

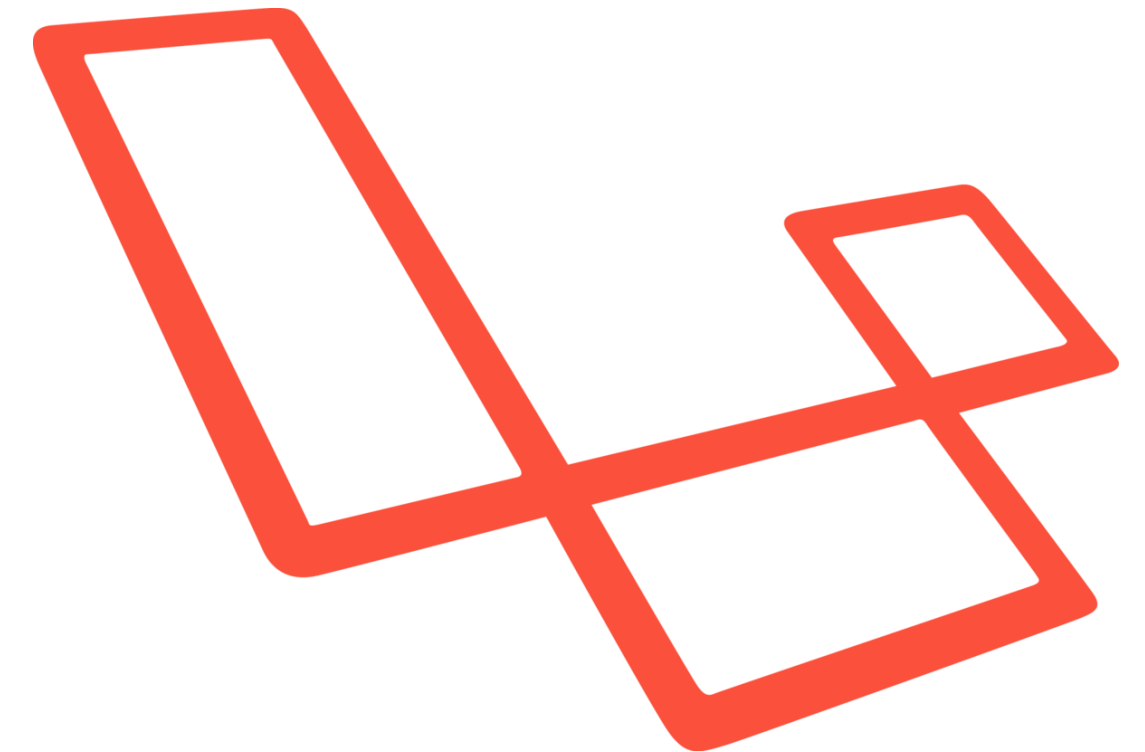


Laravel: Logging, File uploads, Mailables and Authentication

Web, Mobile and Security
Frédéric Vlummens

Agenda

- Logging
- File uploads
- Mailables
- Authentication



howest
hogeschool

Logging

Logging

- We already know logging in JavaScript:
 - `console.log("...")`
 - `console.error("...")`
 - ...
- For debugging/development purposes, it can be useful to perform logging in our Laravel applications as well.

Logging in Laravel

- Uses the Log class
- Can be found in the namespace Illuminate\Support\Facades
- Make sure to include it

```
use Illuminate\Support\Facades\Log;
```

Logging in Laravel

```
namespace App\Http\Controllers;

use Illuminate\Support\Facades\Log;

class LoggingController extends Controller
{
    function index() {
        Log::info("Retrieving vegetables");

        $vegetables = ["cucumber", "lettuce", "cauliflower"];


        return view("vegetables", [ "vegetables" => $vegetables ]);
    }
}
```

Logging in Laravel

```
storage > logs > ≡ laravel.log
1  [2020-04-16 08:34:18] local.INFO: Retrieving vegetables
2
```

Different log levels and methods

- Different log levels:

- emergency
 - alert
 - critical
 - error
 - warning
 - notice
 - info
 - debug
- 

- Corresponding methods:

- `Log::emergency($msg)`
- `Log::alert($msg)`
- `Log::critical($msg)`
- `Log::error($msg)`
- `Log::warning($msg)`
- `Log::notice($msg)`
- `Log::info($msg)`
- `Log::debug($msg)`

