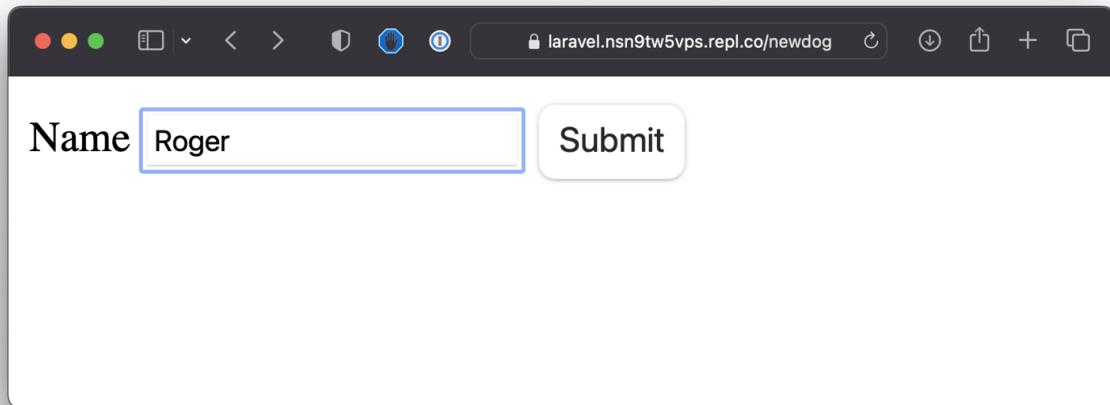
use Illuminate\Http\Request;

use App\Models\Dog;

class NewDogFormController extends Controller
{
    public function create(Request $request)
    {
        $this->validate($request, [
            'name' => 'required',
        ]);
        Dog::create($request->all());

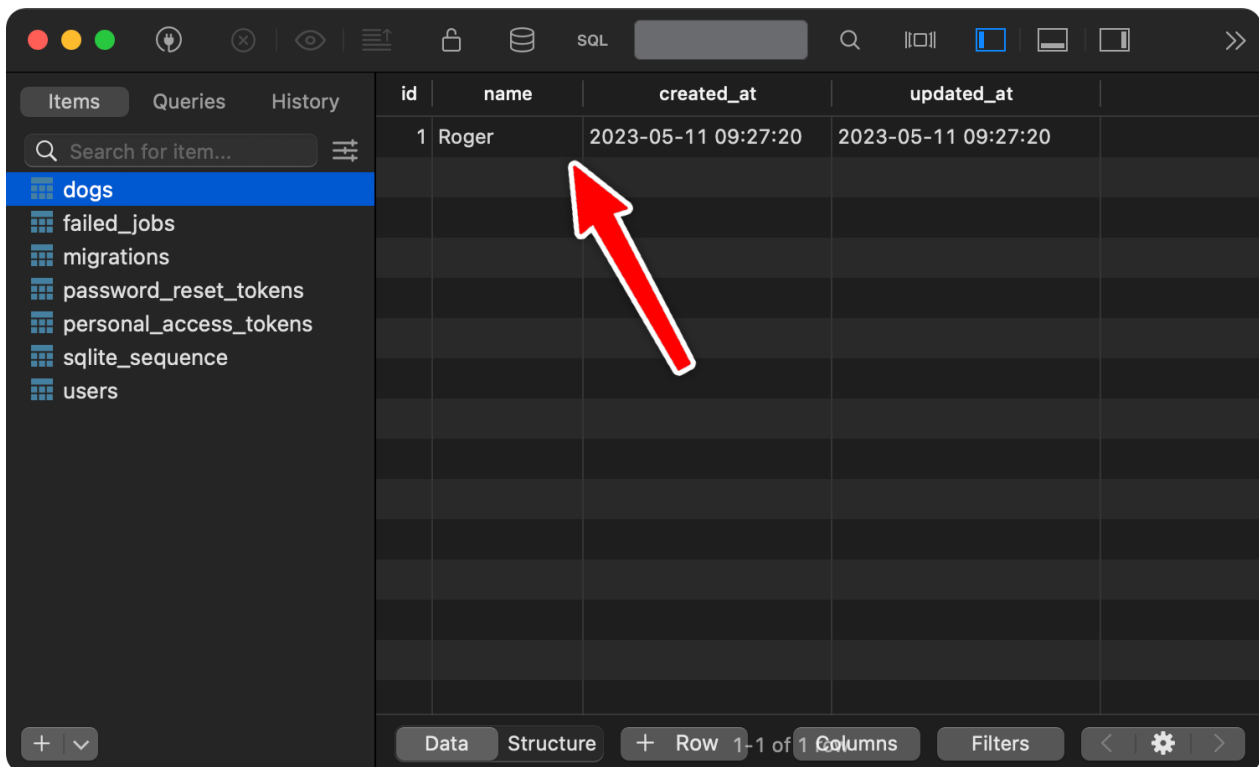
        return to_route('index');
    }
}
```

Now back to the form, enter a name and click “Submit”:



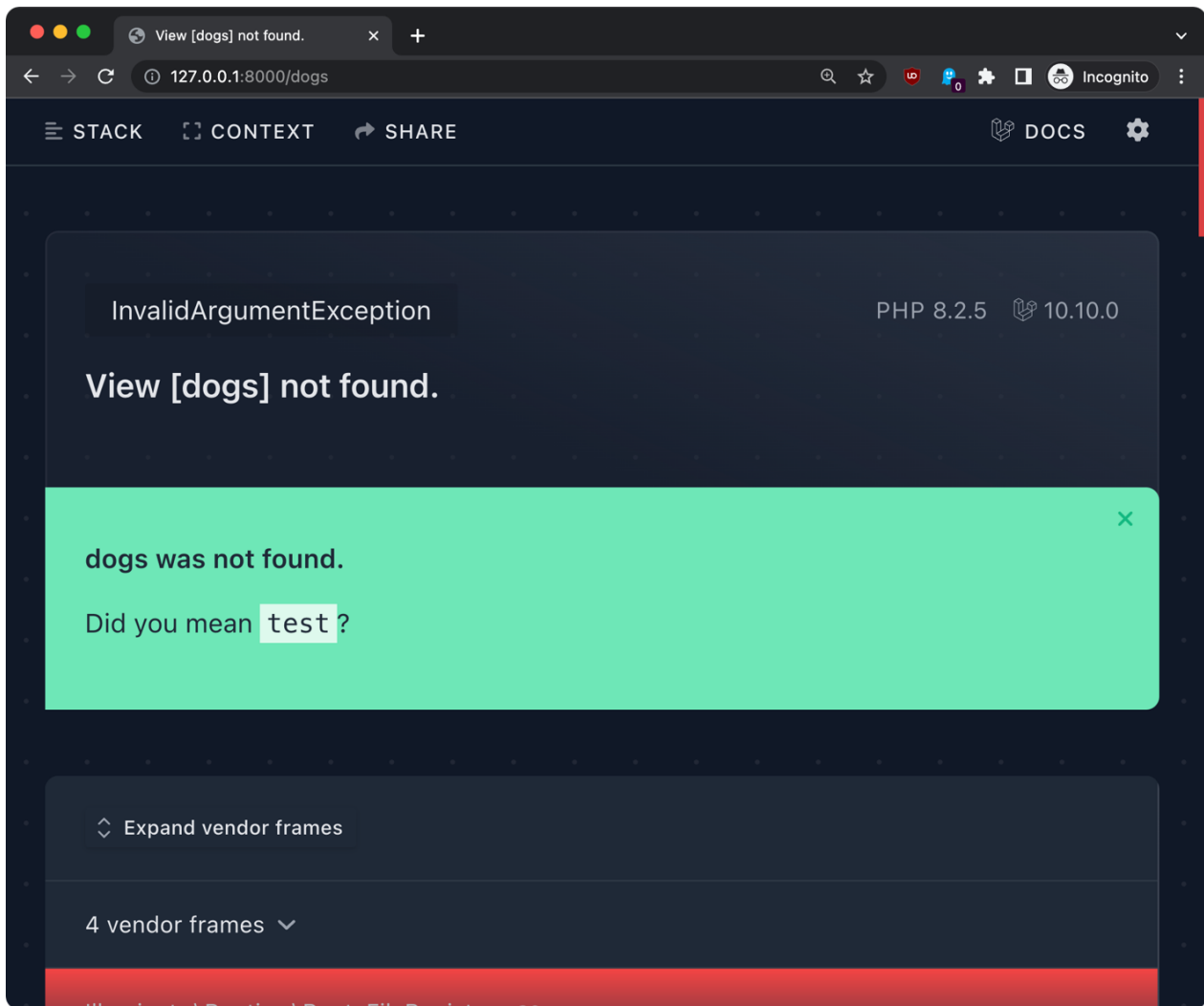
Name

You will be redirected to `/dogs`, after the new dog was saved to the database.



id	name	created_at	updated_at
1	Roger	2023-05-11 09:27:20	2023-05-11 09:27:20

In the browser there's an error now but don't worry - it's because we haven't added a `dogs` view yet.



In this view we'll visualize the database data.

Create the file `resources/views/dogs.blade.php` and in there we're going to loop over the `$dogs` array with Blade to display the data to the user:

```
@foreach ($dogs as $dog)
    {{ $dog->name }}
@endforeach
```

This data does not come from nowhere. It must be passed to the template.

So in `routes/web.php` we now have

```
Route::get('/dogs', function () {
    return view('dogs');
})->name('dogs');
```

and we have to first retrieve the data from the model, and pass it to the view.

First we import the model at the top of the file:

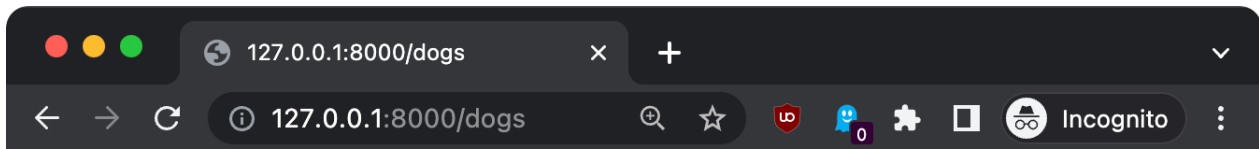
```
use App\Models\Dog;
```

Then in the route we call `Dog::all()` to get all the dogs stored and we assign them to a `$dogs`

variable which we pass to the template:

```
Route::get('/dogs', function () {  
    $dogs = Dog::all();  
    return view('dogs', ['dogs' => $dogs]);  
})->name('dogs');
```

Here's the result:



Roger Syd Botolo Zoe

The screenshot shows a database application with a sidebar on the left containing a list of items: `dogs`, `failed_jobs`, `migrations`, `password_reset_tokens`, `personal_access_tokens`, `sqlite_sequence`, and `users`. The `dogs` item is selected. The main area displays a table with the following data:

id	name	created_at	updated_at
1	Roger	2023-05-11 09:27:20	2023-05-11 09:27:20
2	Syd	2023-05-11 09:29:52	2023-05-11 09:29:52
3	Botolo	2023-05-11 09:29:57	2023-05-11 09:29:57
4	Zoe	2023-05-11 09:30:12	2023-05-11 09:30:12

A red arrow points to the `created_at` column header. The bottom of the interface shows a status bar with 'Data' and 'Structure' tabs, and a message 'Row 4 of 4 rows'.

8. Adding a better layout

Now that we got data working, let's clean up the routes a bit, add a more beautiful design.

I have this list of views we used in our tests:

```

  ✓ resources
    > css
    > js
    ✓ views
      > errors
      🐘 dogs.blade.php
      🐘 newdog.blade.php
      🐘 test.blade.php
      🐘 welcome.blade.php

```

Remove `test.blade.php` and `welcome.blade.php`.

In `routes/web.php` we're going to show the `dogs` view on `/`, which we name the `index` route, and we show the form to add a new dog on `/newdog`. Doing a POST request on that route will trigger the `create` method on the `DogController` to save the dog to the database. Remove all the other routes.

```
<?php
```

```
use Illuminate\Support\Facades\Route;
use App\Models\Dog;
```

```
/*
|-----
| Web Routes
|-----
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider and all of them will
| be assigned to the "web" middleware group. Make something great!
|
*/
```

```
Route::get('/newdog', function () {
```

```

        return view('newdog');
    });

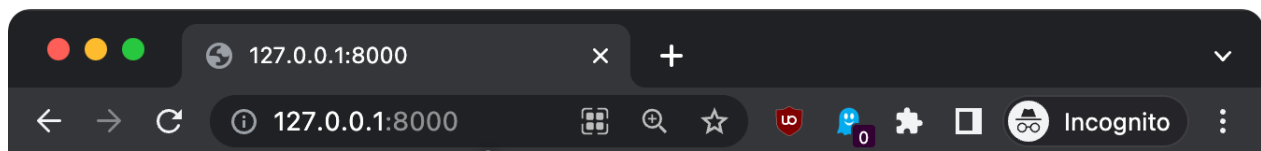
Route::post(
    '/dogs',
    [DogController::class, 'create']
)->name('dog.create');

Route::get('/', function () {
    $dogs = Dog::all();
    return view('dogs', ['dogs' => $dogs]);
})->name('index');

```

Ok!

Now you should see the list of dogs on the / route:



Roger Syd Botolo Zoo

In `resources/views/dogs.blade.php` we now have a super simple

```

@foreach ($dogs as $dog)
    {{ $dog->name }}
@endforeach

```

which does not even contain any HTML.

The browser renders that because it tries its best to display something useful, but let's do things properly.

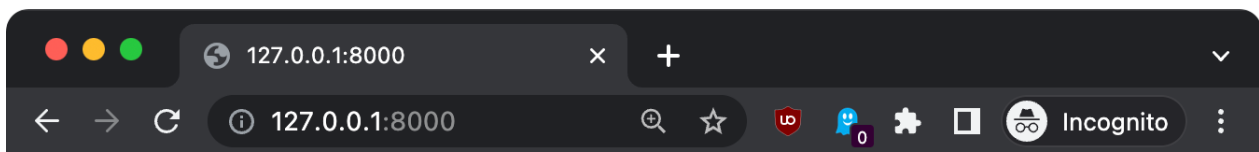
Here's a start: we add the proper HTML structure and we wrap the dogs list in an unordered list:

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
  <meta name="viewport"
    content="width=device-width, initial-scale=1.0">
</head>

<body>
  <h1>
    Dogs
  </h1>
  <ul>
    @foreach ($dogs as $dog)
      <li>{{ $dog->name }}</li>
    @endforeach
  </ul>
</body>

</html>
```



Dogs

- Roger
- Syd
- Botolo
- Zoe

The next thing we'll do is **configure Vite**, so we can enable styling using **Tailwind CSS**, a very useful library.

First go back to the terminal.

Run this:

```
npm install -D tailwindcss postcss autoprefixer
```

If you don't have npm installed yet, [install Node.js](#) first.

This command will create a `package.json` file, a `package-lock.json` and a `node_modules` folder.

Then run this:

```
npx tailwindcss init -p
```

This will create the `tailwind.config.js` and the `postcss.config.js` files.

(see my [npx tutorial](#) if you're new to that, it's installed automatically with Node.js, as `npm`).

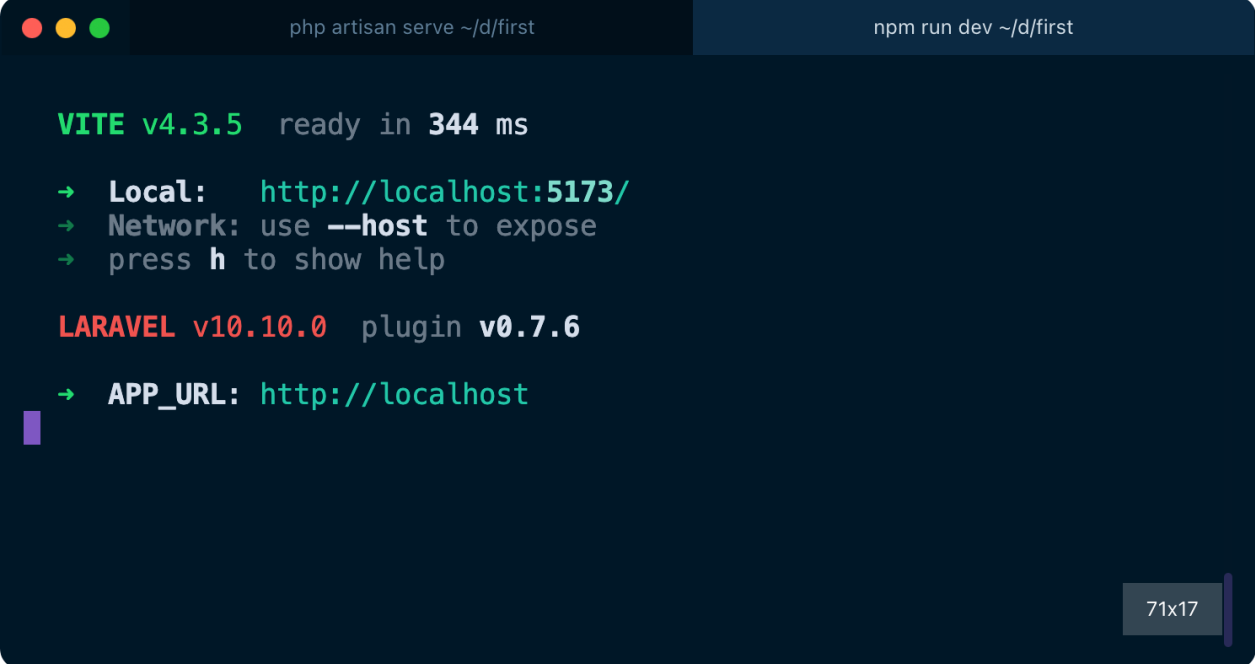
Now open `tailwind.config.js` and add this:

```
/** @type {import('tailwindcss').Config} */
export default {
  content: ["./resources/**/*.blade.php"],
  theme: {
    extend: {},
  },
  plugins: [],
};
```

In `resources/css/app.css` add this:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

Finally, back to the terminal, run `npm run dev` and keep it running while developing the site, as `php artisan serve` (run both in 2 different terminal windows).



The screenshot shows a terminal window with two tabs: 'php artisan serve ~/d/first' and 'npm run dev ~/d/first'. The active tab is 'npm run dev ~/d/first'. The terminal output shows Vite v4.3.5 ready in 344 ms, with local and network URLs. It also shows Laravel v10.10.0 plugin v0.7.6 and the APP_URL. A cursor is visible at the bottom left of the terminal.

```
php artisan serve ~/d/first  npm run dev ~/d/first

VITE v4.3.5  ready in 344 ms
→ Local:   http://localhost:5173/
→ Network: use --host to expose
→ press h to show help

LARAVEL v10.10.0  plugin v0.7.6
→ APP_URL: http://localhost
└─
```

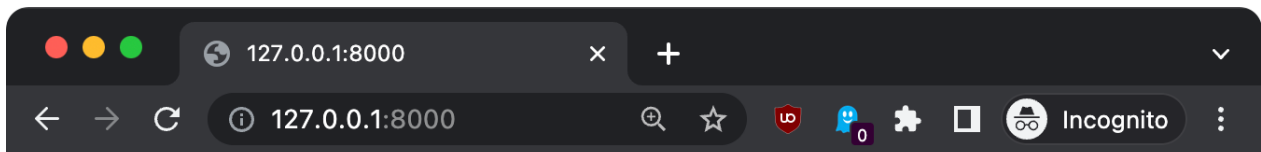
Now we're ready to use Tailwind CSS in our Blade templates!