Logging configuration

- Log levels allow fine grained control over what gets logged where
- Example use case:
 - Informational messages get logged in a text file (cfr. example)
 - Critical and higher messages get logged to a Slack channel to notify admins
- Can be configured in the file **./config/logging.php** of your project

Logging in Laravel

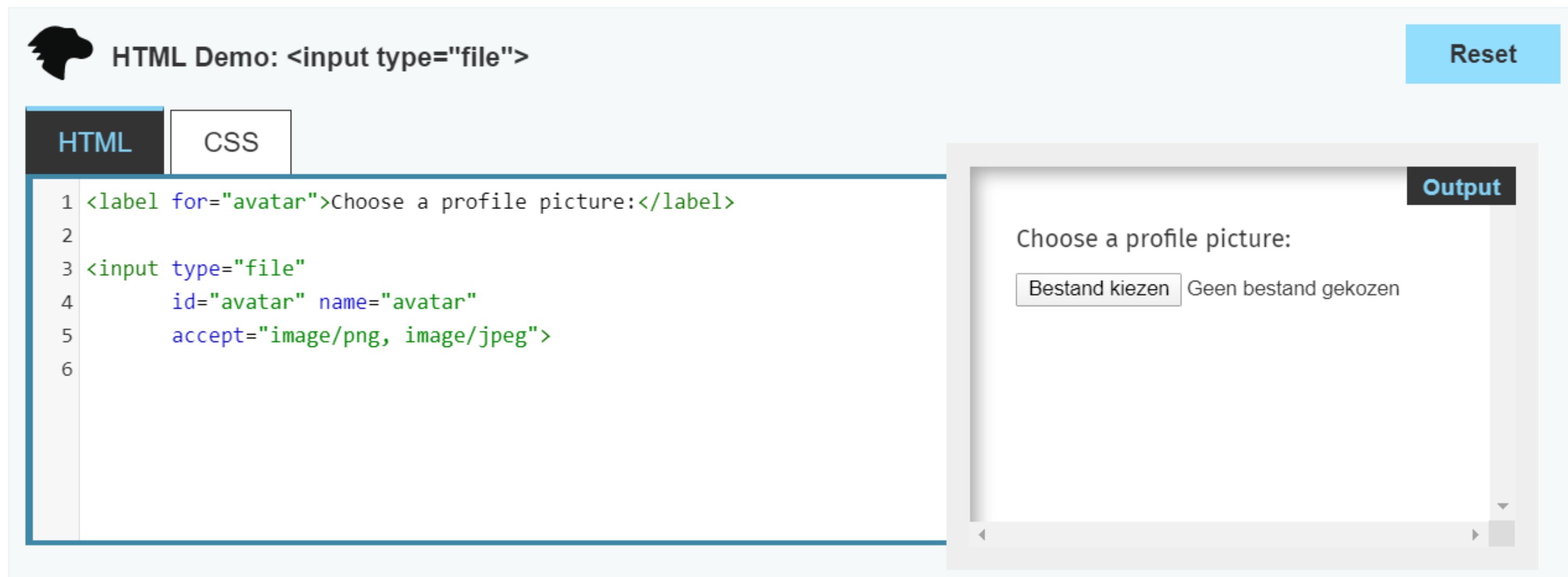
- More information: <https://laravel.com/docs/master/logging#configuration>

howest
hogeschool

File uploads

What are file uploads?

- Allow you to upload files through an HTML form



The screenshot shows a web application interface for demonstrating HTML file uploads. At the top left, there is a logo of a stylized head and the text "HTML Demo: <input type='file'>". To the right of this is a blue "Reset" button. Below the title bar, there are two tabs: "HTML" (selected) and "CSS". The "HTML" tab displays a code editor with the following code:

```
1 <label for="avatar">Choose a profile picture:</label>
2
3 <input type="file"
4       id="avatar" name="avatar"
5       accept="image/png, image/jpeg">
6
```

To the right of the code editor is a preview window titled "Output". It shows the rendered HTML form, which includes the text "Choose a profile picture:" followed by a file input button labeled "Bestand kiezen" and the text "Geen bestand gekozen".