Add this line to the page head:

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
  <meta name="viewport"
    content="width=device-width, initial-scale=1.0">
  @vite('resources/css/app.css')
```

If you refresh the page, you can see immediately that something changed. It's Tailwind adding [some default normalization](#), so that's a sign it's working!



Dogs

Roger

Syd

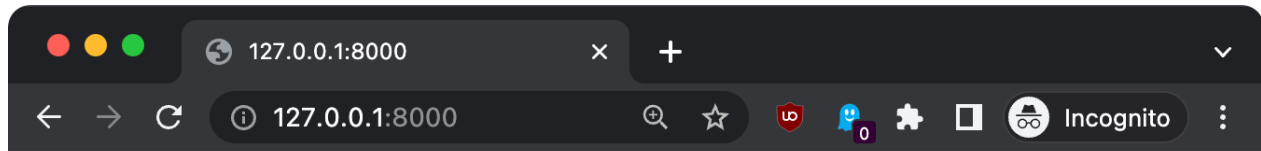
Botolo

Zoe

Now we can add classes to our HTML body to style the page a bit:

```
<body class="p-4">
  <h1 class="font-bold border-b-gray-300 border-b pb-2 mb-3">
    Dogs
  </h1>
  <ul>
    @foreach ($dogs as $dog)
      <li>{{ $dog->name }}</li>
    @endforeach
  </ul>
</body>
```

Here's the result, much better!



Dogs

Roger

Syd

Botolo

Zoe

Notice that changes are applied automatically when you save the file in VS Code, without refreshing the page. Both changes in the Blade template, and in the Tailwind CSS classes. That's some "magic" provided by Vite and Laravel in development mode.

9. Adding the form at the bottom of the list

Now I want to do something. On <http://127.0.0.1:8000/newdog> we still got the "add dog" form. But I want to add it at the bottom of this list.

How do we do that? Using **subviews**.

Using the `@include` directive we can include a view within another view.

So let's include the "new dog form" in the `dogs.blade.php` template:

```
<!doctype html>
<html>

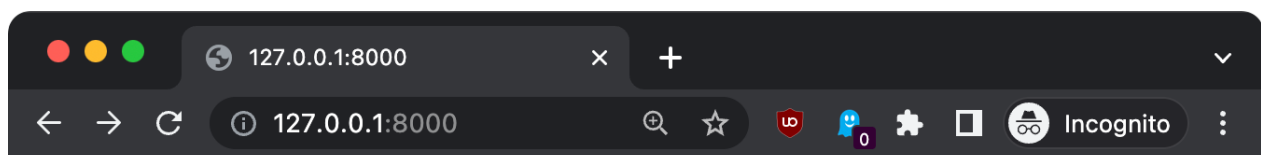
<head>
  <meta charset="utf-8">
  <meta name="viewport"
    content="width=device-width, initial-scale=1.0">
  @vite('resources/css/app.css')
</head>

<body class="p-4">
```

```
<h1 class="font-bold border-b-gray-300 border-b pb-2 mb-3">
  Dogs
</h1>
<ul>
  @foreach ($dogs as $dog)
    <li>{{ $dog->name }}</li>
  @endforeach
</ul>
@include('newdog')
</body>

</html>
```

It works!



Dogs

Roger

Syd

Botolo

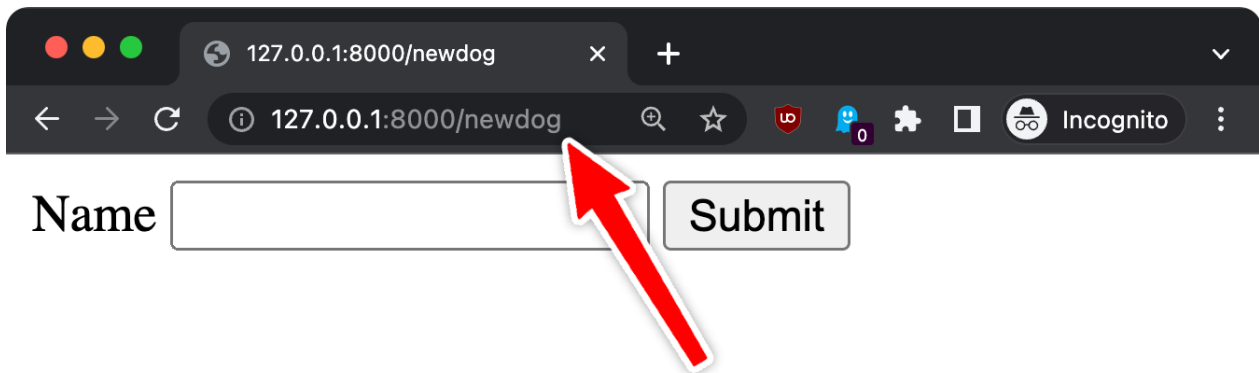
Zoe

Name



Submit

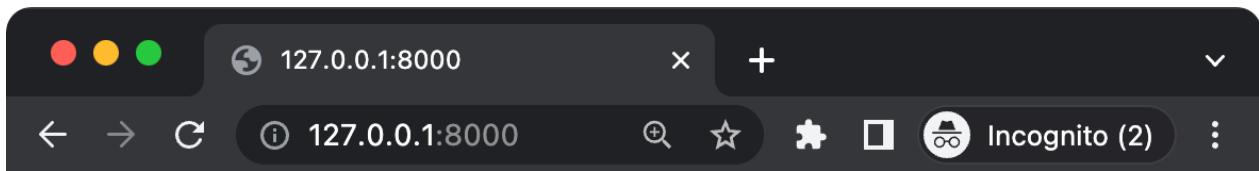
But now since we use Tailwind, the form looks different than the “standalone” route to add a new dog:



Let's style it a bit:

```
<form method="post" action="{{ route('dog.new') }}">
  @csrf
  <h3 class="font-bold border-b-gray-300
            border-b pb-2 mb-3 mt-4">
    Add a new dog</h3>
  <label>Name</label>
  <input type="text" name="name" id="name"
        class="border border-gray-200 p-1">
  <input type="submit" name="send" value="Submit"
        class="bg-gray-200 p-1 cursor-pointer
              border border-black">
</form>
```

Here's the result:



Dogs

Roger

Syd

Perfect

Add a new dog

Name

Submit

Now, we don't want the form to have its own route any more, because we have it on `/`.

So let's create a folder named `partials` in `resources/views` and move the file `resources/views/newdog.blade.php` to `resources/views/partials/form.blade.php`

In `resources/views/dogs.blade.php` change

```
@include('newdog')
```

to

```
@include('partials.form')
```

and in `routes/web.php` you can now delete the GET route that showed that form on `/newdog`:

```
Route::get('/newdog', function () {  
    return view('newdog');  
});
```

10. Allow users to delete dogs from the list

We allow users to add dogs to the list.

Let's allow them to remove them, too.

Here's how.

First we add a “delete” button next to each item:

```
<ul>
  @foreach ($dogs as $dog)
    <li class="flex mb-1">
      <span class="flex-1">{{ $dog->name }}</span>
      <form action="{{ route('dog.delete', $dog->id) }}"
        method="POST">
        @csrf
        @method('DELETE')
        <button type="submit" class="border
          bg-gray-200 p-1 border-black">Delete</button>
      </form>
    </li>
  @endforeach
</ul>
```

We use [Flexbox](#) to align the dog name and the delete button. Adding the `flex-1` class makes the text take all the space available, and “pushes” the button to the far right.

Then we add a route named `dog.delete` to the `routes/web.php` file:

```
Route::delete(
    '/dog/{id}',
    [DogController::class, 'delete']
)->name('dog.delete');
```

This calls the `delete` method on the `DogController`, so we go to `app/Http/Controllers/DogController.php` and we add it:

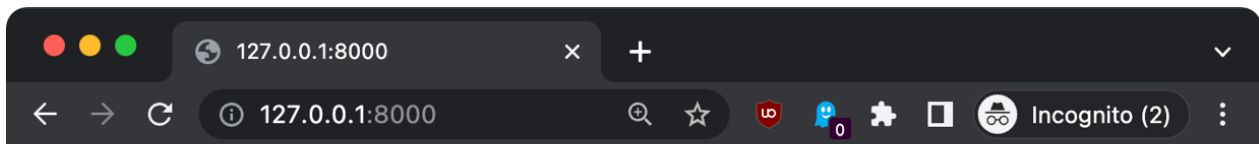
```
//...

class NewDogFormController extends Controller
{
    //...
    public function delete($id)
    {
        $dog = Dog::find($id);
        $dog->delete();

        return to_route('index');
    }
}
```

This method uses the `Dog` model to find a dog with a specific id, which is passed by the form, and deletes it calling the `delete()` method.

Here is how it looks:



Dogs

Roger

Delete

Syd

Delete

Perfect

Delete

Add a new dog

Name

Submit

11. Adding authentication using Laravel Breeze

We don't want random people to come to the website and edit data.

We want people to log in first.

If logged out they will see the list of dogs.

If logged in they will have the ability to edit the list.

Laravel provides us built-in authentication support in the framework.

To make things even easier, it provides [Breeze](#), an application starter kit tool that will create what we need in no time. Breeze scaffolds user registration, login, password reset, profile page, dashboard... and even API authentication. It's great. For more advanced needs we also have [JetStream](#), but Breeze is easier to set up.

First, create a new Laravel project, so we start from a clean slate.

The first one was named **first**, so to continue the tradition we'll call this second project **second**:

```
composer create-project laravel/laravel second
```

Go into that folder:

```
cd second
```

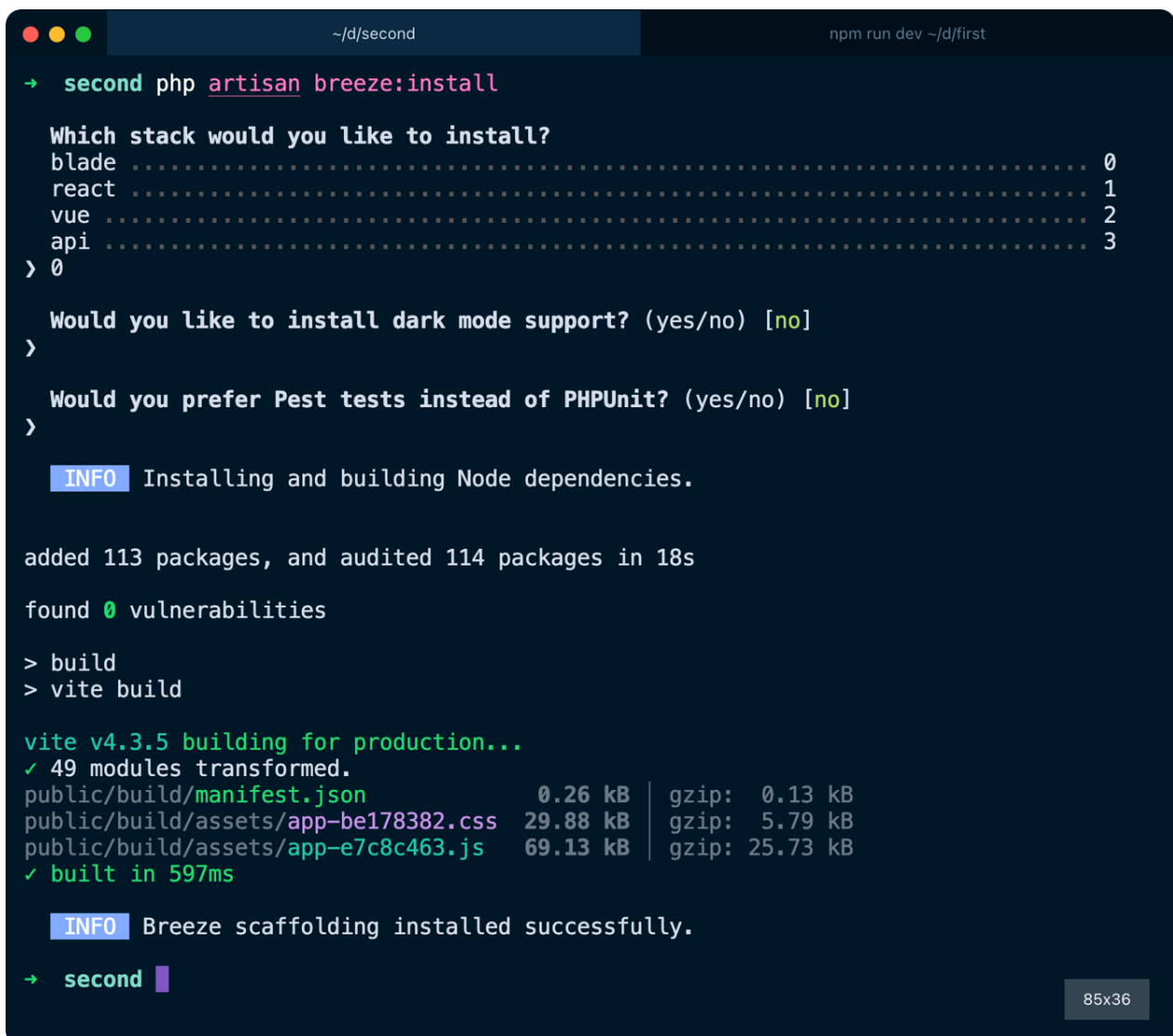
Install breeze using [composer](#):

```
composer require laravel/breeze --dev
```

Now run

```
php artisan breeze:install
```

and pick option 0, “blade”, and pick the default options for the other questions artisan asks you:

A terminal window with a dark background. The title bar shows three colored circles (red, yellow, green) on the left, and the path '~ /d/second' and 'npm run dev ~ /d/first' on the right. The terminal content shows the command 'second php artisan breeze:install' being executed. It prompts 'Which stack would you like to install?' with a list of options: blade (0), react (1), vue (2), and api (3). Option 0 is selected. It then asks 'Would you like to install dark mode support? (yes/no) [no]' and 'Would you prefer Pest tests instead of PHPUnit? (yes/no) [no]', both with 'no' selected. An 'INFO' message states 'Installing and building Node dependencies.' followed by 'added 113 packages, and audited 114 packages in 18s' and 'found 0 vulnerabilities'. It shows commands '> build' and '> vite build'. A green message says 'vite v4.3.5 building for production...' followed by '✓ 49 modules transformed.' and a table of file sizes. The table has two columns: file names and their sizes in kB, with a sub-column for gzipped sizes. The files listed are 'public/build/manifest.json' (0.26 kB, gzipped 0.13 kB), 'public/build/assets/app-be178382.css' (29.88 kB, gzipped 5.79 kB), and 'public/build/assets/app-e7c8c463.js' (69.13 kB, gzipped 25.73 kB). It ends with '✓ built in 597ms' and another 'INFO' message: 'Breeze scaffolding installed successfully.' The prompt 'second' is shown at the bottom left, and '85x36' is in a box at the bottom right.

```
→ second php artisan breeze:install

Which stack would you like to install?
blade ..... 0
react ..... 1
vue ..... 2
api ..... 3
> 0

Would you like to install dark mode support? (yes/no) [no]
>

Would you prefer Pest tests instead of PHPUnit? (yes/no) [no]
>

INFO Installing and building Node dependencies.

added 113 packages, and audited 114 packages in 18s
found 0 vulnerabilities

> build
> vite build

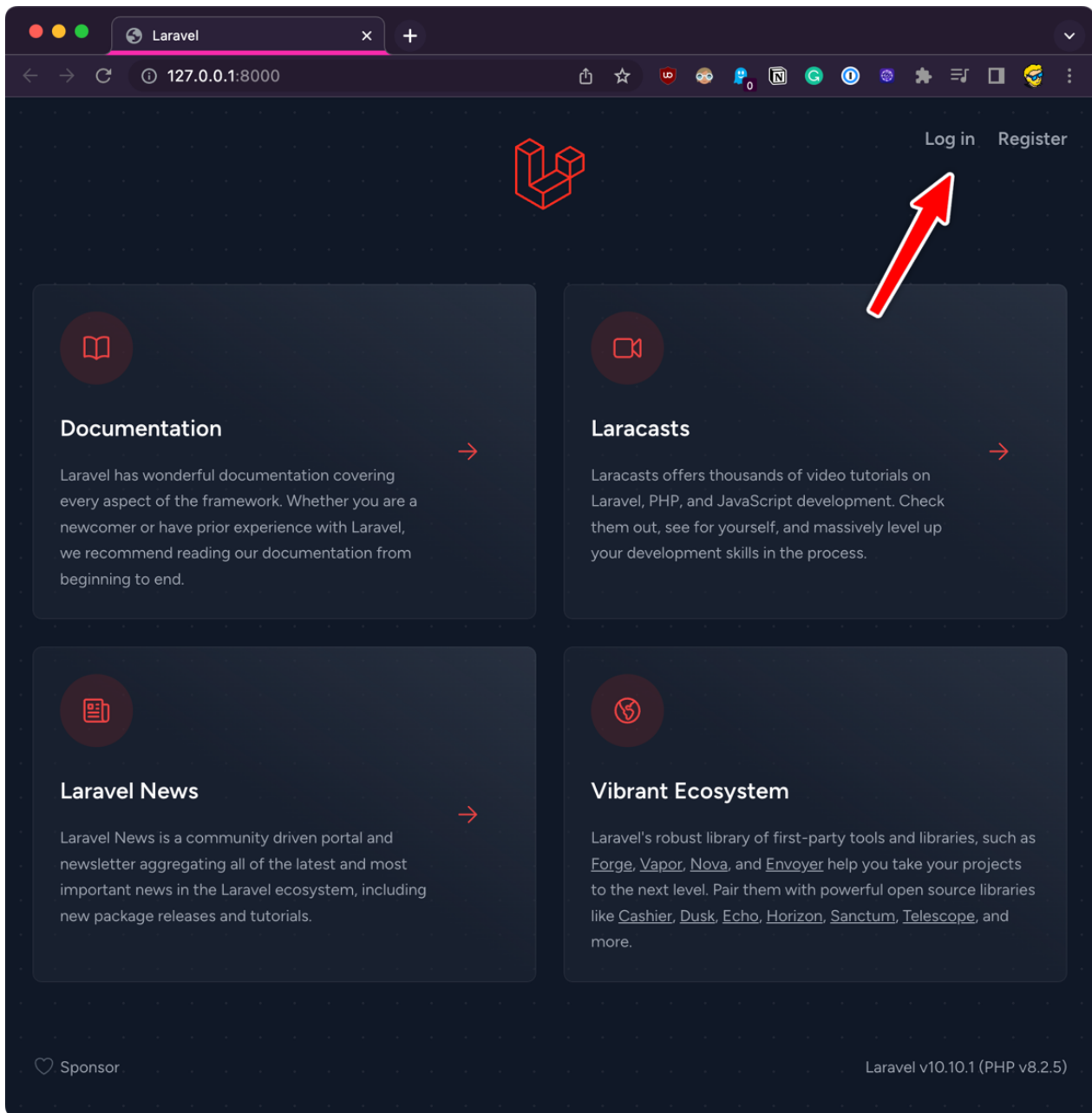
vite v4.3.5 building for production...
✓ 49 modules transformed.
public/build/manifest.json    0.26 kB | gzip: 0.13 kB
public/build/assets/app-be178382.css 29.88 kB | gzip: 5.79 kB
public/build/assets/app-e7c8c463.js 69.13 kB | gzip: 25.73 kB
✓ built in 597ms

INFO Breeze scaffolding installed successfully.

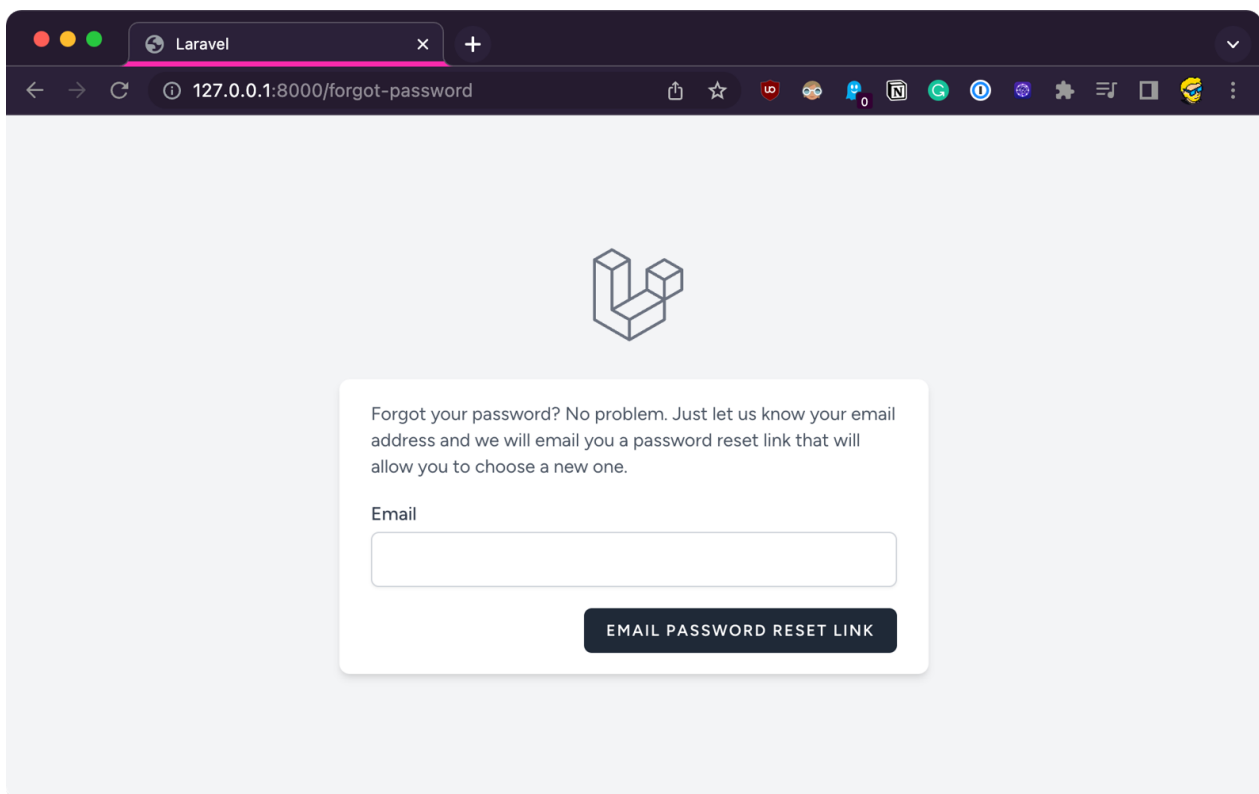
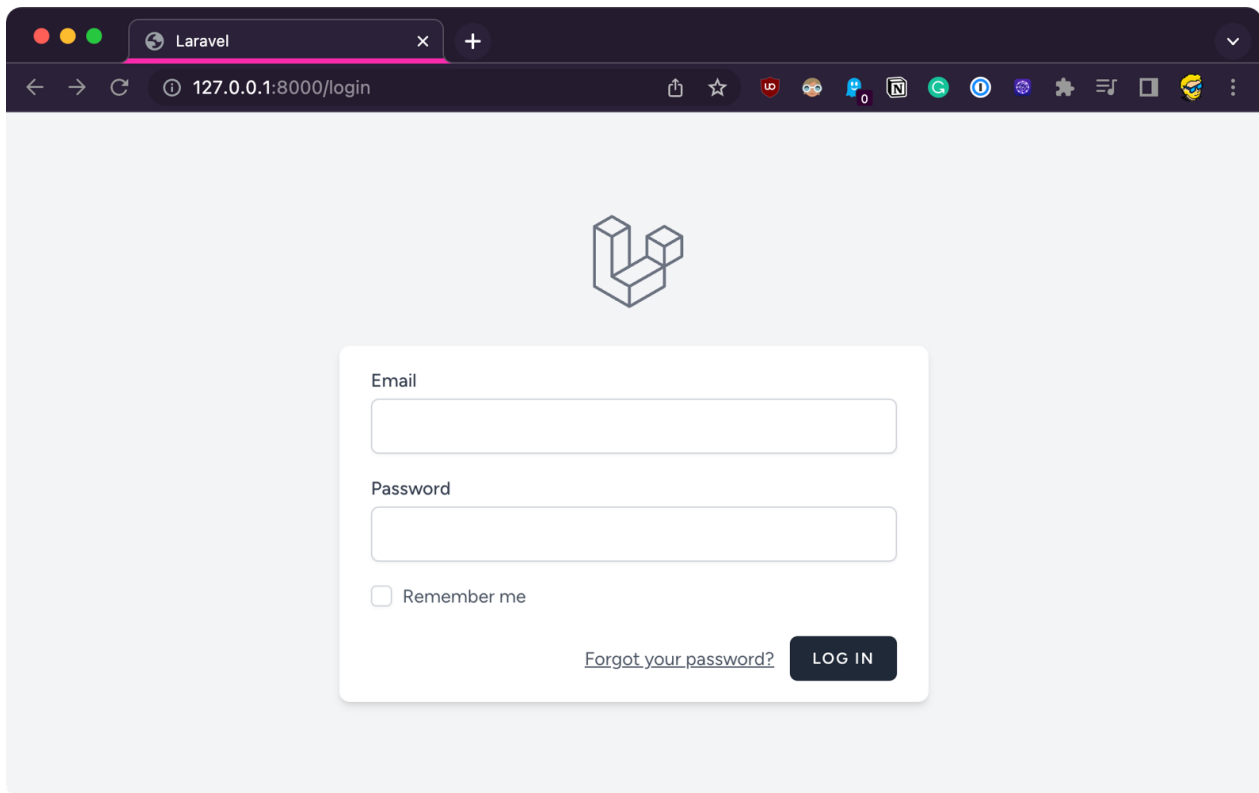
→ second
```

Now you can run `php artisan serve` and go to <http://127.0.0.1:8000/>.

You’ll see the “Log in” and “Register” links:



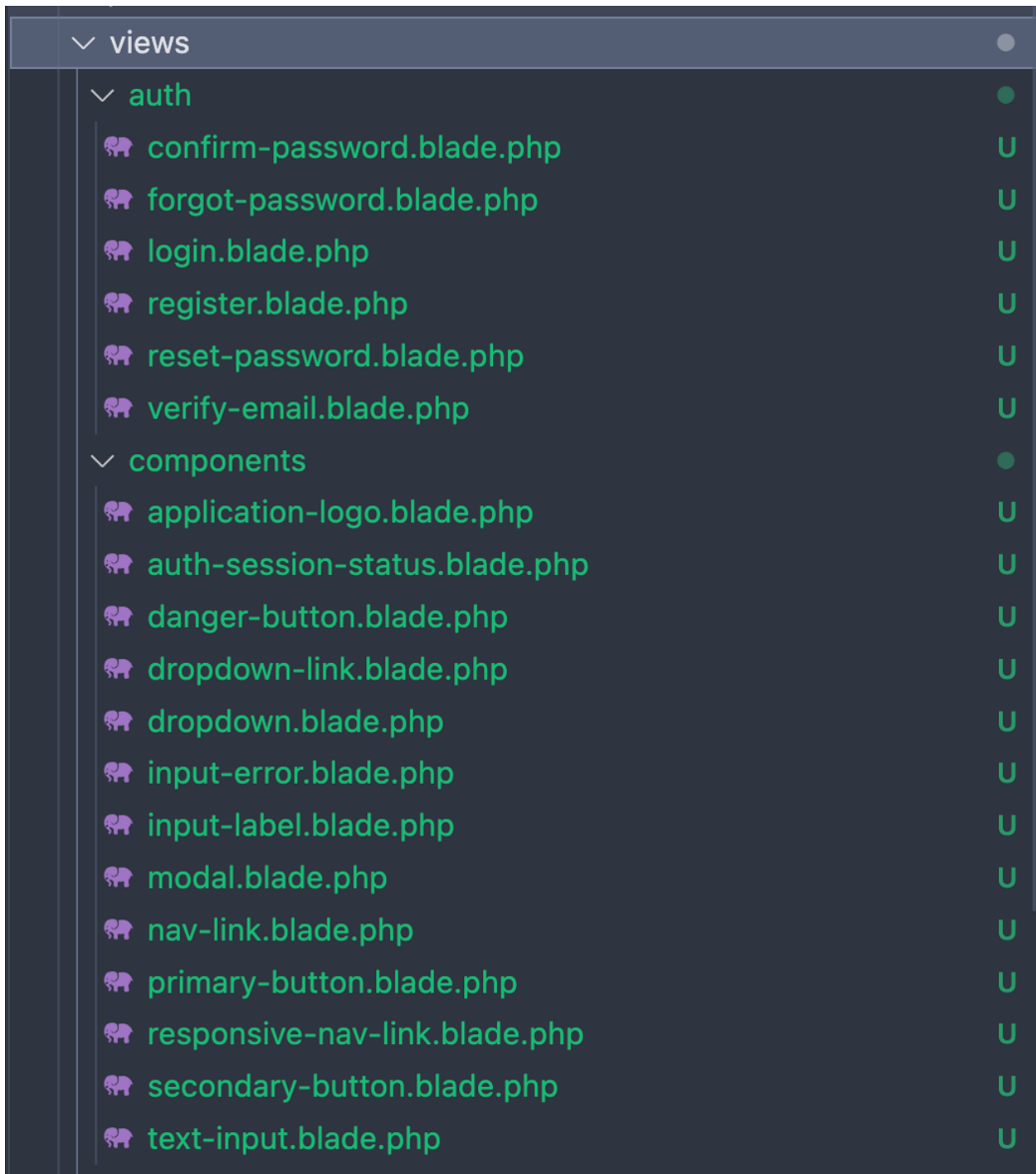
All the login functionality is working out of the box:



Laravel added a *ton* of resources to achieve this.

Easily days of work for a developer, and it's battle-tested code which you don't want to write yourself, as it's a quite important and needs to be well tested for security issues.

I recommend you take a look at the file structure and compare it to the first project. Lots of new stuff has been added, for example views:



But before we can go on, we have to set up the database for this project, doing what we did in the first one. We go to the `.env` file and comment those lines:

```
# DB_CONNECTION=mysql
# DB_HOST=127.0.0.1
# DB_PORT=3306
# DB_DATABASE=laravel
# DB_USERNAME=root
# DB_PASSWORD=
```

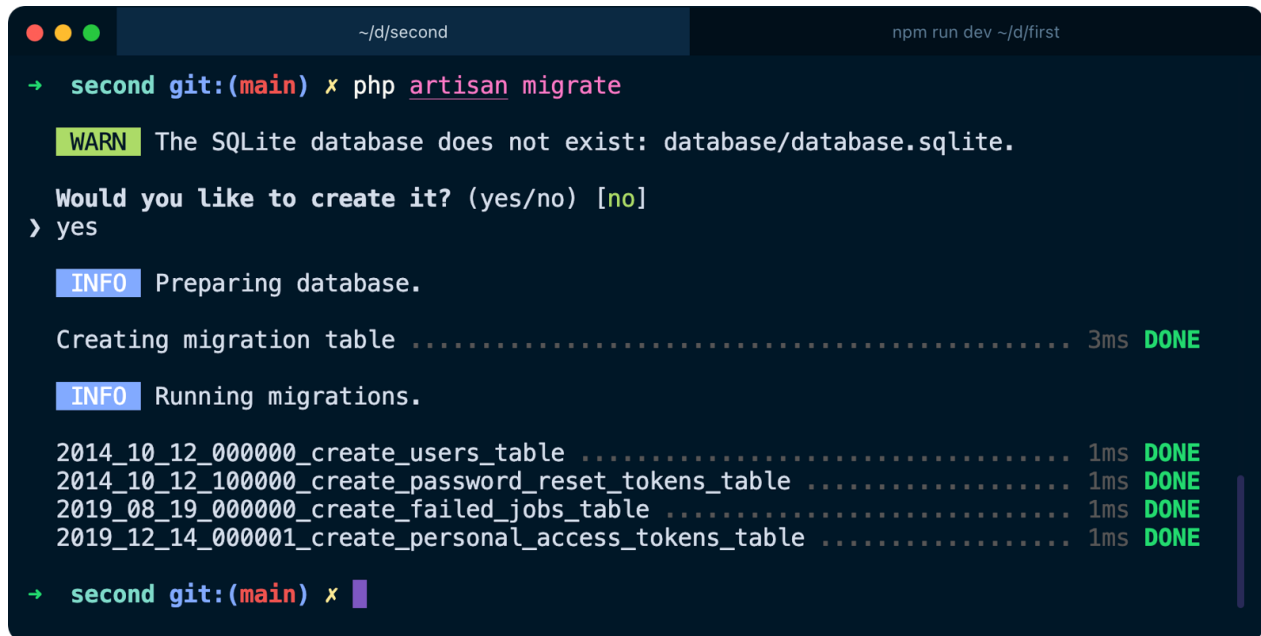
and add

```
DB_CONNECTION=sqlite
```

to configure the SQLite database.

Now run

```
php artisan migrate
```



```
→ second git:(main) x php artisan migrate

[WARN] The SQLite database does not exist: database/database.sqlite.

Would you like to create it? (yes/no) [no]
> yes

[INFO] Preparing database.

Creating migration table ..... 3ms DONE

[INFO] Running migrations.

2014_10_12_000000_create_users_table ..... 1ms DONE
2014_10_12_100000_create_password_reset_tokens_table ..... 1ms DONE
2019_08_19_000000_create_failed_jobs_table ..... 1ms DONE
2019_12_14_000001_create_personal_access_tokens_table ..... 1ms DONE

→ second git:(main) x
```

In another terminal folder, run `npm install` followed by `npm run dev`, which is a long running process you'll keep running alongside `php artisan serve` (Warning: just make sure you're running those in the `second` folder and not the `first` project, I just spent 15 minutes trying to figure out why I had a problem).

The Blade templates provided by Breeze use Tailwind CSS, and the setup of Tailwind was done automatically when we ran `php artisan breeze:install`

As you can see we already have a `tailwind.config.js` file.

Now you can open `resources/views/welcome.blade.php` and look at all that content. For the sake of simplicity, swap everything in that file with this trimmed-down version:

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
  <meta name="viewport"
    content="width=device-width, initial-scale=1.0">
  @vite('resources/css/app.css')
</head>

<body class="p-4">

  @if (Route::has('login'))
```

```

<div class="text-right">
    @auth
        <a href="{{ url('/dashboard') }}" class="">Dashboard</a>
    @else
        <a href="{{ route('login') }}" class="">Log in</a>
        @if (Route::has('register'))
            <a href="{{ route('register') }}" class="ml-4">
                Register</a>
        @endif
    @endauth
</div>
@endif

<h1 class="pb-2 mb-3 font-bold border-b border-b-gray-300">
    Dogs
</h1>

<div>
    @auth
        <p>Logged in</p>
    @endauth

    @guest
        <p>Not logged in</p>
    @endguest
</div>

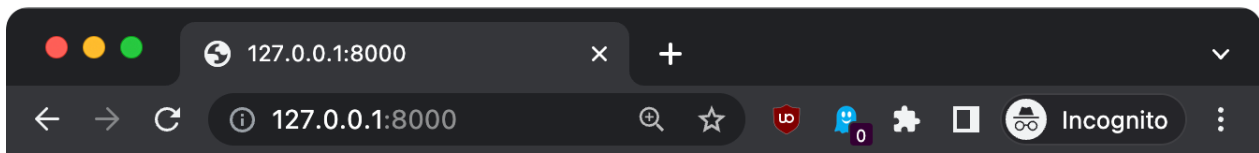
</body>

</html>

```

`@auth` / `@endauth` and `@guest` / `@endguest` are two Blade directives that allow you to show content (or not) depending on the logged in state.

This should be the result in the browser:



[Log in](#) [Register](#)

Dogs

Not logged in

Click the Register link to create a new account:



Name

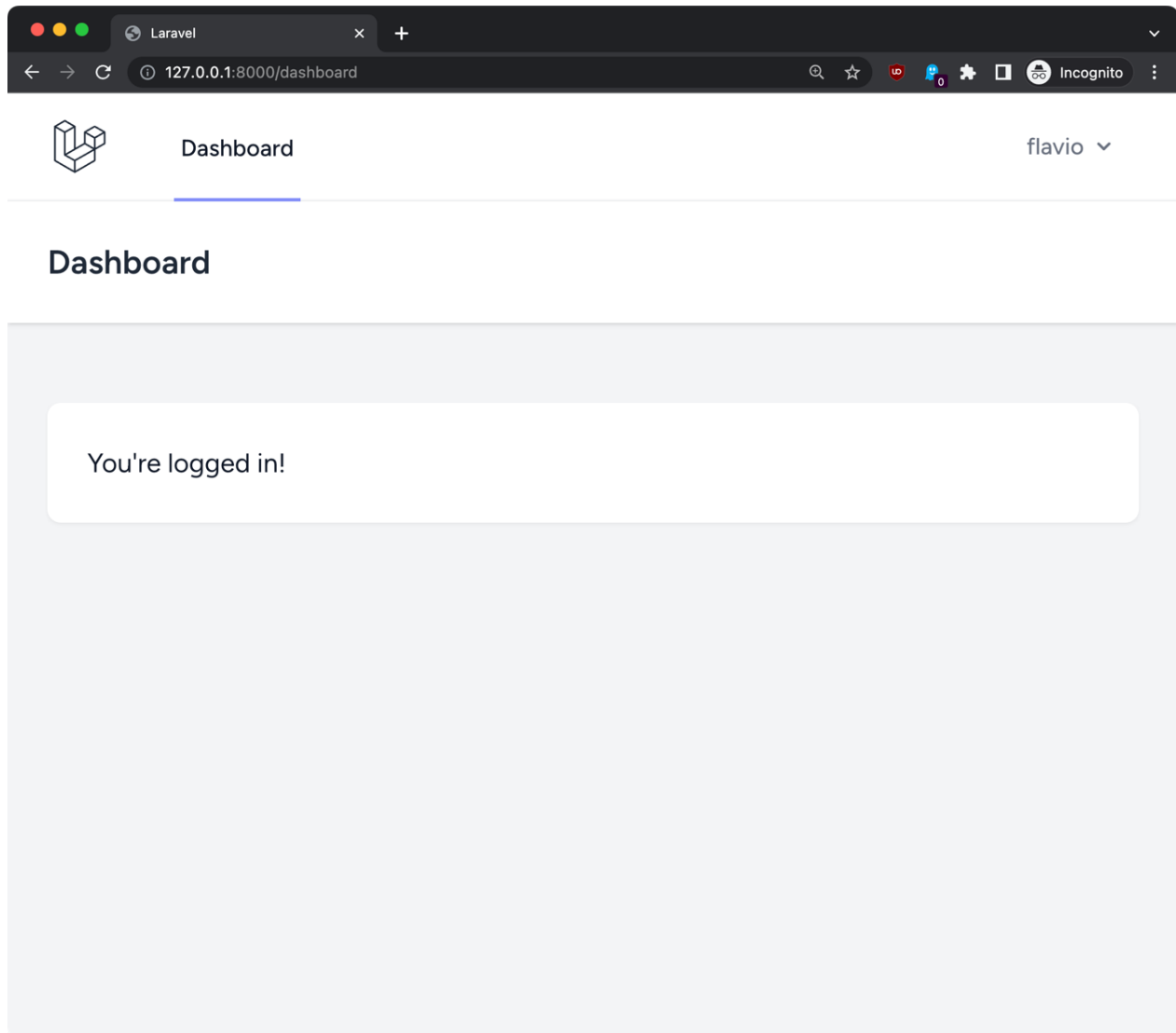
Email

Password

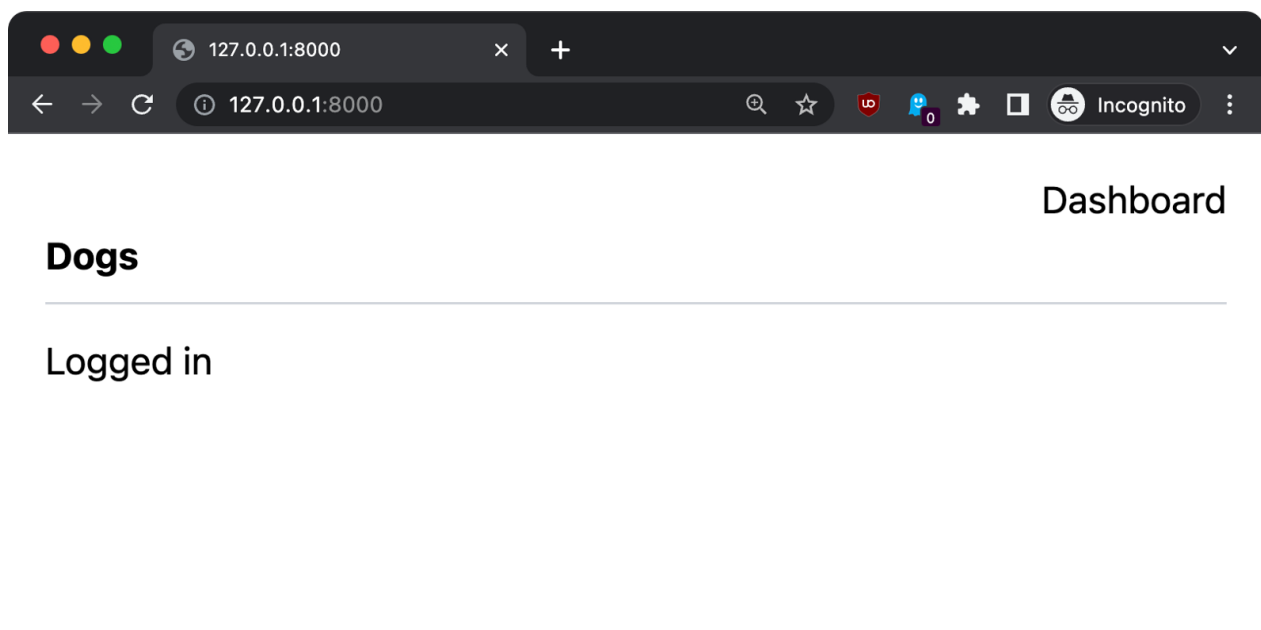
Confirm Password

[Already registered?](#) [REGISTER](#)

Create an account and you will be shown a dashboard page at the `/dashboard` route:



If you go back to the home, you will see the page in the logged in state:



12. Only authenticated users can add items to the database

Now let's re-implement what we did in the first project but this time we show the dogs list when logged out, but we'll only allow logged in users to modify the data.

First we create a new migration:

```
php artisan make:migration create_dogs_table
```

Open the newly created migration file, in my case `database/migrations/2023_05_12_164831_create_dogs_table.php`

```
<?php
```

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('dogs', function (Blueprint $table) {
            $table->id();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('dogs');
    }
};
```

We just modify the migration a little adding a name for the dog:

```
Schema::create('dogs', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->timestamps();
});
```

Save the file, go back to the terminal, run `php artisan migrate`

```

→ second git:(main) php artisan migrate

INFO Running migrations.

2023_05_12_164831_create_dogs_table ..... 2ms DONE

→ second git:(main) █

```

Now we scaffold the Dog model:

```
php artisan make:model Dog
```

Go to routes/web.php.

At the top, add

```
use App\Models\Dog;
```

then find the / route:

```
Route::get('/', function () {
    return view('welcome');
});
```

and change it to this to retrieve the dogs list and pass it to the view, which we label `index`:

```
Route::get('/', function () {
    $dogs = Dog::all();
    return view('welcome', ['dogs' => $dogs]);
})->name('index');
```

Now in `resources/views/welcome.blade.php` we can loop over the dogs array using `@foreach` like this:

```

<h1 class="pb-2 mb-3 font-bold border-b border-b-gray-300">
    Dogs
</h1>

<div>
    @foreach ($dogs as $dog)
        <li class="flex mb-1">
            <span class="flex-1">{{ $dog->name }}</span>
        </li>
    @endforeach

    @auth
        <p>Logged in</p>
    @endauth

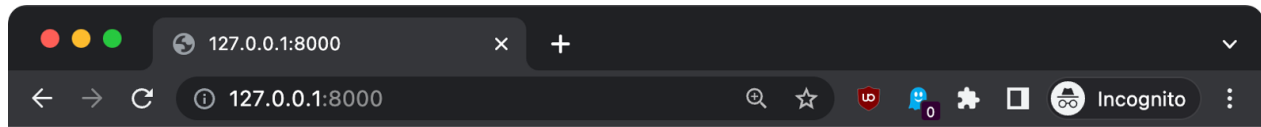
```

```

    @guest
    <p>Not logged in</p>
  @endguest
</div>

```

If you refresh the home you'll see nothing changed because we don't have any dog in the list.



Dogs

Logged in

We can actually show an “empty state” using @forelse.

Instead of

```

@foreach ($dogs as $dog)
  <li class="flex mb-1">
    <span class="flex-1">{{ $dog->name }}</span>
  </li>
@endforeach

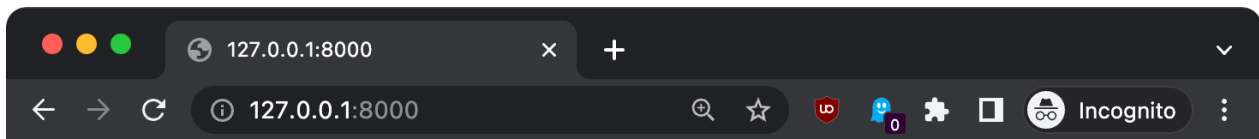
```

use this:

```

@forelse ($dogs as $dog)
  <li class="flex mb-1">
    <span class="flex-1">{{ $dog->name }}</span>
  </li>
@empty
  <p>No dogs yet</p>
@endforelse

```



Dashboard

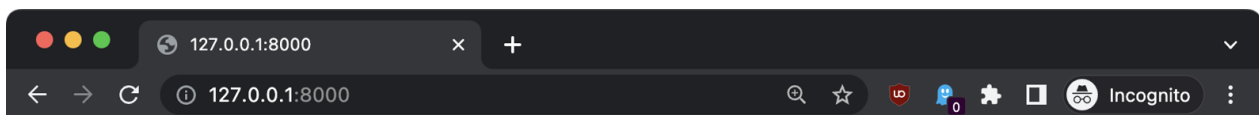
Dogs

No dogs yet
Logged in



We don't have dogs in the table yet, but you can open the database using [TablePlus](#) and insert data using this SQL query:

```
INSERT INTO "dogs" ("id", "name", "created_at", "updated_at") VALUES
('1', 'Roger', '2023-05-11 09:27:20', '2023-05-11 09:27:20'),
('2', 'Syd', '2023-05-11 09:29:52', '2023-05-11 09:29:52'),
('3', 'Botolo', '2023-05-11 09:29:57', '2023-05-11 09:29:57'),
('4', 'Zoe', '2023-05-11 09:30:12', '2023-05-11 09:30:12');
```



Dashboard

Dogs

Roger

Syd

Botolo

Zoe

Logged in

Now when we're logged in I want to display the form to add a new dog, and the delete button for each dog in the list.

First, inside the Dog model class we add the `name` to an array named `$fillable`:

```
protected $fillable = ['name'];
```

```
<?php
```

```
namespace App\Models;
```

```
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
```

```
class Dog extends Model
{
    use HasFactory;
    protected $fillable = ['name'];
}
```

We create a controller named DogController:

```
php artisan make:controller DogController
```

This created the app/Http/Controllers/DogController.php file:

```
<?php
```

```
namespace App\Http\Controllers;
```

```
use Illuminate\Http\Request;
```

```
class DogController extends Controller
{
    //
}
```

Add `use App\Models\Dog;` at the top, and add those 2 methods to the class, `create` and `delete`, as we did before, but this time both first check that the user is logged in:

```
<?php
```

```
namespace App\Http\Controllers;
```

```
use Illuminate\Support\Facades\Auth;
use Illuminate\Http\Request;
use App\Models\Dog;
```

```
class DogController extends Controller
{
    public function create(Request $request)
    {
        if (Auth::check()) {
            $this->validate($request, [
```

```

        'name' => 'required',
    ]);
    Dog::create($request->all());
}
return to_route('index');
}

public function delete($id)
{
    if (Auth::check()) {
        $dog = Dog::find($id);
        $dog->delete();
    }

    return to_route('index');
}
}

```

Ok now we need a route to add a new dog, and one to delete a dog. In `routes/web.php`, add:

```

use App\Http\Controllers\DogController;

//...

Route::post(
    '/dogs',
    [DogController::class, 'create']
)->name('dog.create');

Route::delete(
    '/dog/{id}',
    [DogController::class, 'delete']
)->name('dog.delete');

```

Now we can display the buttons to remove dogs in `resources/views/welcome.blade.php`:

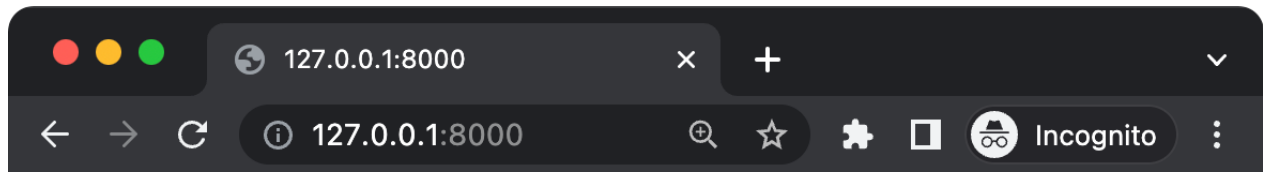
```

@forelse ($dogs as $dog)
    <li class="flex mb-1">
        <span class="flex-1">{{ $dog->name }}</span>
        @auth
            <form action="{{ route('dog.delete', $dog->id) }}"
                method="POST">
                @csrf
                @method('DELETE')
                <button type="submit" class="p-1 bg-gray-200 border
                    border-black">Delete</button>
            </form>
        @endauth
    </li>

```

```
@empty
  <p>No dogs yet</p>
@endforelse
```

We wrap it into `@auth` to make it only visible if logged in.



Dashboard

Dogs

Roger

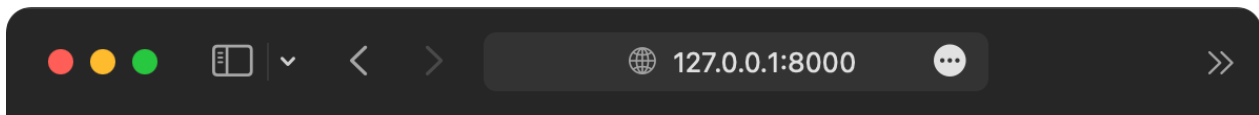
Delete

Syd

Delete

Try clicking one “delete” button, the corresponding row should disappear.

If logged out, here’s what you see:



[Log in](#) [Register](#)

Dogs

Roger

Syd

Now let's add the form to add a new dog. Before we used a partial, to see how you can use partials, but now let's just add it to the `welcome` template:

```
@auth
  <form method="post" action="{{ route('dog.create') }}">
    @csrf
    <h3 class="pb-2 mt-4 mb-3 font-bold border-b border-b-gray-300">
      Add a new dog</h3>
    <div class="flex">
      <div class="flex-1">
        <label>Name</label>
        <input type="text" name="name" id="name"
          class="p-1 border border-gray-200 ">
      </div>
      <input type="submit" name="send" value="Submit"
        class="p-1 bg-gray-200 border border-black
          cursor-pointer">
    </div>
  </form>
@endauth
```

Here's the full code for reference:

```
<!doctype html>
<html>

<head>
  <meta charset="utf-8">
```

```

<meta name="viewport"
  content="width=device-width, initial-scale=1.0">
@vite('resources/css/app.css')
</head>

<body class="p-4">

  @if (Route::has('login'))
    <div class="text-right">
      @auth
        <a href="{{ url('/dashboard') }}" class="">Dashboard</a>
      @else
        <a href="{{ route('login') }}" class="">Log in</a>
        @if (Route::has('register'))
          <a href="{{ route('register') }}" class="ml-4">
            Register</a>
        @endif
      @endauth
    </div>
  @endif

  <h1 class="pb-2 mb-3 font-bold border-b border-b-gray-300">
    Dogs
  </h1>

  <ul>
    @forelse ($dogs as $dog)
      <li class="flex mb-1">
        <span class="flex-1">{{ $dog->name }}</span>
        @auth
          <form action="{{ route('dog.delete', $dog->id) }}"
            method="POST">
            @csrf
            @method('DELETE')
            <button type="submit"
              class="p-1 bg-gray-200 border border-black">
              Delete</button>
          </form>
        @endauth
      </li>
    @empty
      <p>No dogs yet</p>
    @endforelse
  </ul>

  @auth
    <form method="post" action="{{ route('dog.create') }}">
      @csrf
      <h3 class="pb-2 mt-4 mb-3 font-bold border-b

```

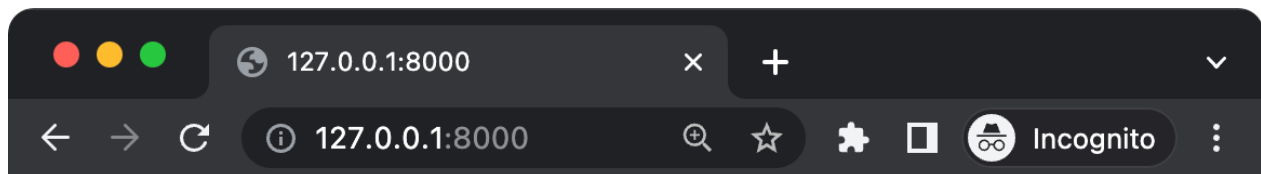
```

border-b-gray-300">Add a new dog</h3>
<div class="flex">
  <div class="flex-1">
    <label>Name</label>
    <input type="text" name="name" id="name"
      class="p-1 border border-gray-200 ">
  </div>
  <input type="submit" name="send" value="Submit"
    class="p-1 bg-gray-200 border border-black
      cursor-pointer">
  </div>
</form>
@endauth