Source: <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/file>

File uploads: client-side attention points

- Your form must have **enctype** attribute equal to **multipart/form-data**

```
<form method="post" enctype="multipart/form-data">
  @csrf
  Avatar: <input type="file" name="avatar" />
  <input type="submit" value="Upload" />
</form>
```

- This encoding allows files to be uploaded through a POST.
- More information: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST>

File uploads: processing at the server-side

- Retrieve the file using **file** method
- Store in the correct upload folder using **store** method, which returns upload path, which e.g. can be added to DB table
- In this example, file will be stored in `./storage/app/avatars`

```
class FileUploadController extends Controller
{
    function index() {
        return view("index");
    }


    function upload(Request $request) {
        $path = $request → file("avatar") → store("avatars");
        Log::info("Stored avatar in " . $path);

        return view("index");
    }
}
```

File uploads: processing at the server-side

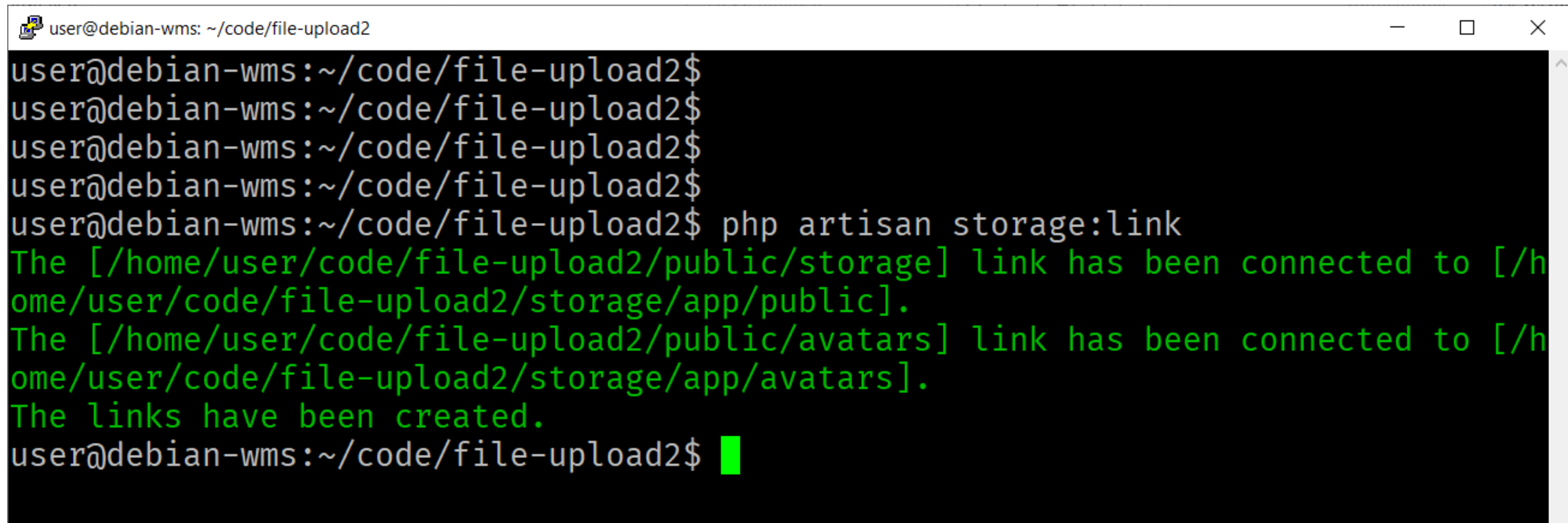
- Files stored in `./storage/app` are **not accessible** from the outside world
- If you want to include your avatar image in a blade file, you need to make sure it is available through **`./public`**
- Solution: create a link from `./storage/app/avatars` to `./public/avatars`
- First, edit **`./config/filesystems.php`** and add a new entry to the 'links' array:

```
80
81     'links' => [
82         public_path('storage') => storage_path('app/public'),
83         public_path('avatars') => storage_path('app/avatars'),
84     ],
85
86 ];
```