</body>

</html>

```



Dashboard

Dogs

Roger

Delete

Add a new dog

Name

Submit

Try it, it should work!

13. Push the app code to GitHub

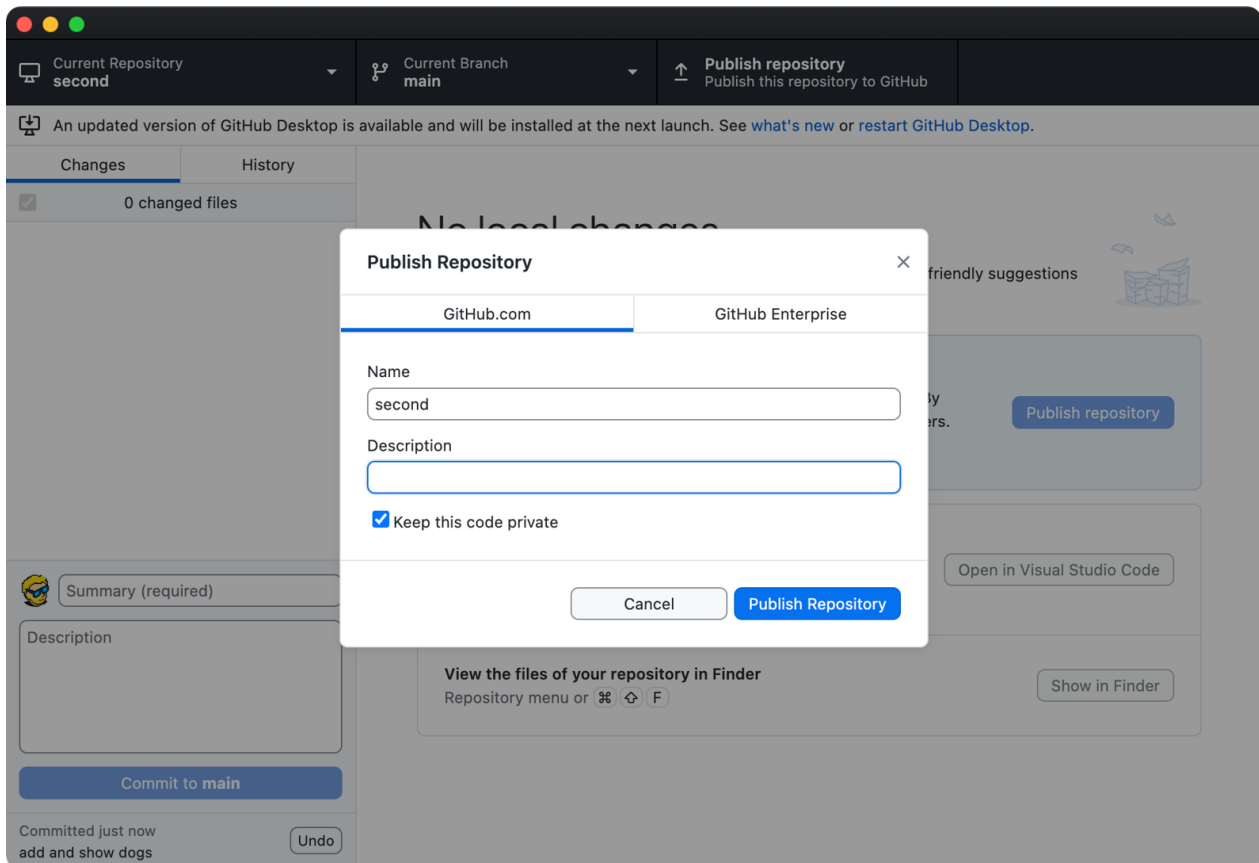
Before going to the next chapter, deployment, let's push the code to GitHub.

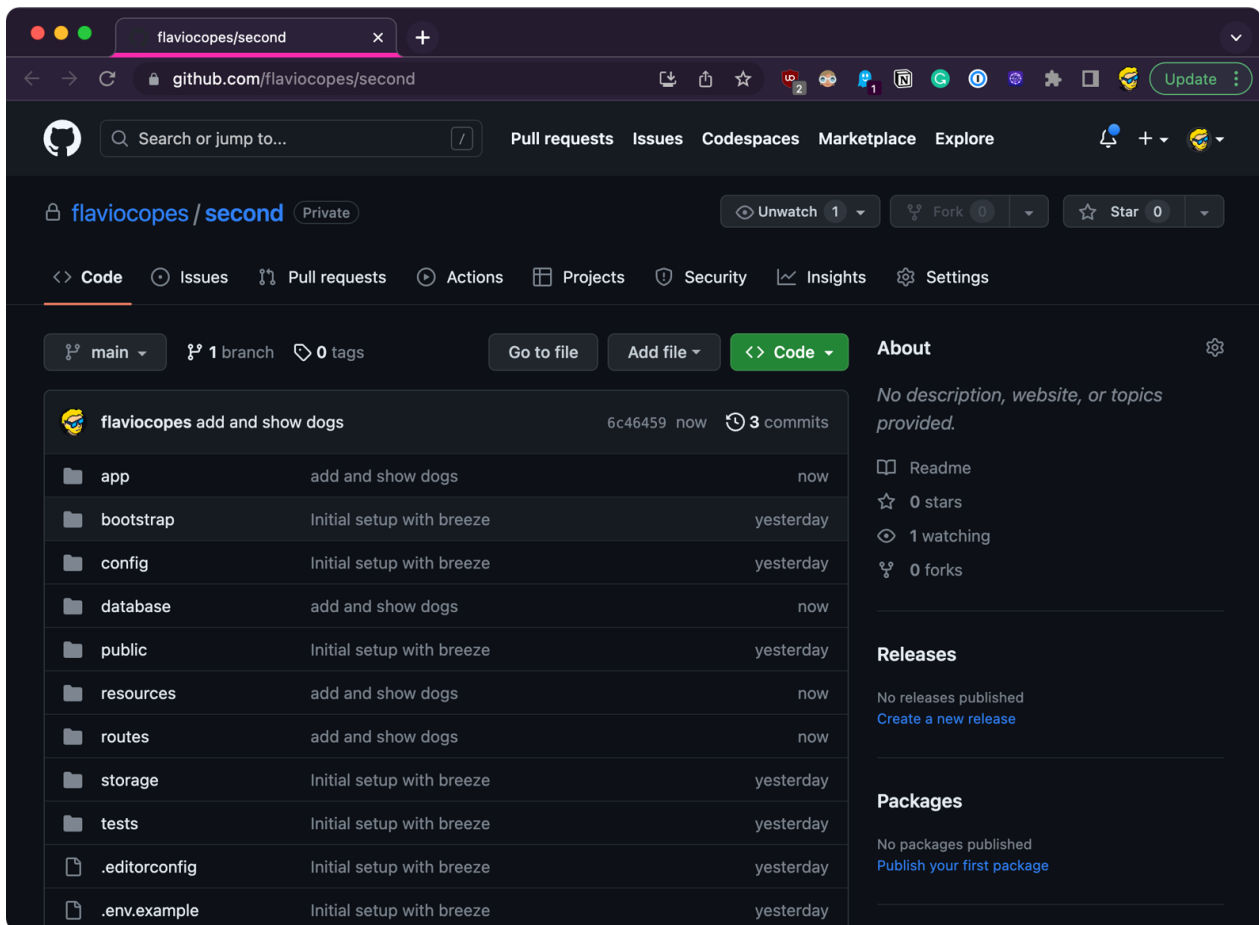
If you don't have a GitHub account yet, and you have no idea what is Git, check out

- <https://flaviocopes.com/git/>
- <https://flaviocopes.com/github/>

and the tutorial <https://flaviocopes.com/github-setup-from-zero/>

I want you to create a Git repository, push the repository to GitHub, so that you have the code up there, ready to be deployed:





14. Deployment

It's all fun and all, but now we want to deploy the application on a real server, on the Internet, so people can use it.

There are many different ways to deploy a Laravel application.

Probably the best one is using [Laravel Forge](#), the official deployment platform, combined with DigitalOcean.

When it comes to servers, and unless you like managing servers and you're actually an expert, I am a fan of investing some money and saving time instead.

Forge in particular is official, made by the core team of Laravel, lots and lots of people use it (they claim over 500,000 sites are powered by Forge), and we can trust that to work as expected.

Forge does not provide a server to you. But it's a service that connects to DigitalOcean and other VPS - virtual private server - providers like Hetzner, AWS, Vultr and more and it creates a server for you on that platform.

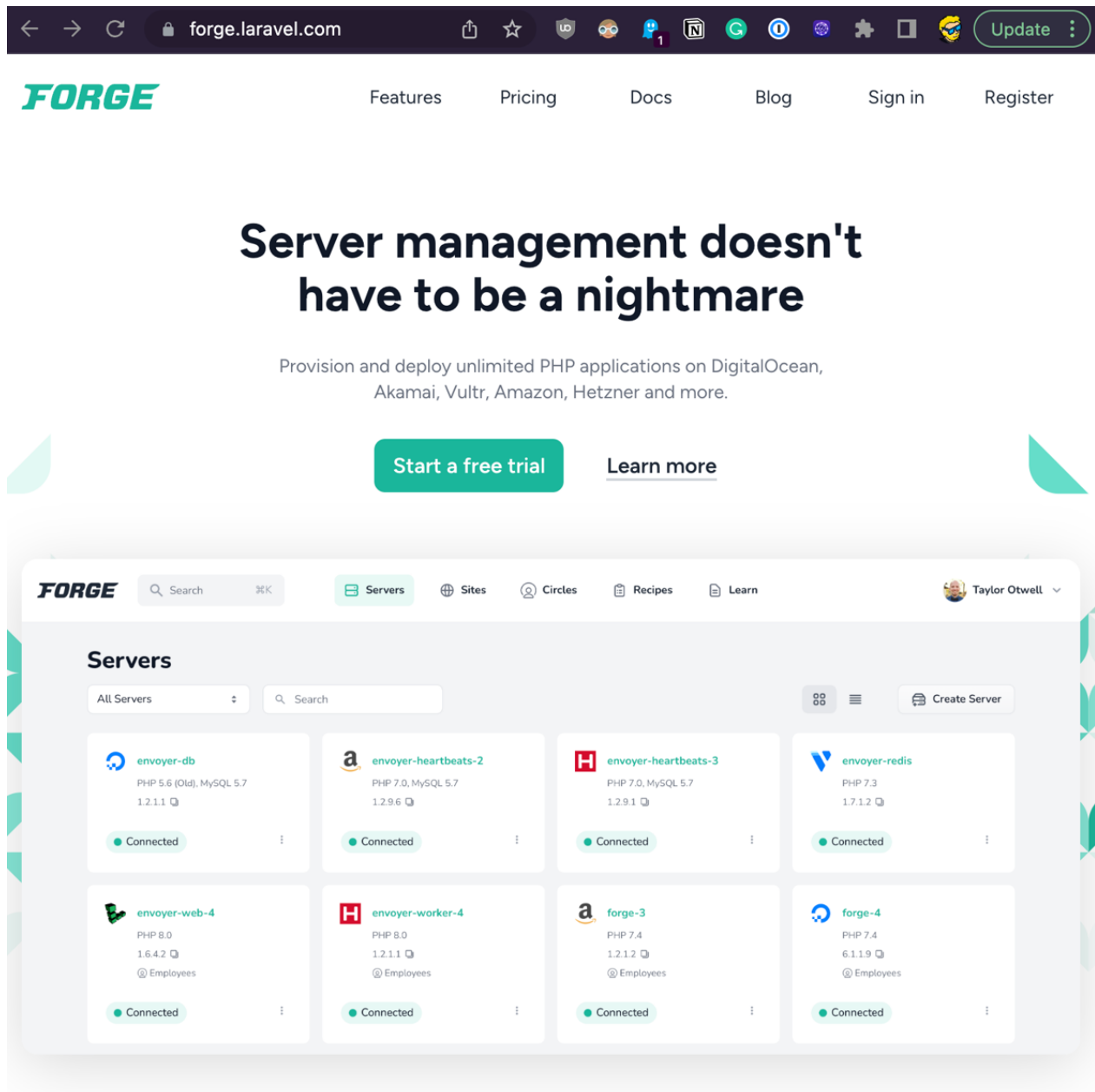
You could directly use a VPS, of course. Follow a tutorial, set everything up, invest hours into basically doing what Forge can do with a few simple steps. It's a matter of convenience.

And in the long run, Forge can upgrade PHP for example with a simple click. If that's left to you to manage, it's more complicated.

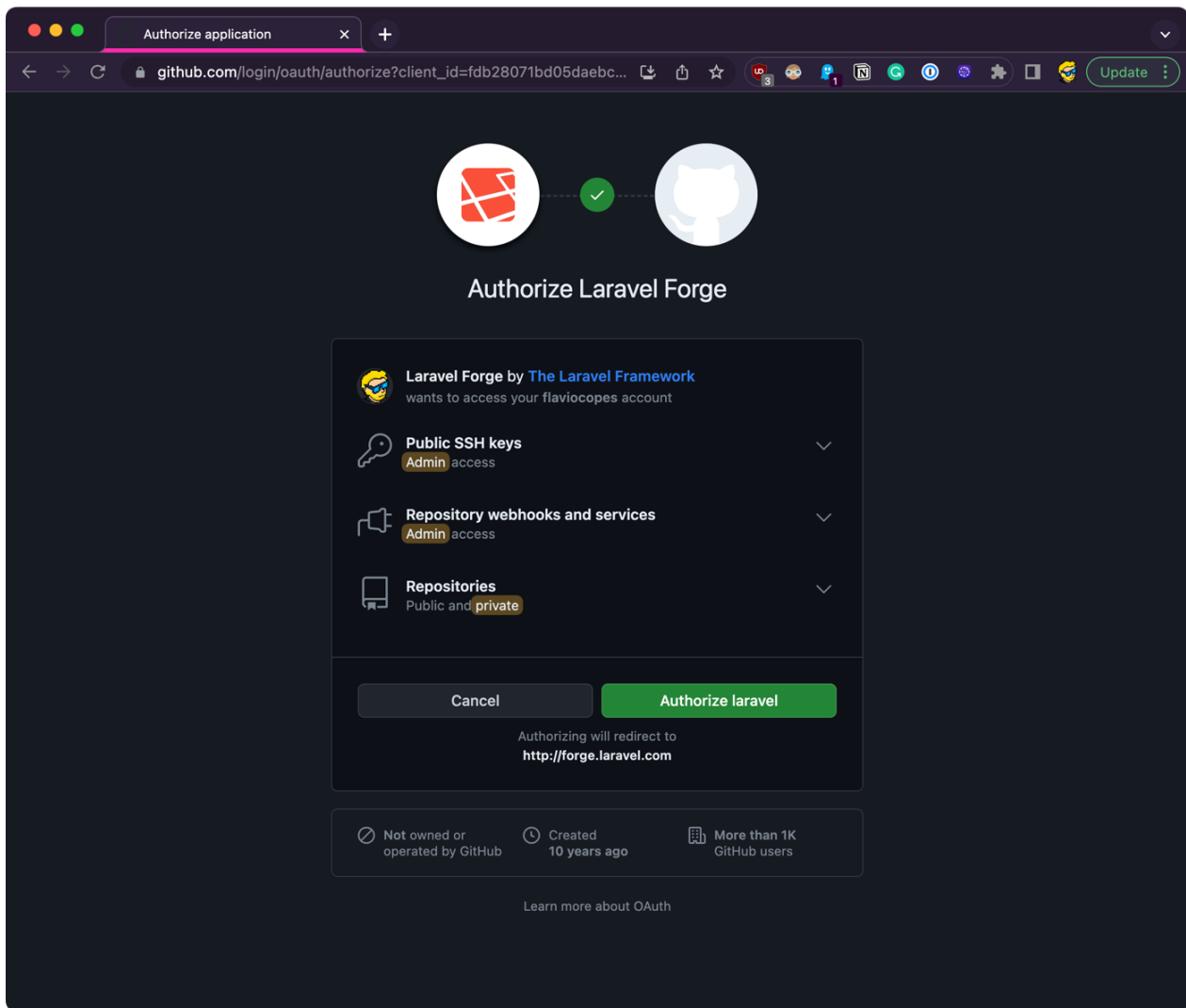
Anyway, pick your poison. Spend time and effort, or spend some money and focus on your app.

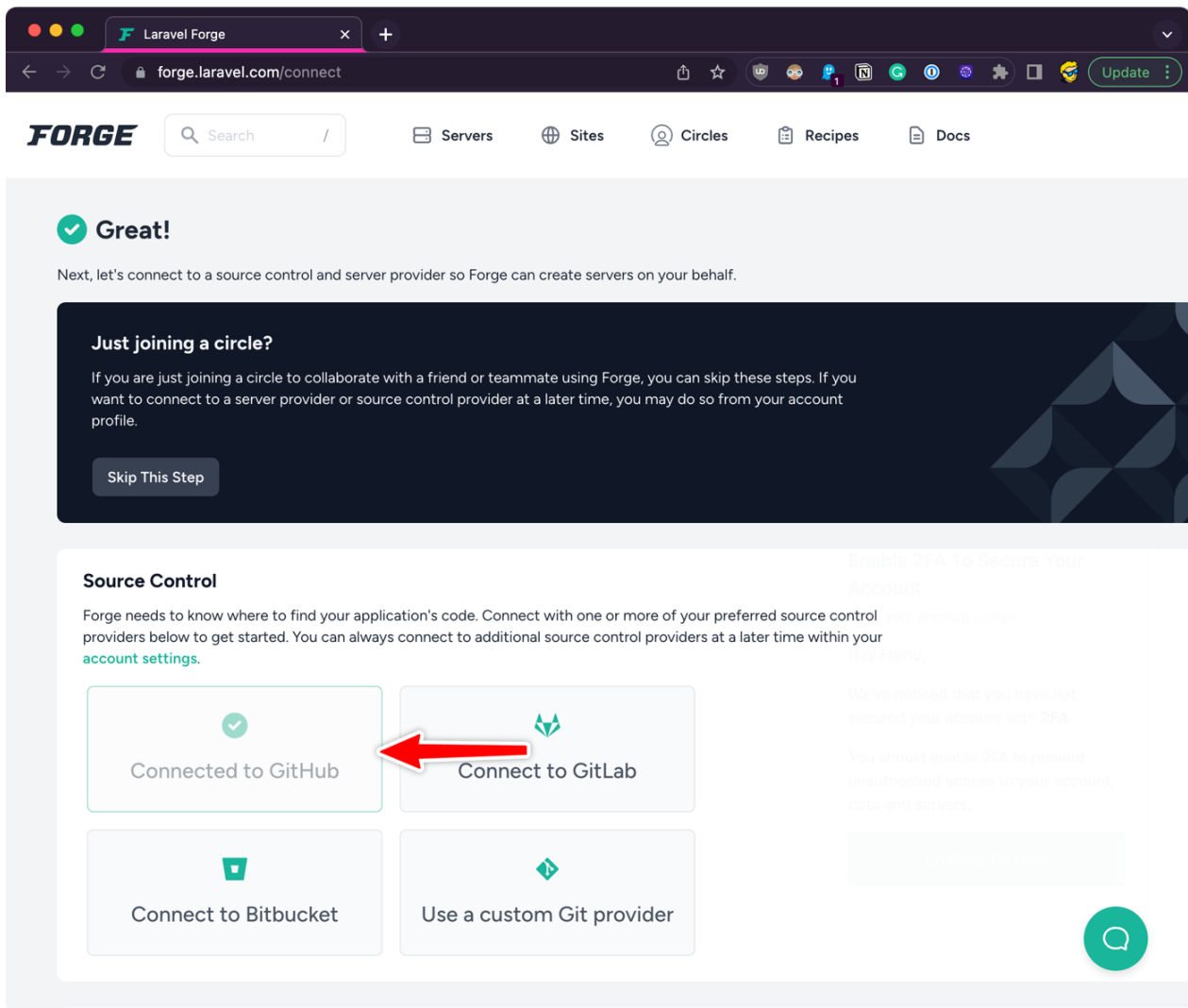
How much money? Not much, \$12/month. And it has a free trial.

Go to <https://forge.laravel.com/> and click the “Start a free trial” button:

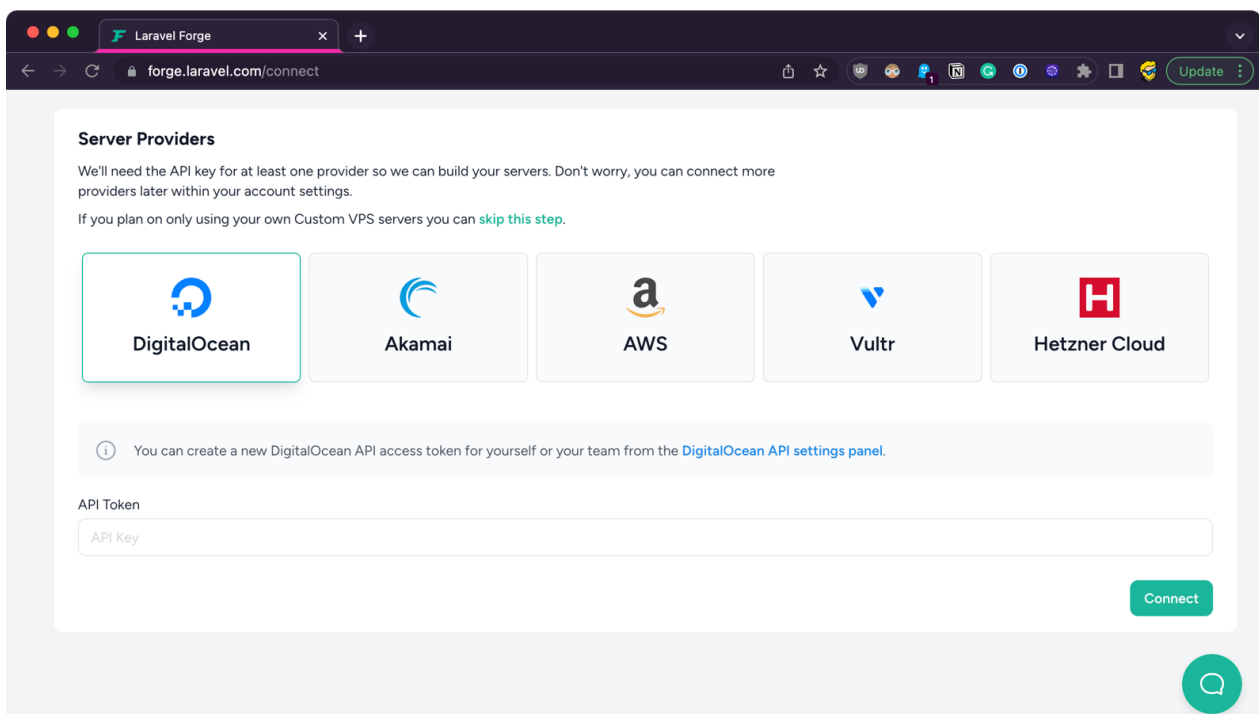


Once you’re in, click “Connect to GitHub” to connect Forge to GitHub so it can pull your code:





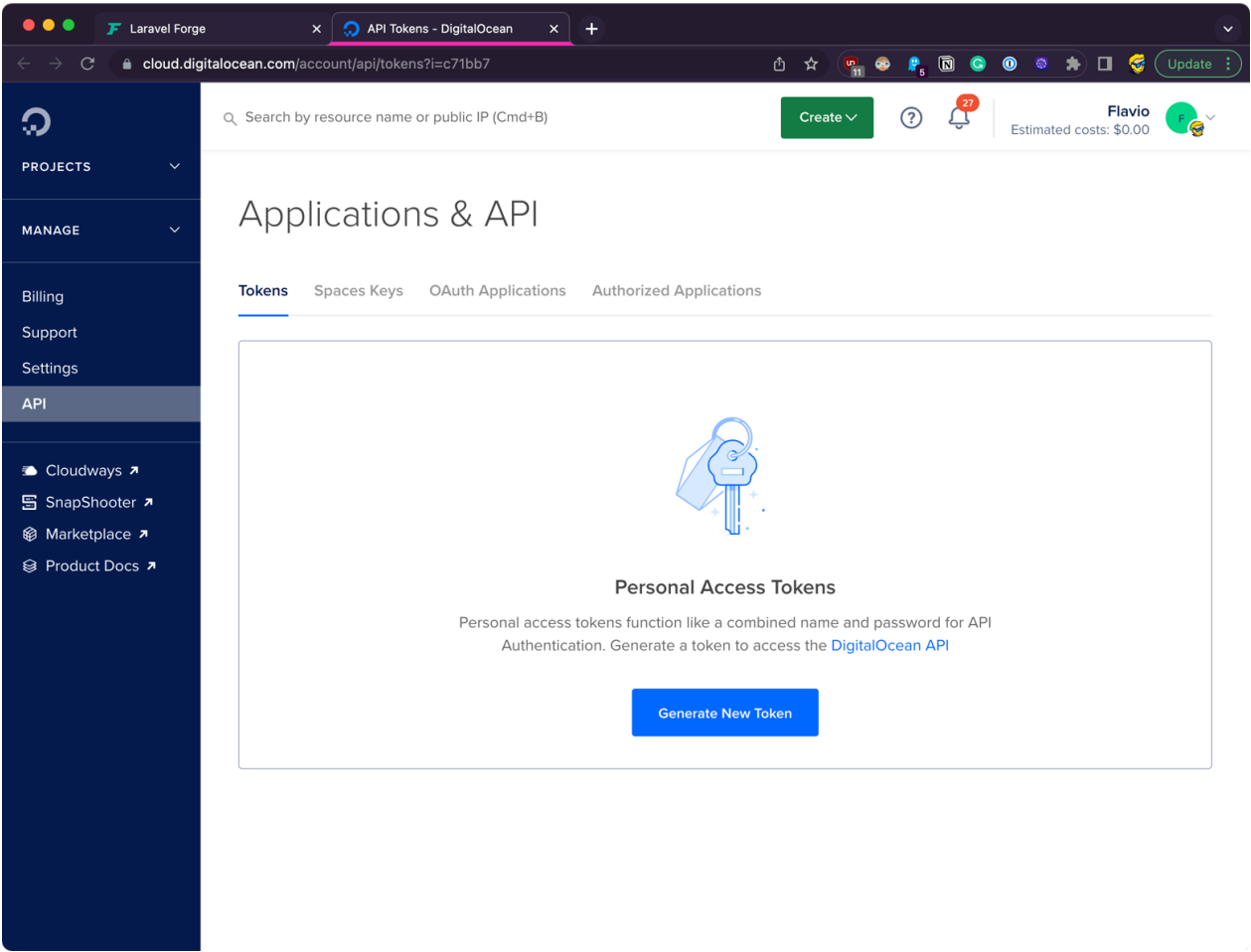
Now it's time to connect to a server provider.

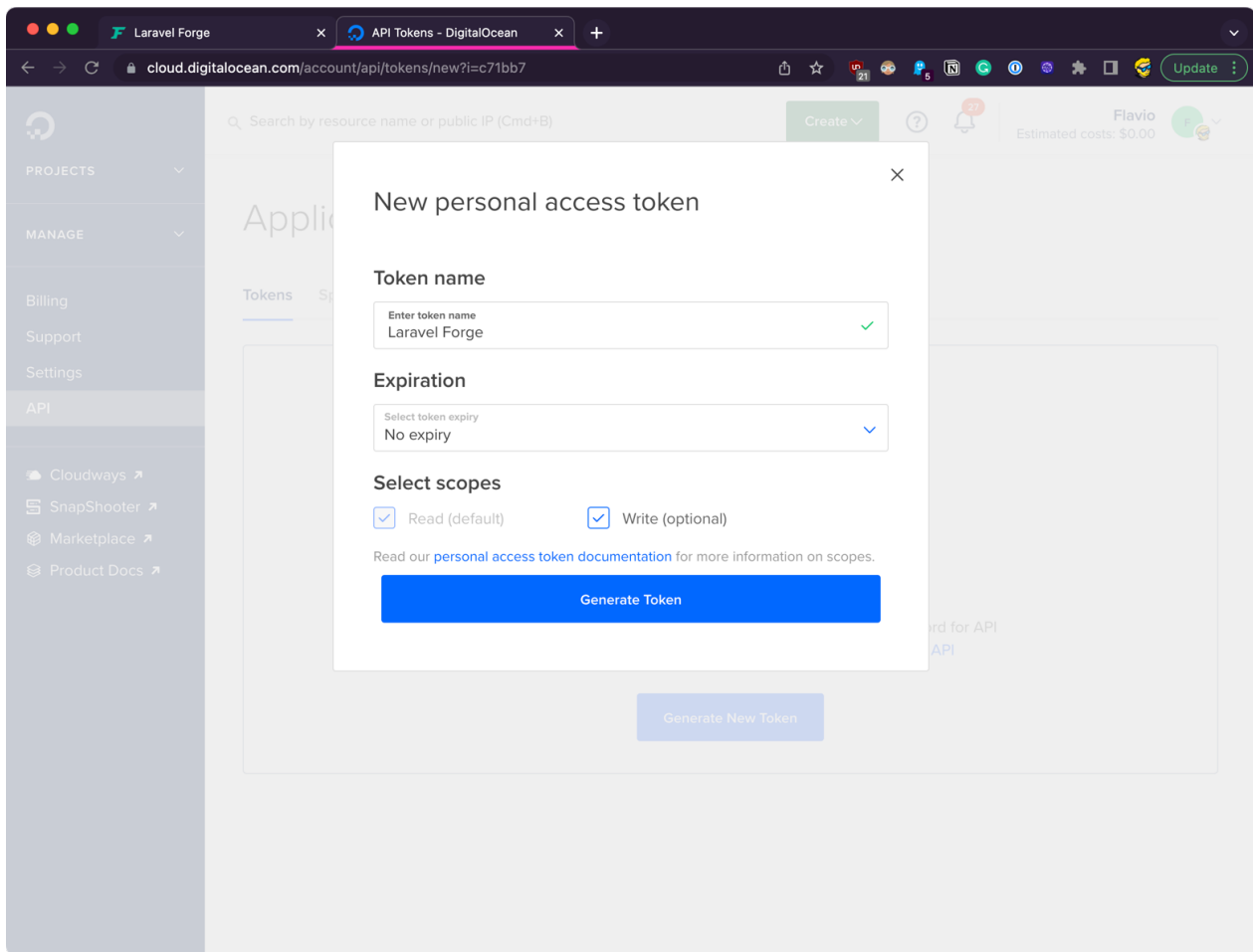


I pick DigitalOcean.

If you're unsure, DigitalOcean gives you free credits, so you can try it out.

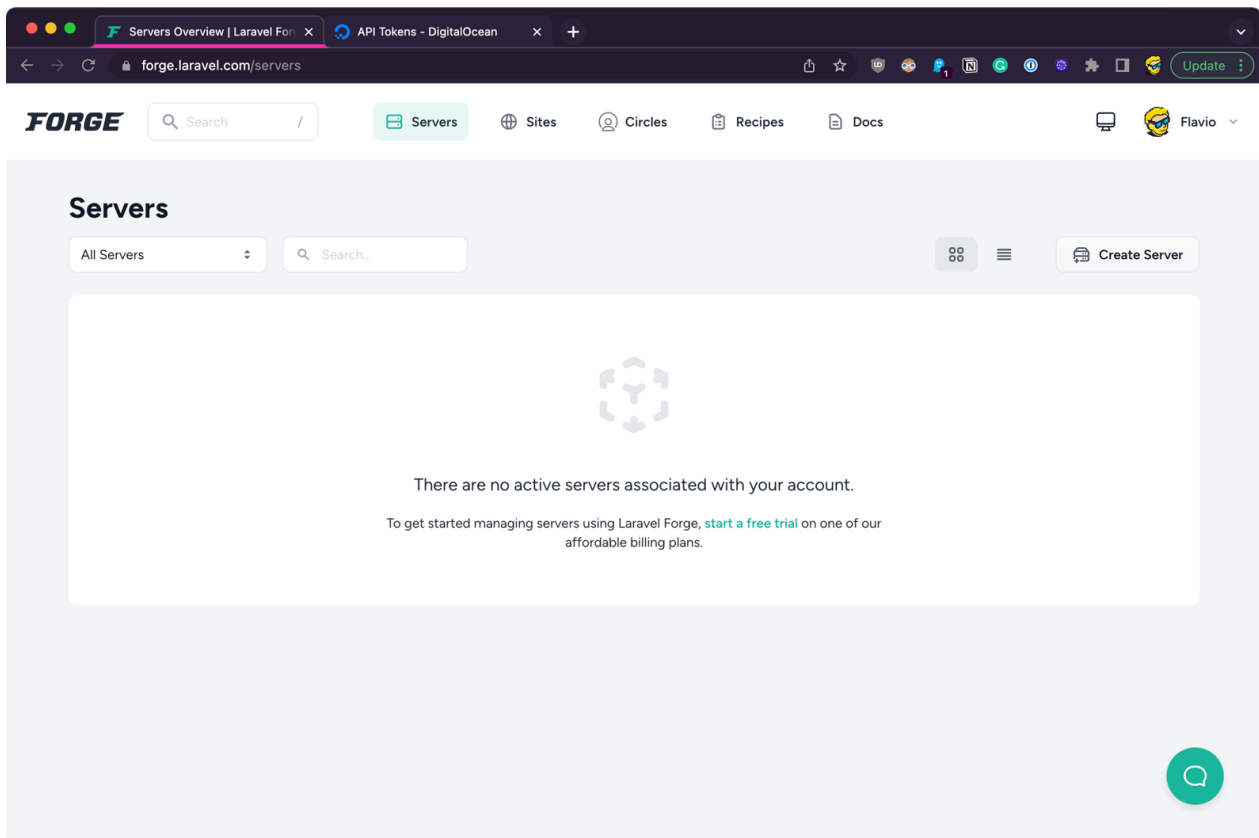
I click the link to create an API token and I generate one



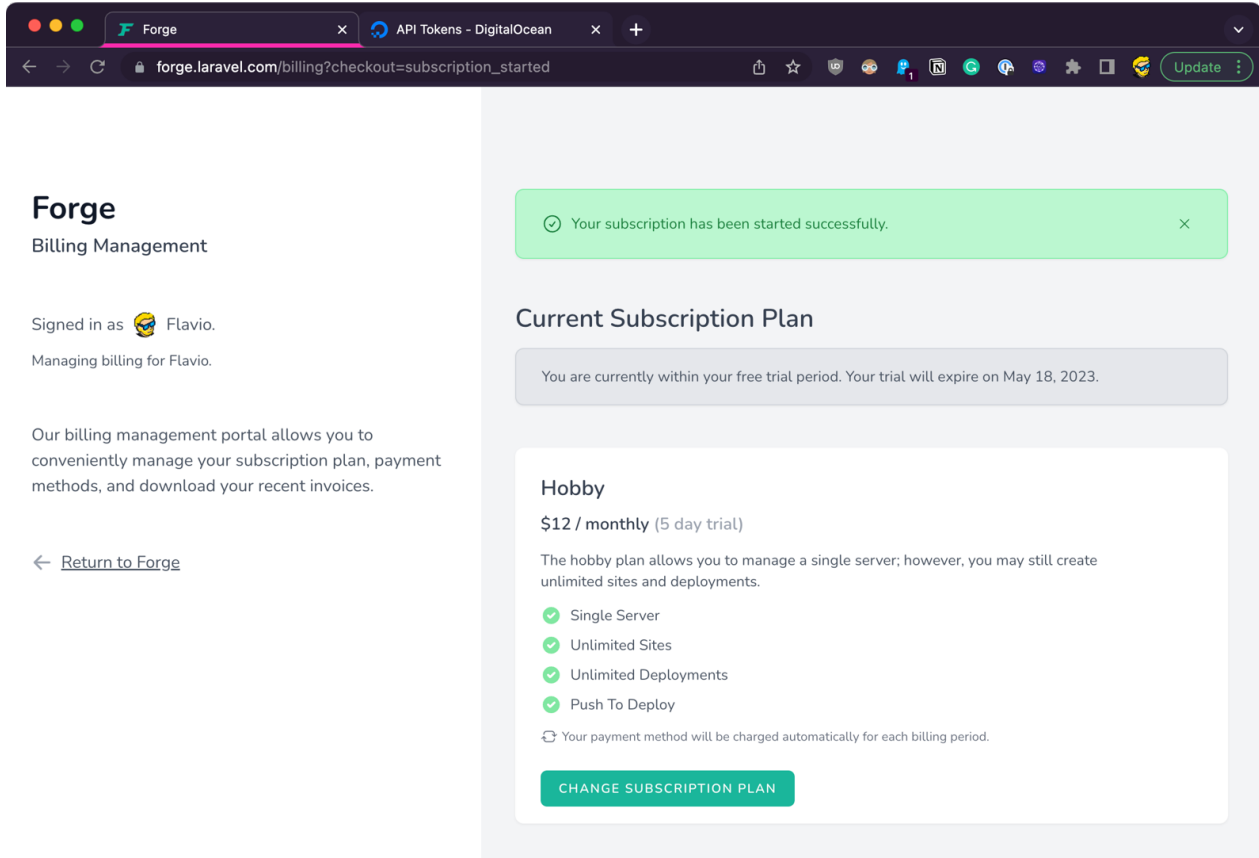


and finally I copy the code to Forge.

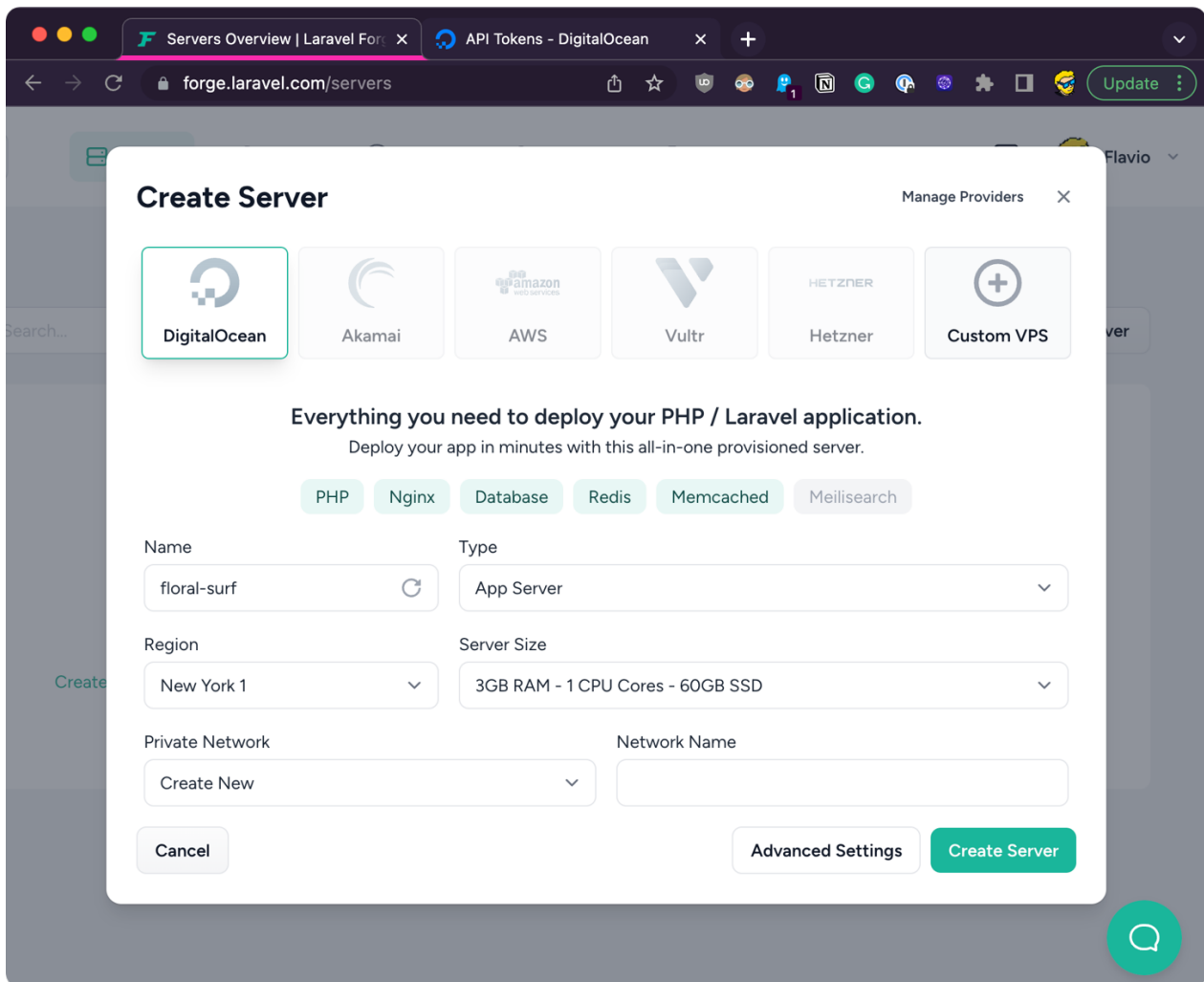
I now have access to the servers dashboard



Here's where you need to create the subscription. I picked the Hobby plan for \$12/m, with a free trial of 5 days:



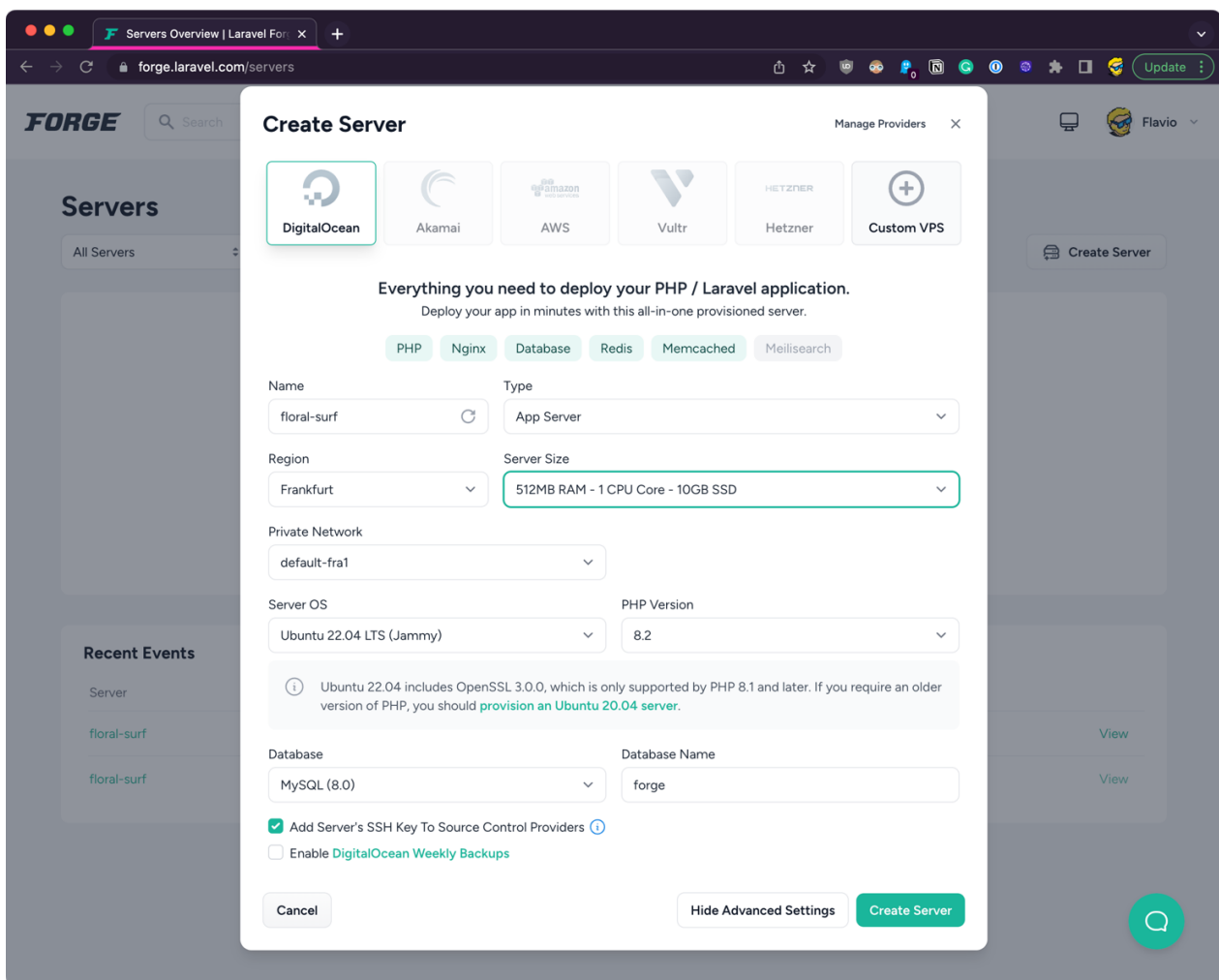
Now back to the servers page, I can create a new server:



Here you can change the type of server you want to create. Pick “App Server” as it contains all you’re going to need.

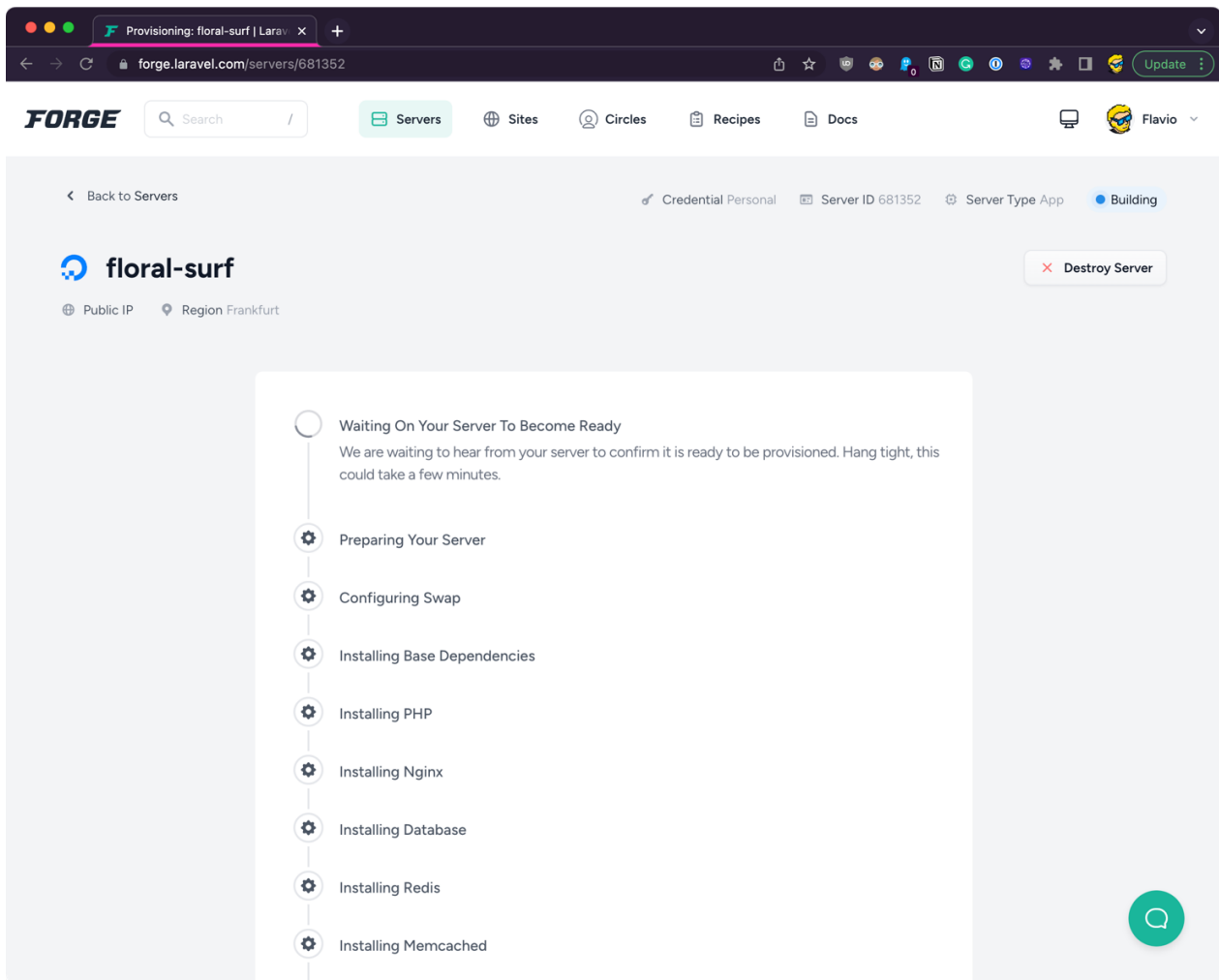
Pick a region near you, and pick a modest server size, so you can save on server costs until someone actually uses your app (you can always upgrade the server later via the DigitalOcean panel)

In **Advanced Settings** you can configure more details such as the Operating System, Database and PHP version:



I picked Postgres because I like that more, but it's just a preference.

Click **Create Server** and the installation process starts:



It will take some minutes, after which you'll have your server up and running.

A server perfectly configured to run Laravel, already set up with the Nginx server, database, and much more.

Once it's done, here is the control panel of your server. On the left, there's a menu that gives you access to specific menus.

The screenshot shows the Forge Laravel Forge web interface. The browser tab is 'floral-surf | Laravel Forge'. The URL is 'forge.laravel.com/servers/681352/sites'. The top navigation bar includes 'FORGE', a search bar, and tabs for 'Servers', 'Sites', 'Circles', 'Recipes', and 'Docs'. The user 'Flavio' is logged in. The main content area is for a server named 'floral-surf' (Server ID 681352, Server Type App, Connected). It shows public and private IP addresses and the region 'Frankfurt'. A sidebar on the left lists various server management options: Sites, Database, SSH Keys, Monitoring, Backups, PHP, Packages, Nginx Templates, Scheduler, Daemons, Network, Logs, Events, Integrations, and Meta. The 'New Site' form is displayed, with a note explaining that a site represents a 'domain' on the server. The form includes fields for 'Root Domain' (example.com), 'Project Type' (General PHP / Laravel), and 'Aliases' (alias.example.com, another-example.com). There are 'Restart', 'Stop', and 'Add Site' buttons. Below the form is a table of existing sites:

SITE	PHP	DEPLOYED
default	8.2	Never Deployed

A 'Destroy Server' link is visible at the bottom right of the main content area.

For example you have access to server logs through “Logs”:

The screenshot shows the Laravel Forge web interface. At the top, there's a navigation bar with the 'FORGE' logo, a search bar, and links to 'Servers', 'Sites', 'Circles', 'Recipes', and 'Docs'. The user's name 'Flavio' is in the top right. Below the navigation bar, the main content area shows the details for a server named 'floral-surf'. It includes the server's public and private IP addresses, its region (Frankfurt), and buttons for 'Restart' and 'Stop'. A left sidebar contains a list of server management tools: Sites, Database, SSH Keys, Monitoring, Backups, PHP, Packages, Nginx Templates, Scheduler, Daemons, Network, Logs (highlighted), Events, Integrations, and Meta. The 'Server Logs' panel is open, displaying a list of log entries from the file `/var/log/php8.2-fpm.log`. The logs show the FPM process starting, terminating, and starting again multiple times. At the bottom right of the logs panel, there's a button to 'Destroy Server'.

Back to Servers Credential Personal Server ID 681352 Server Type App Connected

floral-surf Restart Stop

Public IP 209.38.237.58 Private IP 10.114.0.3 Region Frankfurt

Sites Database SSH Keys Monitoring Backups PHP Packages Nginx Templates Scheduler Daemons Network Logs Events Integrations Meta

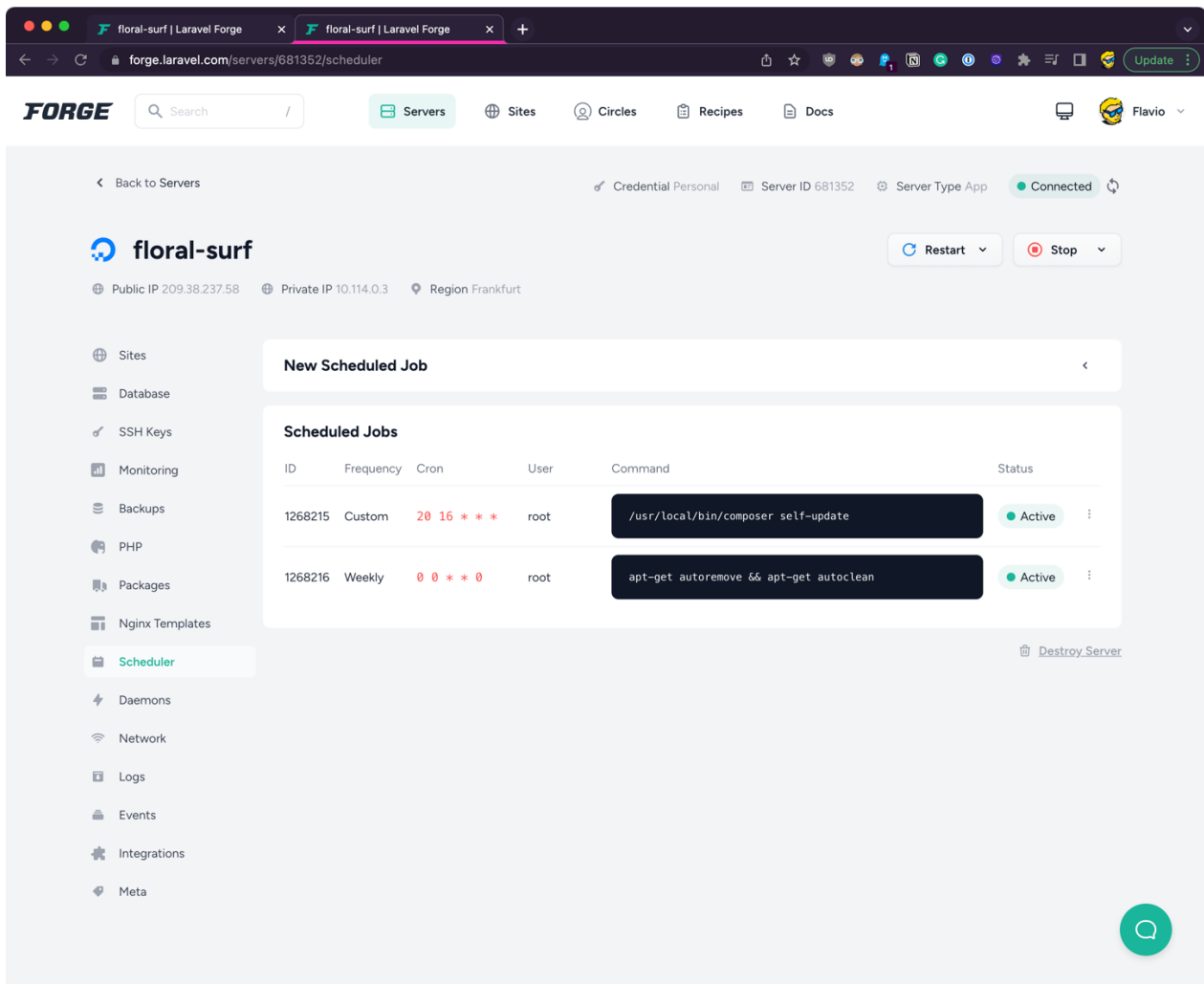
Server Logs View Log (/var/log/php8.2-fpm.log)

```
1 [13-May-2023 08:41:40] NOTICE: fpm is running, pid 30569
2 [13-May-2023 08:41:40] NOTICE: ready to handle connections
3 [13-May-2023 08:41:40] NOTICE: systemd monitor interval set to 10000ms
4 [13-May-2023 08:41:42] NOTICE: Terminating ...
5 [13-May-2023 08:41:42] NOTICE: exiting, bye-bye!
6 [13-May-2023 08:41:42] NOTICE: fpm is running, pid 31070
7 [13-May-2023 08:41:42] NOTICE: ready to handle connections
8 [13-May-2023 08:41:42] NOTICE: systemd monitor interval set to 10000ms
9 [13-May-2023 08:41:53] NOTICE: Terminating ...
10 [13-May-2023 08:41:53] NOTICE: exiting, bye-bye!
11 [13-May-2023 08:41:53] NOTICE: fpm is running, pid 31364
12 [13-May-2023 08:41:53] NOTICE: ready to handle connections
13 [13-May-2023 08:41:53] NOTICE: systemd monitor interval set to 10000ms
14 [13-May-2023 08:43:44] NOTICE: Terminating ...
15 [13-May-2023 08:43:44] NOTICE: exiting, bye-bye!
16 [13-May-2023 08:43:45] NOTICE: fpm is running, pid 31993
17 [13-May-2023 08:43:45] NOTICE: ready to handle connections
18 [13-May-2023 08:43:45] NOTICE: systemd monitor interval set to 10000ms
19 [13-May-2023 08:47:26] NOTICE: Terminating ...
20 [13-May-2023 08:47:26] NOTICE: exiting, bye-bye!
21 [13-May-2023 08:47:29] NOTICE: fpm is running, pid 41414
22 [13-May-2023 08:47:29] NOTICE: ready to handle connections
23 [13-May-2023 08:47:29] NOTICE: systemd monitor interval set to 10000ms
24
```

Showing the last 500 lines from /var/log/php8.2-fpm.log

Destroy Server

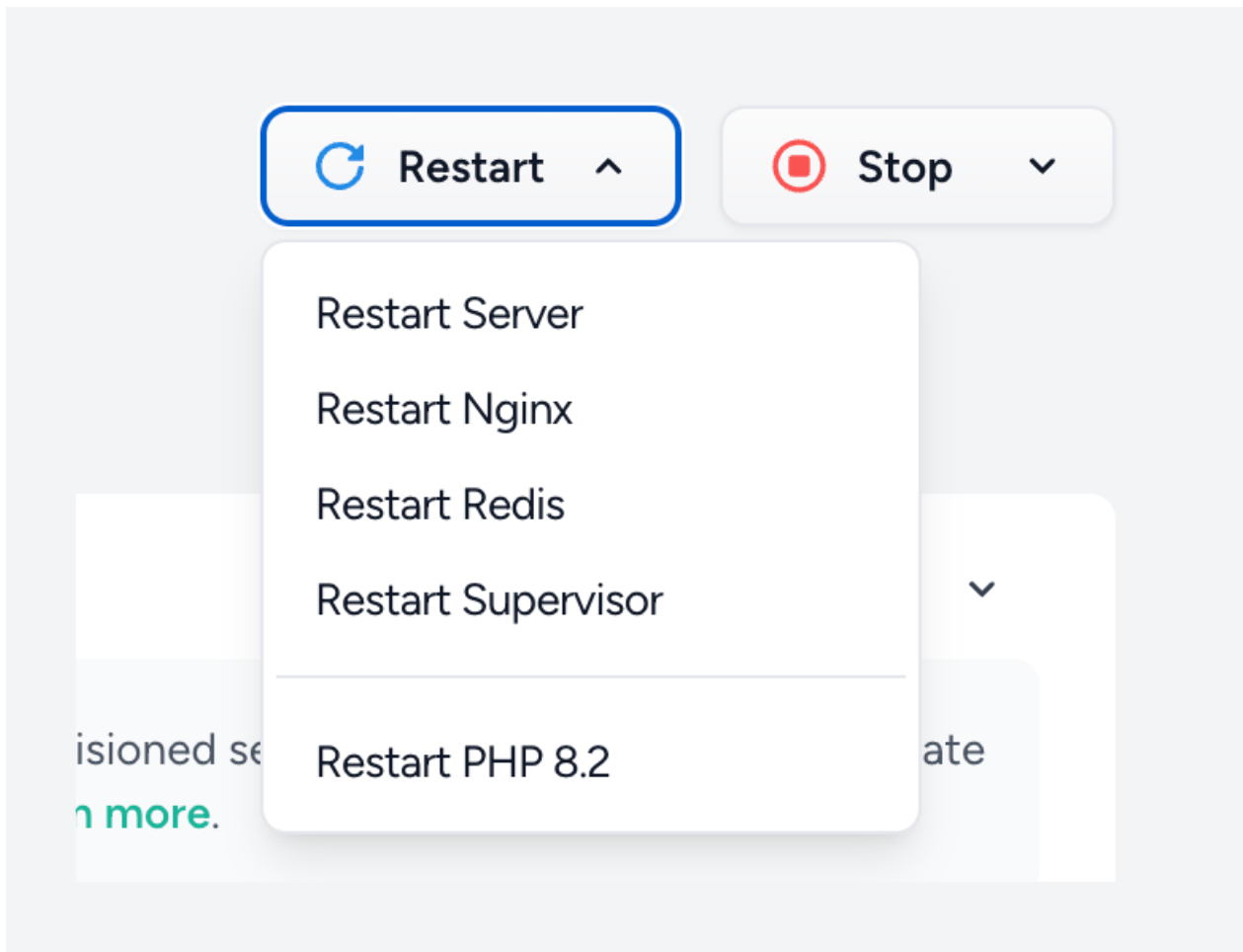
You can see the scheduled jobs in “Scheduler”:



...and lots more.

Back to the Sites menu.

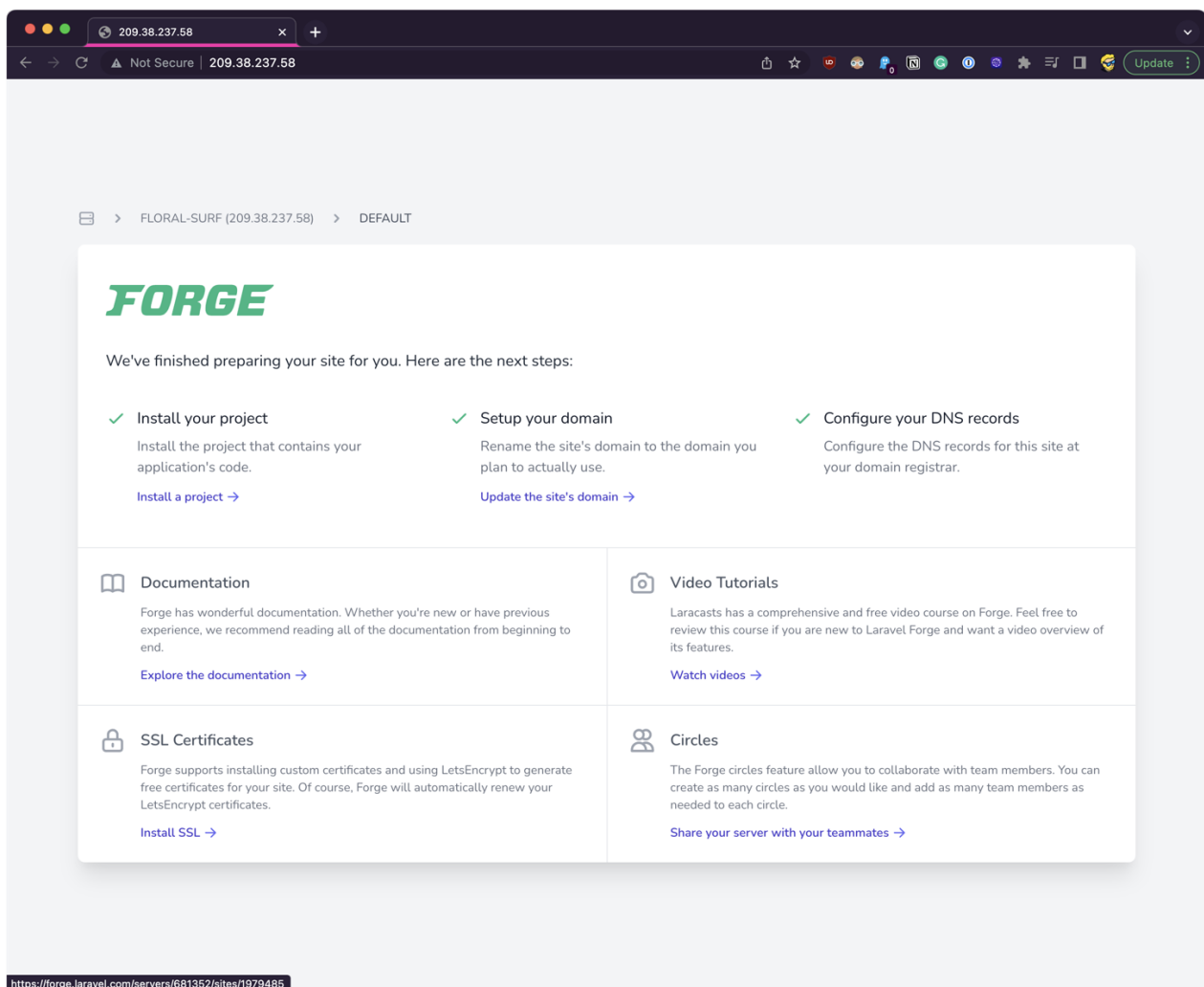
You can restart the entire server, or specific services, through the **Restart** drop down menu:



which is very handy, and you can deploy new sites on this server.

Each server can host multiple different sites.

There is a default one already set up, and if you copy and paste the public IP address of the server in your browser, you'll see it working:

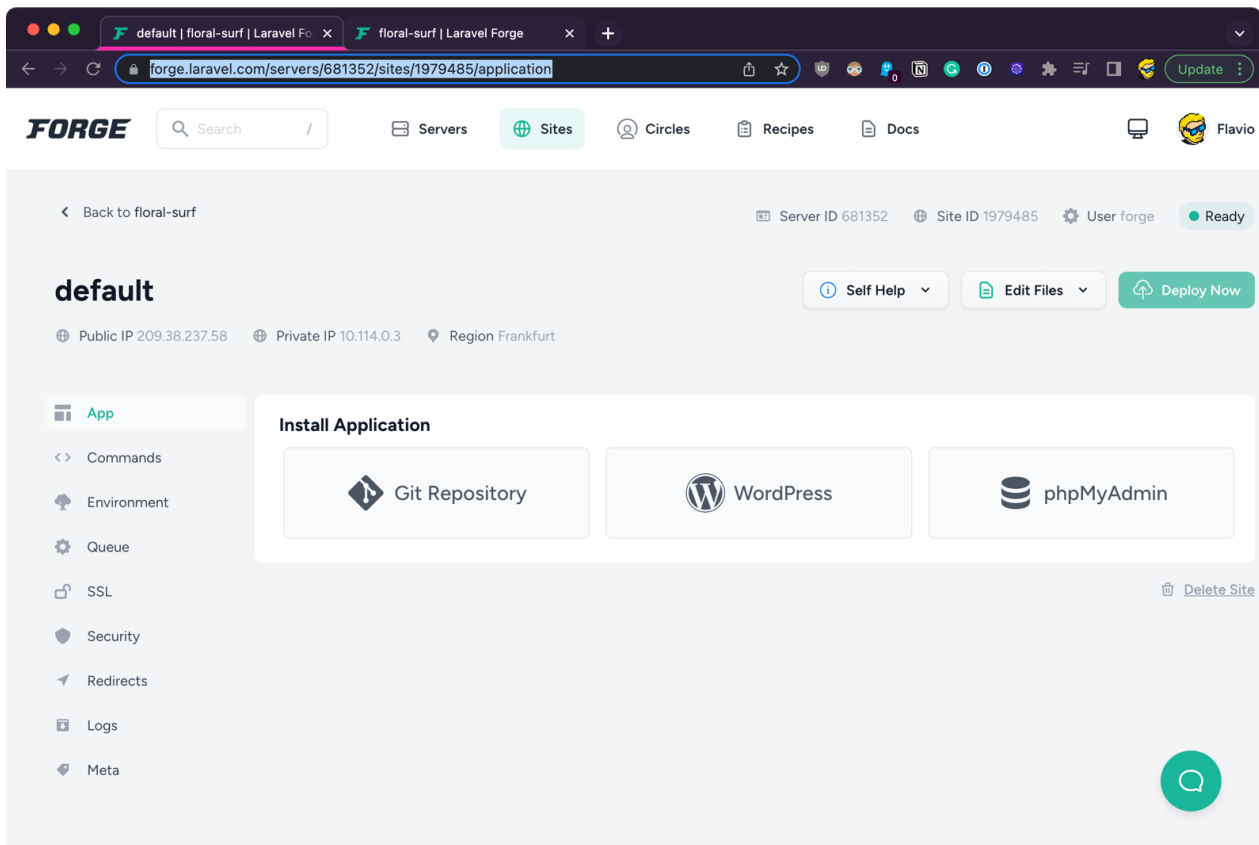


Now let's deploy the application on this site.

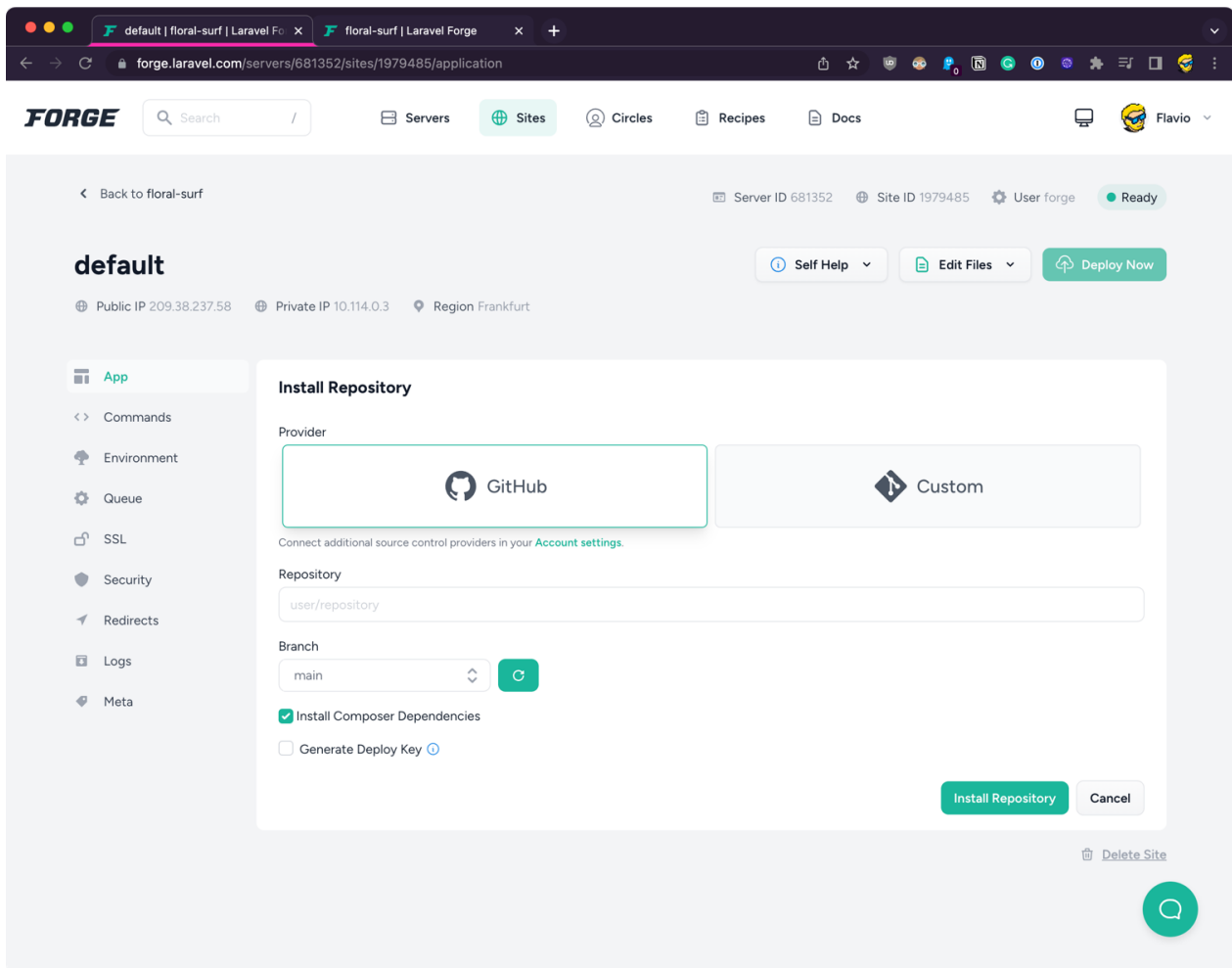
Ideally what you want to do is, you create a new site with a domain / subdomain assigned.

But it's starting to become complicated for this handbook, so we'll just use the default site which works on the IP address instead.

Click the default site in the dashboard and you'll see the site panel:



We have the site in GitHub, so click “Git Repository”:



Now type the name of the repository you want to deploy prefixed with the GitHub account name, in my case `flaviocopes/second`, select the branch (usually `main`) and click **Install Repository**. After a while it's done!

The screenshot shows the Forge Laravel web interface. At the top, there's a navigation bar with the Forge logo, a search bar, and links for Servers, Sites, Circles, Recipes, and Docs. The user 'Flavio' is logged in. The main content area is for a site named 'default' (Server ID 681352, Site ID 1979485, Region Frankfurt). It includes buttons for 'Self Help', 'Edit Files', and 'Deploy Now'. A sidebar on the left lists various management options like App, Deployments, Commands, Packages, Notifications, Environment, Queue, SSL, Security, Redirects, Logs, and Meta. The 'Deployment' section explains the quick deploy process and has an 'Enable Quick Deploy' button. The 'Deploy Script' section shows a bash script for pulling code, installing Composer, and restarting FPM. Below the script is a checkbox for 'Make .env variables available to deploy script' and an 'Update' button. The 'Deployment Trigger URL' section provides a URL to trigger deployments and a 'Refresh Site Token' button.

default

Public IP 209.38.237.58 Private IP 10.114.0.3 Region Frankfurt

Server ID 681352 Site ID 1979485 User forge Active

Deployment

Quick deploy allows you to easily deploy your projects when you push to source control. When you push to this application's deployment branch, Forge will pull your latest code from source control and execute your deployment script.

[Enable Quick Deploy](#) [View Latest Deployment Log](#)

Deploy Script