File uploads: processing at the server-side

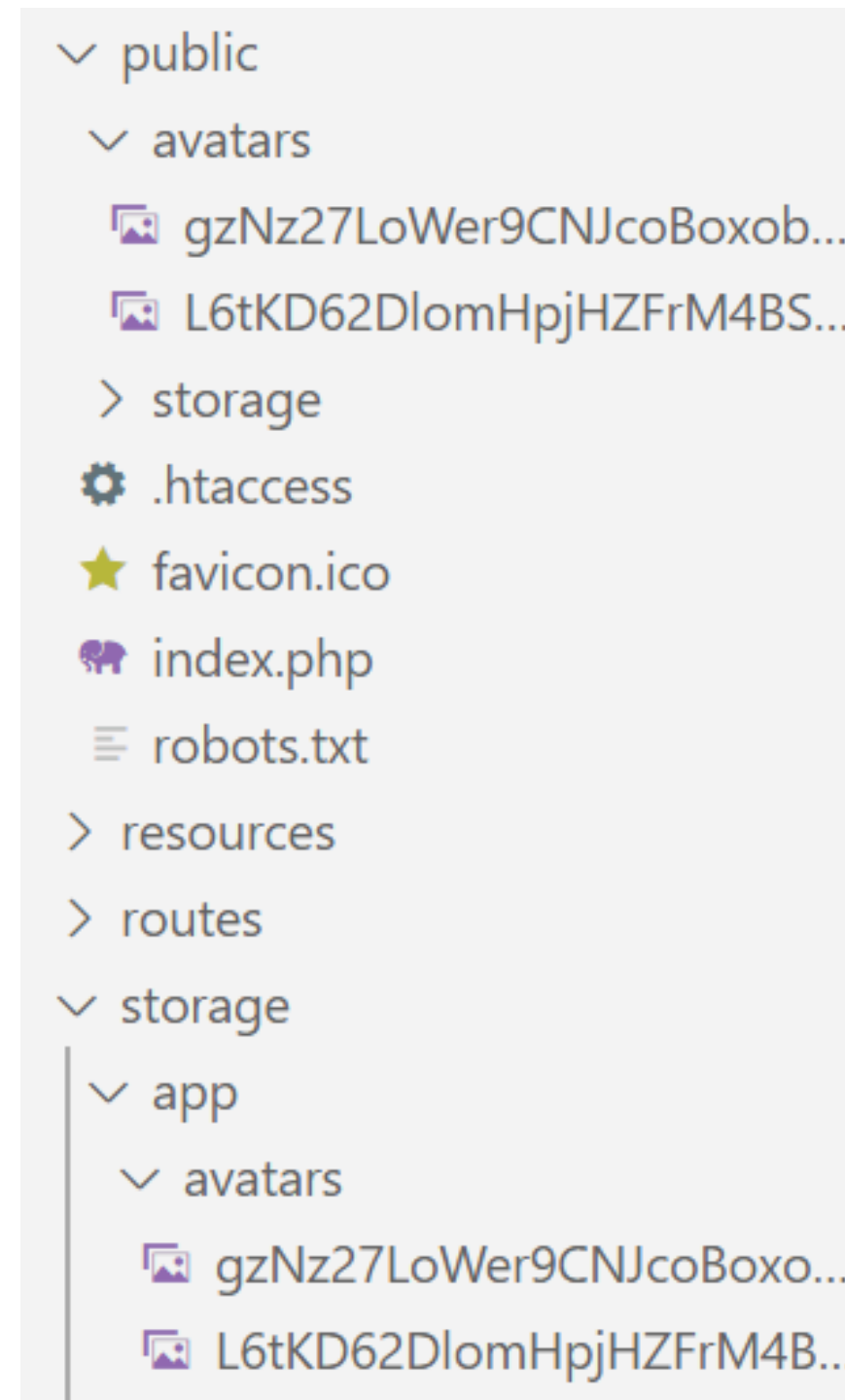
- Next, execute the command **php artisan storage:link**
- This will create the symbolic links for each entry in the 'links' array

A terminal window titled 'user@debian-wms: ~/code/file-upload2' with standard window controls. The terminal shows a series of five empty prompts, followed by the command 'php artisan storage:link'. The output is displayed in green text: 'The [/home/user/code/file-upload2/public/storage] link has been connected to [/home/user/code/file-upload2/storage/app/public].', 'The [/home/user/code/file-upload2/public/avatars] link has been connected to [/home/user/code/file-upload2/storage/app/avatars].', and 'The links have been created.' The prompt returns to 'user@debian-wms: ~/code/file-upload2\$' with a green cursor.

```
user@debian-wms: ~/code/file-upload2
user@debian-wms:~/code/file-upload2$
user@debian-wms:~/code/file-upload2$
user@debian-wms:~/code/file-upload2$
user@debian-wms:~/code/file-upload2$
user@debian-wms:~/code/file-upload2$ php artisan storage:link
The [/home/user/code/file-upload2/public/storage] link has been connected to [/h
ome/user/code/file-upload2/storage/app/public].
The [/home/user/code/file-upload2/public/avatars] link has been connected to [/h
ome/user/code/file-upload2/storage/app/avatars].
The links have been created.
user@debian-wms:~/code/file-upload2$
```