```

1 cd /home/forge/default
2 git pull origin $FORGE_SITE_BRANCH
3
4 $FORGE_COMPOSER install --no-dev --no-interaction --prefer-dist --optimize-autoloader
5
6 { flock -w 10 9 || exit 1
7   echo 'Restarting FPM...'; sudo -S service $FORGE_PHP_FPM reload } 9>/tmp/fpmlock
8
9 if [ -f artisan ]; then
10   $FORGE_PHP artisan migrate --force
11 fi
12

```

☐ Make `.env` variables available to deploy script

[Update](#)

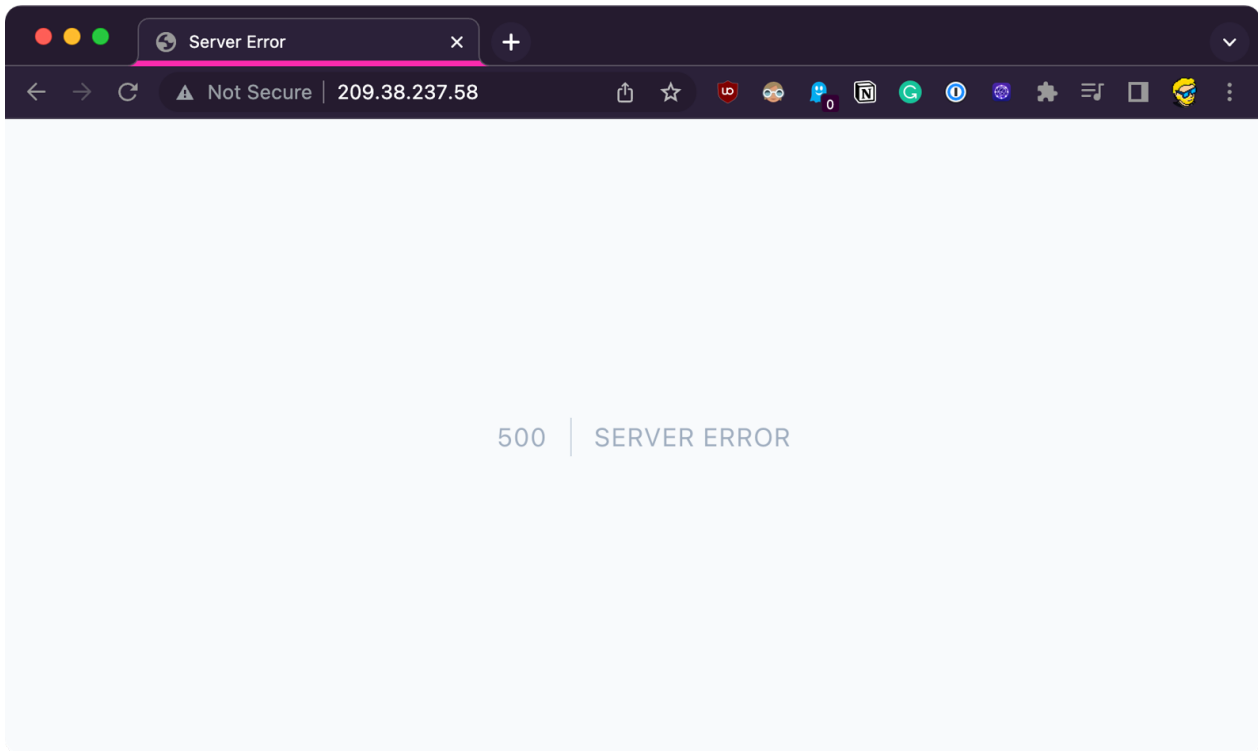
Deployment Trigger URL

Using a custom Git service, or want a service like [Chipper CI](#) to run your tests before your application is deployed to Forge? It's simple. When you commit fresh code, or when your continuous integration service finishes testing your application, instruct the service to make a **GET** or **POST** request to the following URL. Making a request to this URL will trigger your Forge deployment script:

<https://forge.laravel.com/servers/681352/sites/1979485/deploy/http?token=YqaFBbAqZyoAzpVWw0ImHJqRMHL9Hub7m36jTRw>

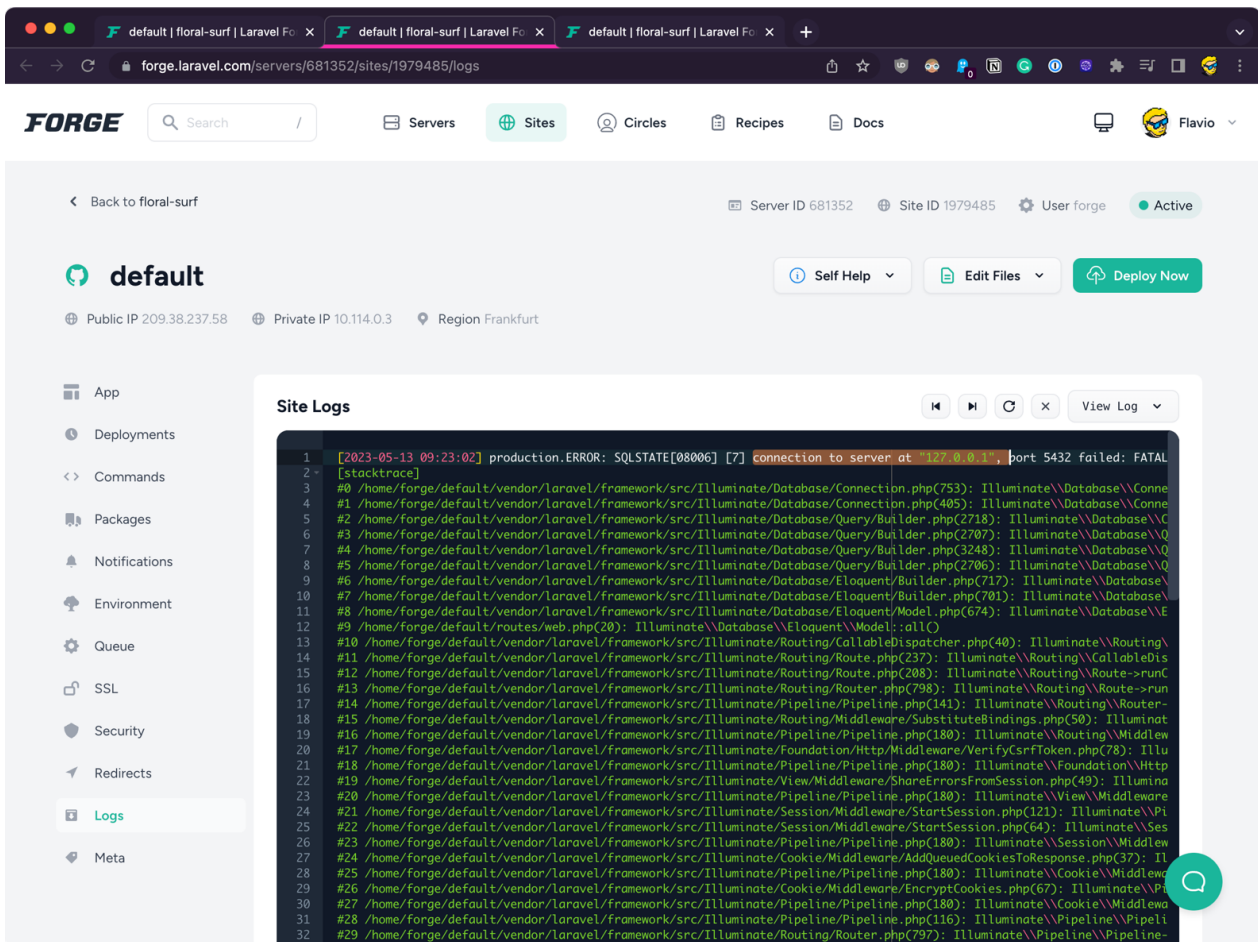
[Refresh Site Token](#)

But if you go to the IP address again, there's an error:



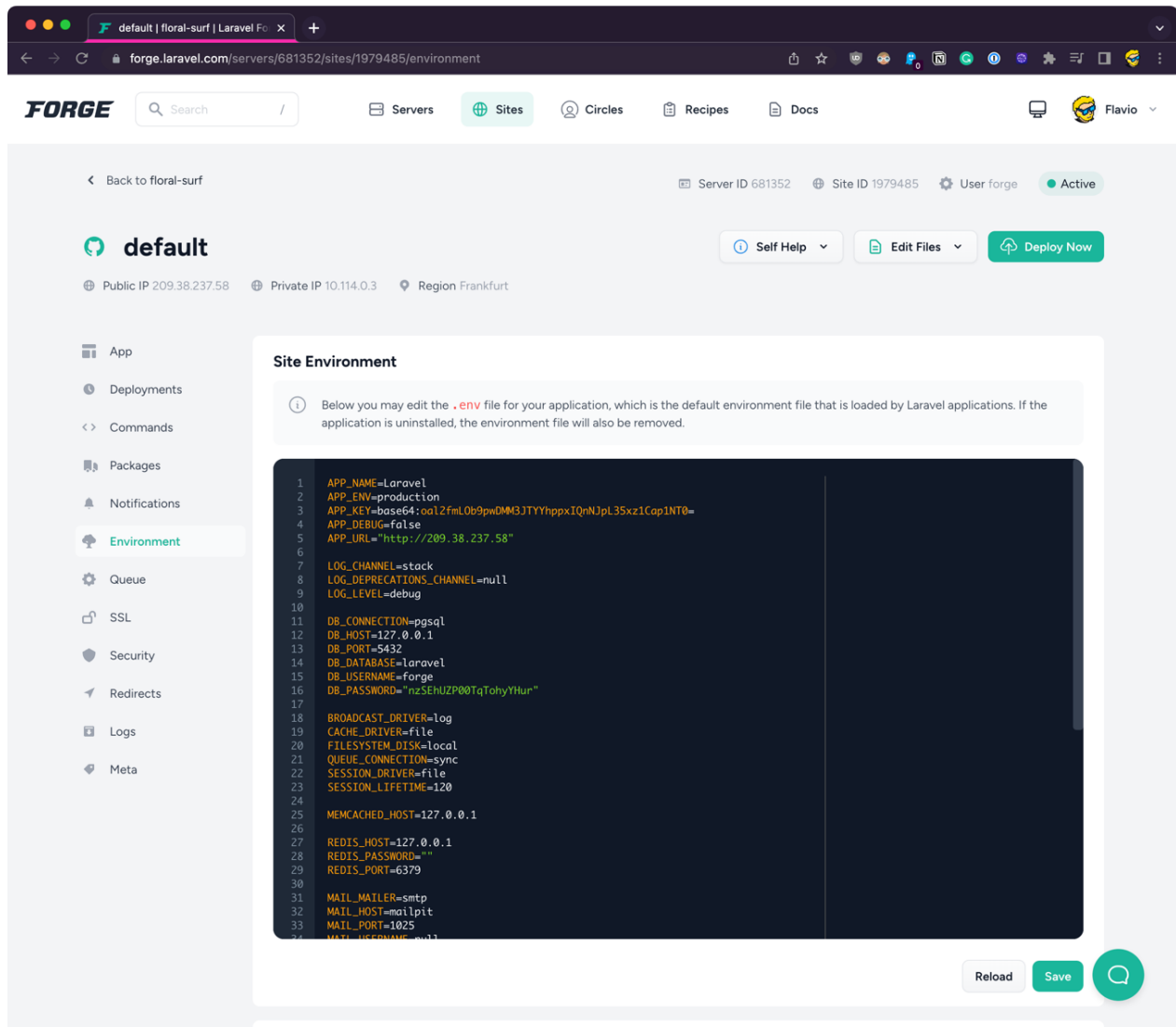
Mmmm! We don't see more details because now the site is in a production environment, and we don't show detailed logs to users.

To figure out the problem let's go back to the panel, open Logs and you'll see the error is related to connecting to the database.



If you look closely in the GitHub repository you will see the `.env` file was not pushed to GitHub, and this is correct because you don't want to store the environment variables in Git.

In the Forge site config click the **Environment** tab, this is where you will edit your environment variables:

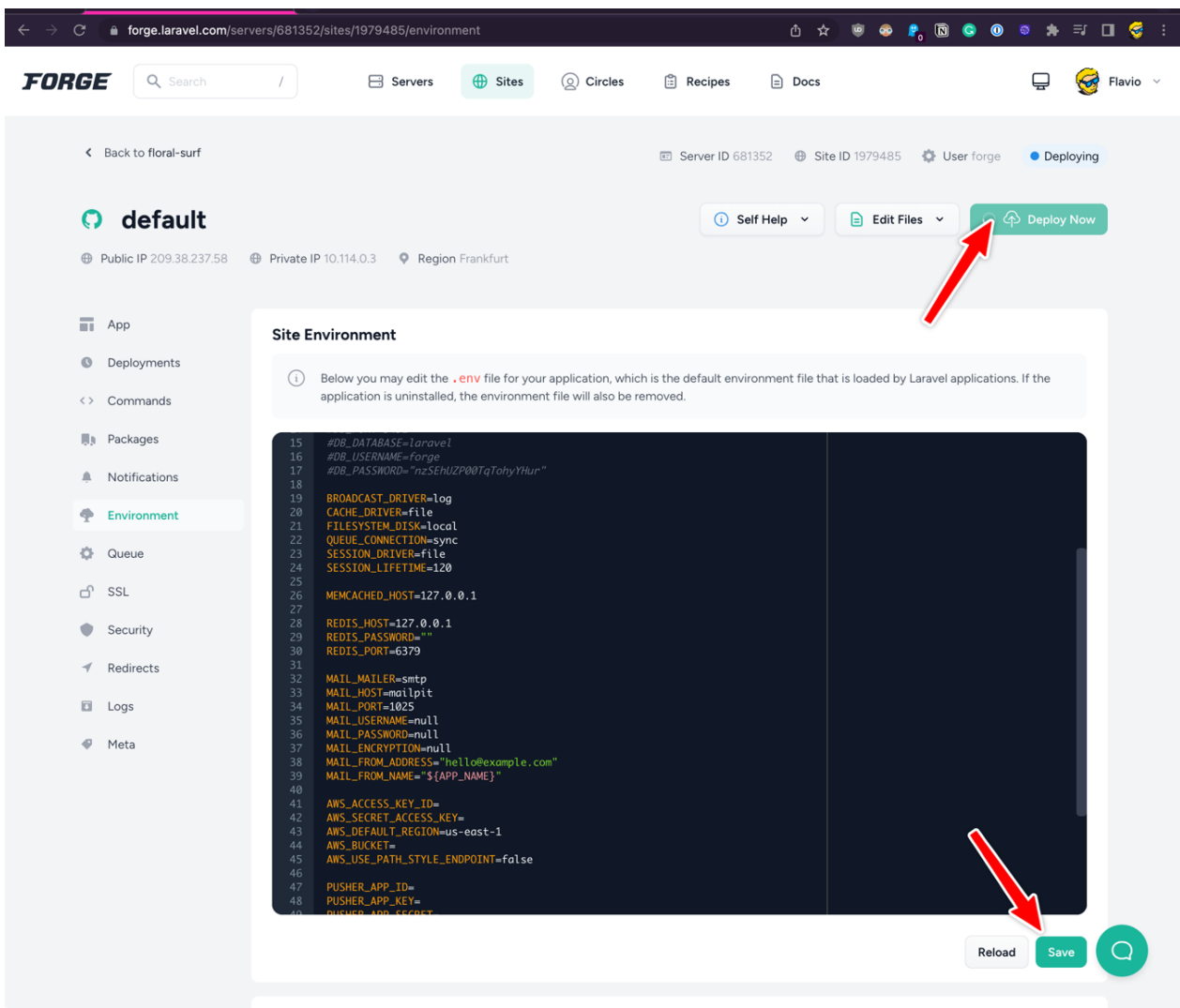


Comment the `DB_*` fields and add

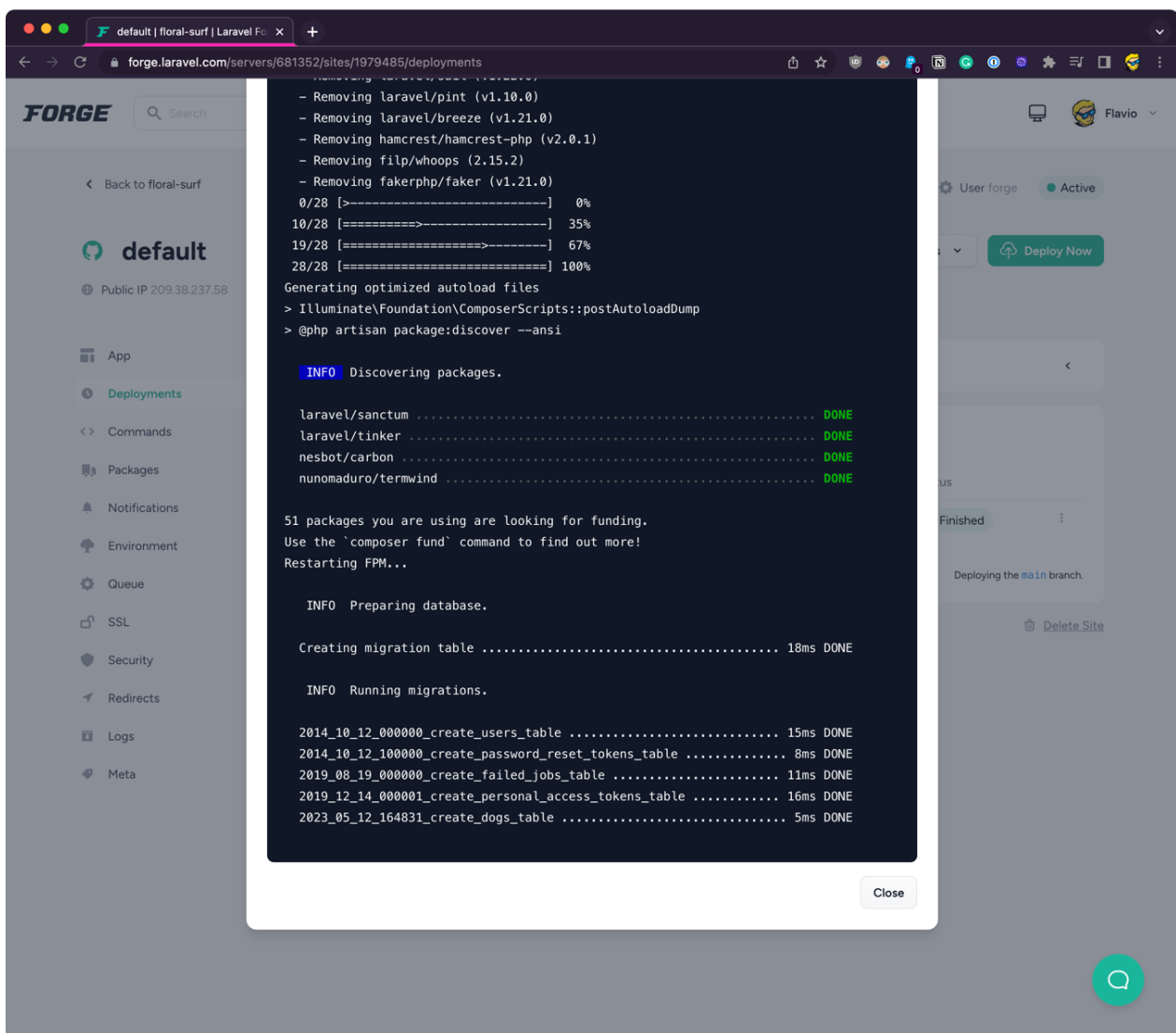
```
DB_CONNECTION=sqlite
```

```
DB_CONNECTION=sqlite
#DB_CONNECTION=pgsql
#DB_HOST=127.0.0.1
#DB_PORT=5432
#DB_DATABASE=laravel
#DB_USERNAME=forge
#DB_PASSWORD="nzSEhUZP00TqTohyYHur"
```

Click **Save** and then **Deploy Now**.



If you go to the deployments menu you can inspect the deployment output, which is handy:



If the build fails, you'll get an alert and also an email, pretty nice.

The build seems to work fine but if you reload the browser we have another error now. Something about Vite.

Remember we ran `npm run dev` to start Vite in development?

Now we have to run `npm run build`, after running `npm install`.

We need to update our Deploy Script from the **App** tab, adding this at the end:

```
npm install
npm run build
```

default | floral-surf | Laravel Forge

forge.laravel.com/servers/681352/sites/1979485/application

FORGE Search / Servers Sites Circles Recipes Docs Flavio

Back to floral-surf Server ID 681352 Site ID 1979485 User forge Active

default Self Help Edit Files Deploy Now

Public IP 209.38.237.58 Private IP 10.114.0.3 Region Frankfurt

App Deployments Commands Packages Notifications Environment Queue SSL Security Redirects Logs Meta

Deployment

Quick deploy allows you to easily deploy your projects when you push to source control. When you push to this application's deployment branch, Forge will pull your latest code from source control and execute your deployment script.

Enable Quick Deploy View Latest Deployment Log

Deploy Script

```

1 cd /home/forge/default
2 git pull origin $FORGE_SITE_BRANCH
3
4 $FORGE_COMPOSER install --dev --no-interaction --prefer-dist --optimize-autoloader
5
6 ( flock -w 10 9 & echo 'Restarting PHP-FPM'; sudo -S service $FORGE_PHP_FPM reload ) 9>/tmp/fpmlock
7
8 if [ -f artisan ]; then
9     $FORGE_PHPartisan --clear-compiled
10 fi
11
12 npm install
13 npm run build
14
15

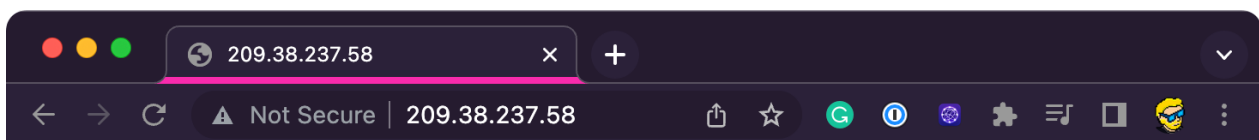
```

☐ Make .env variables available to deploy script

Update

Then click **Update** and **Deploy Now**.

Now it works!

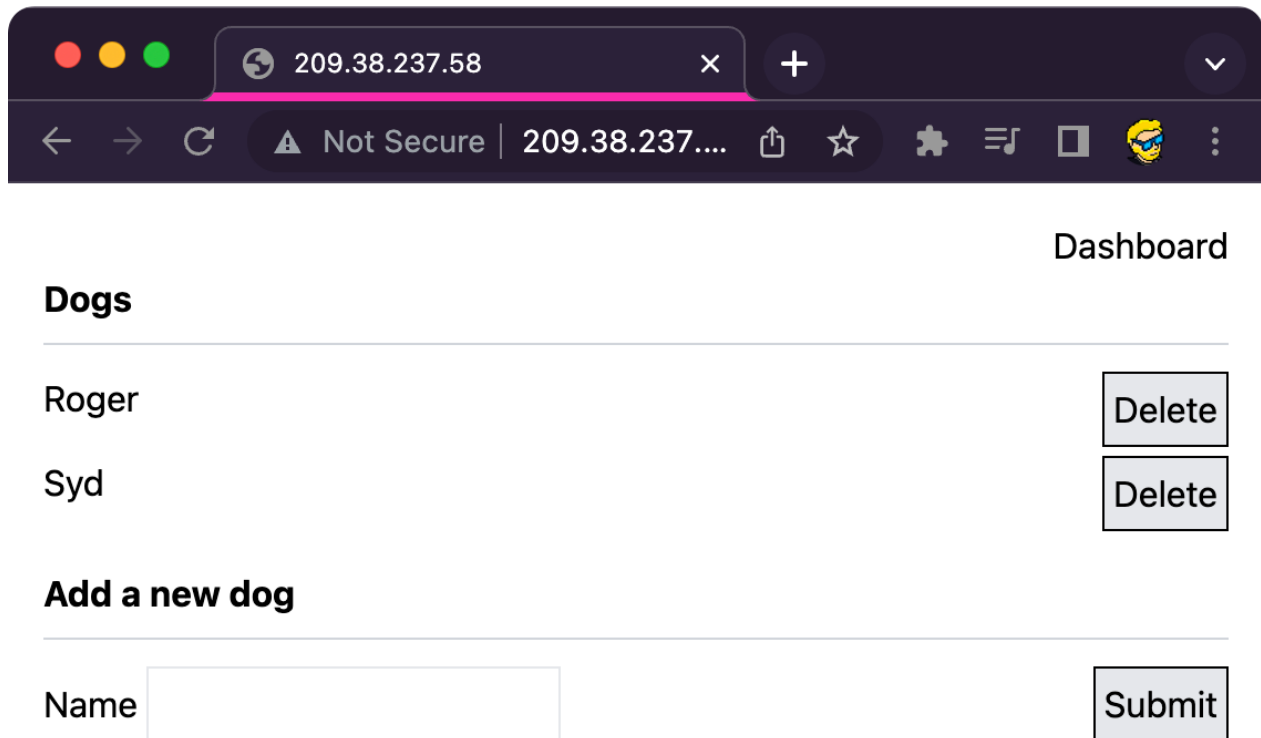


[Log in](#) [Register](#)

Dogs

No dogs yet

Also try registering in, it will work as expected and we'll be able to add and edit data:



Dashboard

Dogs

Roger	Delete
Syd	Delete

Add a new dog

Name		Submit
------	----------------------	--------

Nice! We're done with deployments and Forge.

We could spend more time on this topic, but there's so much more to explore.

We've seen how to create a Web Application, as simple as it could be, just a form that stores a field into a database, but complete with ready-made authentication provided by the Breeze starter kit.

We've seen basic routing, and models, views and controllers interact to store and retrieve data through the Eloquent ORM.

Let's now move to other aspects of Laravel.

15. Dynamic routes

We've seen how to create a route in the `routes/web.php` file:

```
Route::get('/dogs', function () {  
    return view('dogs');  
})->name('dogs');
```

This is a **static route**, that responds on the `/dogs` URL.

Now suppose you want to create a page for each single dog, maybe you'll fill that with a description, an image, whatever.

You can't create a static route for each dog in the database, because you don't know the name of

the dog.

Imagine you have 2 dogs Max and Daisy, this would display a “dog” view (which we don’t have yet) on the `/dogs/max` and `/dogs/daisy`:

```
Route::get('/dogs/max', function () {
    return view('dog');
})
```

```
Route::get('/dogs/daisy', function () {
    return view('dog');
})
```

What we do instead is, we have a **dynamic segment** in the URL:

```
Route::get('/dogs/{slug}', function () {
    return view('dog');
})
```

slug is a term that identifies a URL portion in lowercase and without spaces, for example if the name of the dog is Max, the slug is `max`.

Now we can pass the `slug` value to the callback function (the function that’s called when the route is hit), and inside the function we can pass it to the view:

```
Route::get('/dogs/{slug}', function ($slug) {
    return view('dog', ['slug' => $slug]);
})
```

Now the `$slug` variable is available inside the Blade template.

But we want to retrieve the actual dog data. We have the slug, which we can imagine it’s stored in the database when we add the dog.

To do that, we use the Dog model in the route, like this:

```
use App\Models\Dog;

Route::get('/dogs/{slug}', function ($slug) {
    $dog = Dog::find($slug)
    return view('dog', ['dog' => $dog]);
})
```

16. Non-web routes

In the `routes` folder you have `web.php`, but not just that file. We have `api.php`, `channels.php` and `console.php`.

- `web.php` handles HTTP requests from web browsers
- `api.php` handles API endpoints. We use it to create an API that can be used for example by a mobile application, or directly by the users (if that’s something you want them to)
- `console.php` contains routes used by the command line interface, **Artisan**. We can write

command line applications for our app, in PHP, and execute them, it's pretty handy

17. Creating commands

We've used Artisan, the Laravel command line tool, to perform various actions:

- `php artisan serve`
- `php artisan make:migration`
- `php artisan migrate`
- `php artisan make:model`
- `php artisan make:controller`
- `php artisan breeze:install`

Those are all built-in commands.

There are many, many more.

Some you'll use often, some you'll never use.

Run `php artisan` to see them all with a short explanation:

```
→ second git:(main) php artisan
Laravel Framework 10.10.1

Usage:
  command [options] [arguments]

Options:
  -h, --help                Display help for the given command. When no command is given display help
for the list command
  -q, --quiet                Do not output any message
  -V, --version              Display this application version
                          --ansi|--no-ansi Force (or disable --no-ansi) ANSI output
  -n, --no-interaction       Do not ask any interactive question
                          --env[=ENV] The environment the command should run under
  -v|vv|vvv, --verbose       Increase the verbosity of messages: 1 for normal output, 2 for more verbos
e output and 3 for debug

Available commands:
  about                    Display basic information about your application
  clear-compiled            Remove the compiled class file
  completion               Dump the shell completion script
  db                       Start a new database CLI session
  docs                     Access the Laravel documentation
  down                     Put the application into maintenance / demo mode
  env                       Display the current framework environment
  help                     Display help for a command
  inspire                  Display an inspiring quote
  list                     List commands
  migrate                  Run the database migrations
  optimize                  Cache the framework bootstrap files
  serve                    Serve the application on the PHP development server
  test                     Run the application tests
  tink                     Interact with your application
  up                       Bring the application out of maintenance mode
  auth
  auth:clear-resets        Flush expired password reset tokens
  breeze
  breeze:install            Install the Breeze controllers and resources
  cache
  cache:clear              Flush the application cache
  cache:forget             Remove an item from the cache
  cache:prune-stale-tags   Prune stale cache tags from the cache (Redis only)
  cache:table              Create a migration for the cache database table
  channel
  channel:list             List all registered private broadcast channels
  config
  config:cache              Create a cache file for faster configuration loading
  config:clear             Remove the configuration cache file
  db
  db:monitor               Monitor the number of connections on the specified database
  db:seed                  Seed the database with records
  db:show                  Display information about the given database
  db:table                 Display information about the given database table
  db:wipe                  Drop all tables, views, and types
  env
  env:decrypt              Decrypt an environment file
  env:encrypt              Encrypt an environment file
  event
```

And to see how to use a command in particular, run `php artisan <command> -h`:

```
~d/second npm run dev ~d/second
→ second git:(main) php artisan optimize -h
Description:
  Cache the framework bootstrap files

Usage:
  optimize

Options:
  -h, --help            Display help for the given command. When no command
                        is given display help for the list command
  -q, --quiet           Do not output any message
  -V, --version         Display this application version
      --ansi|--no-ansi Force (or disable --no-ansi) ANSI output
  -n, --no-interaction Do not ask any interactive question
      --env[=ENV]      The environment the command should run under
  -v|vv|vvv, --verbose Increase the verbosity of messages: 1 for normal ou
                        tput, 2 for more verbose output and 3 for debug
→ second git:(main) █
```

You can create your own commands, too.

Run

```
php artisan make:command MyCommand
```

This creates a file in `app/Console/Commands/MyCommand.php` pre-filled with some code:

```
<?php

namespace App\Console\Commands;

use Illuminate\Console\Command;

class MyCommand extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'app:my-command';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Command description';
}
```

```

/**
 * Execute the console command.
 */
public function handle()
{
    //
}
}

```

`$signature` defines how the command will be called, in this case you can run it using

```
php artisan app:my-command
```

In the `handle()` method you'll write the code that runs when the command is executed.

```

public function handle()
{
    //
}

```

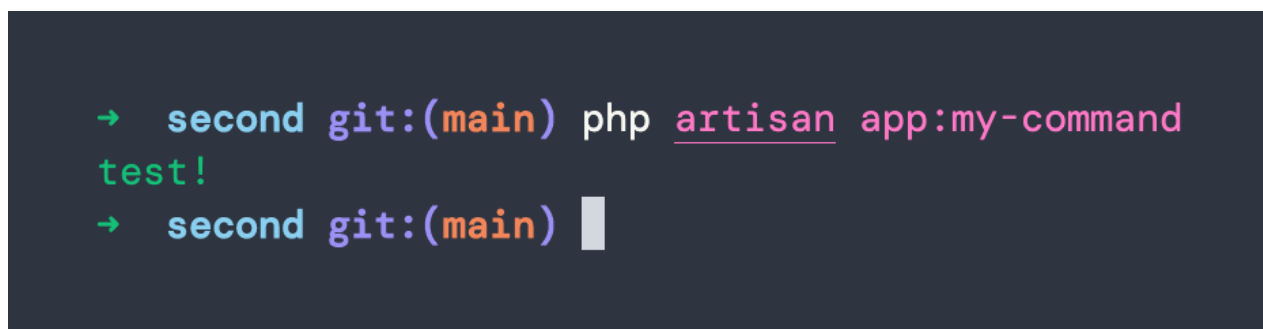
The simplest code could be printing something to the console, for example:

```

public function handle()
{
    $this->info('test!');
}

```

Now try running `php artisan app:my-command`:



A terminal window with a dark background. The prompt is `second git:(main)`. The user enters `php artisan app:my-command` and presses enter. The output is `test!`. The user then enters another `php artisan app:my-command` and presses enter, but the output is not yet visible.

You can do lots of things in a command. You can accept arguments, interactively ask something to the user using prompts to confirm or asking for input, or let them choose between different options, you can format output in different ways...

Commands are great to perform one-off tasks, maintenance, and much more. Inside a command you have access to all the goodies provided by Laravel, including your own code, classes, and more.

You can also call other commands. And commands can be ran by any part of your Laravel app.

You can also schedule commands using schedules. The server can be configured to run Laravel's schedules, and then any schedule configured in Laravel will be executed as needed.

18. Where to go from here

We've reached the end of the handbook!

This is intended to be a hands-on, quick introduction to Laravel.

Definitely not a complete guide. But I think that now you can go start building an app using Laravel.

What are you waiting for?

Conclusion

Thanks a lot for reading this book.

For more, head over to flaviocopes.com.

Send any feedback, errata or opinions at flavio@flaviocopes.com