File uploads: processing at the server-side

- You can now reference the files using the **asset** function (cfr. previous classes)



Symlinked files, can be served to outside world

Actual files, not accessible from outside world

howest
hogeschool

Mailables

Mailables

- Allow you to send mail messages from Laravel
- E.g.: after successfully booking a hotel room, send out confirmation message

Mailables

- First, configure SMTP server settings in .env file

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=465
MAIL_USERNAME=[REDACTED]
MAIL_PASSWORD=[REDACTED]
MAIL_FROM_ADDRESS=noreply-ti@howest.be
MAIL_FROM_NAME=noreply
```

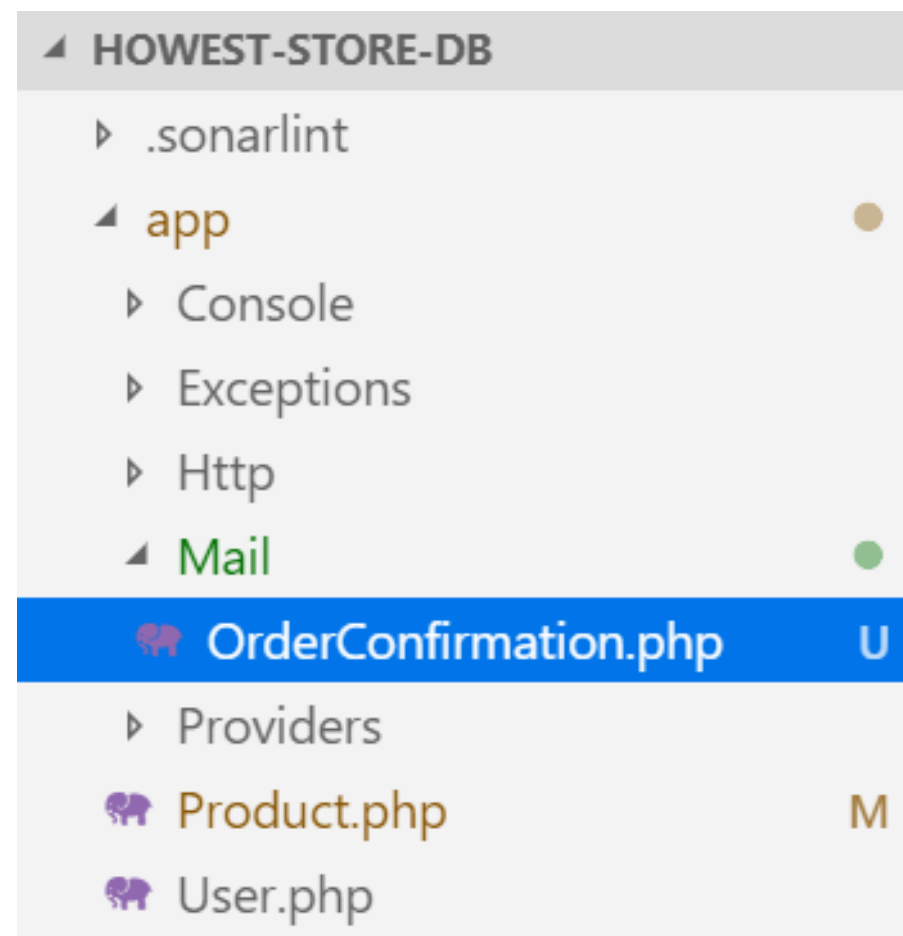
- We will be using Mailtrap, which simulates a real SMTP server
- Instead of delivering the messages, they will be added to a Mailtrap “inbox”

Mailables

- Next, generate a mail class:

```
user@debian-wms:~/code/howest-store-db$ php artisan make:mail OrderConfirmation
Mail created successfully.
user@debian-wms:~/code/howest-store-db$
```

- Mail class is created in ./app/Mail:



Mailables

- Mail: linking to a Blade file

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>

  <h1>Thank you</h1>

  <p>We have received your confirmation</p>

  <ul>
    <li>Restaurant: {{ $restaurant → name }}</li>
    <li>Address: {{ $restaurant → address }}</li>
    ...
  </ul>

</body>
</html>
```

```
class OrderConfirmation extends Mailable
{
    use Queueable, SerializesModels;

    private $restaurant;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct(Restaurant $restaurant)
    {
        $this → restaurant = $restaurant;
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this→view('emails.confirmation',
            ["restaurant" ⇒ $this → restaurant]);
    }
}
```

Mailables

- Sending the actual mail from within our controller

```
use Illuminate\Support\Facades\Mail;
```

```
$conf = new OrderConfirmation($restaurant);  
Mail::to($email) → send($conf);
```

```
class OrderConfirmation extends Mailable  
{  
    use Queueable, SerializesModels;  
  
    private $restaurant;  
  
    /**  
     * Create a new message instance.  
     *  
     * @return void  
     */  
    public function __construct(Restaurant $restaurant)  
    {  
        $this → restaurant = $restaurant;  
    }  
}
```


Mailables

The screenshot displays the Mailtrap web application interface. At the top, the browser tab is labeled 'Mailtrap - Safe Email Testing' and the address bar shows 'mailtrap.io/inboxes/571923/messages/1205503056'. The Mailtrap header includes the logo, navigation links for 'Shared Inboxes' and 'Billing', an 'Upgrade' button, and a user profile for 'frederic.vlummens@howest.be'. Below the header, a sidebar on the left lists a series of emails, all with the subject 'Thank you for your order' and the sender 'frederic.vlummens@howest.be'. The main content area shows a selected email titled 'Reservation Confirmation' from 'No Reply Howest <noreply@howest.be>' to 'frederic.vlummens@howest.be', dated '2019-03-28 11:41'. The email body is rendered in HTML, featuring the heading 'Restaurant Booker' and a 'Thank you' message to 'frederic.vlummens@howest.be'. It lists reservation details: 'Restaurant: The Seafood Buffet', 'Date: 2019-03-30', and 'Time: 18:30'. The footer of the email reads '© Howest Bruges'. The Mailtrap interface also includes a bottom bar with social media icons and a copyright notice for 'Railware Products, Inc.'.



Authentication

Authentication

- In a lot of web apps there is a need to authenticate users
- Usually done based on username and password
- Laravel has support for authentication, which can be enabled by installation additional packages/modules
- Uses sessions

Authentication: installation/scaffolding

- To add authentication, issue the following command **from within your app's directory**:

```
$ composer require laravel/ui  
$ php artisan ui vue --auth  
$ npm install && npm run dev
```

- These three commands (which will take some time to complete), take care of installation and scaffolding of the entire authentication mechanism.

Authentication: storing login info in database

- We need a place where we can store user information (usernames, passwords, ...)
- Solution: database!
- Laravel needs specific database tables (and associated models)
- Instead of creating these tables manually, we will **generate** them using **migrations**

Migrations

- Migrations are like version control for databases
- You can modify the database schema using code (=migration files)
- When running the migrations, the necessary changes are applied to the database
- Excerpt from Laravel.com:
“If you have ever had to tell a teammate to manually add a column to their local database schema, you've faced the problem that database migrations solve.”

Migrations

- The migration files to create the **users** and **password_resets** tables already exists and can be found in **./database/migrations**:

```
user@debian-wms: ~/code/auth-demo3/database/migrations
user@debian-wms:~/code$ cd auth-demo3
user@debian-wms:~/code/auth-demo3$ ls
app                composer.lock      package.json       README.md          storage
artisan           config             package-lock.json  resources          tests
bootstrap         database           phpunit.xml        routes             vendor
composer.json     node_modules       public             server.php         webpack.mix.js
user@debian-wms:~/code/auth-demo3$ cd database
user@debian-wms:~/code/auth-demo3/database$ cd migrations
user@debian-wms:~/code/auth-demo3/database/migrations$ ls
2014_10_12_000000_create_users_table.php
2014_10_12_100000_create_password_resets_table.php
2019_08_19_000000_create_failed_jobs_table.php
user@debian-wms:~/code/auth-demo3/database/migrations$
```

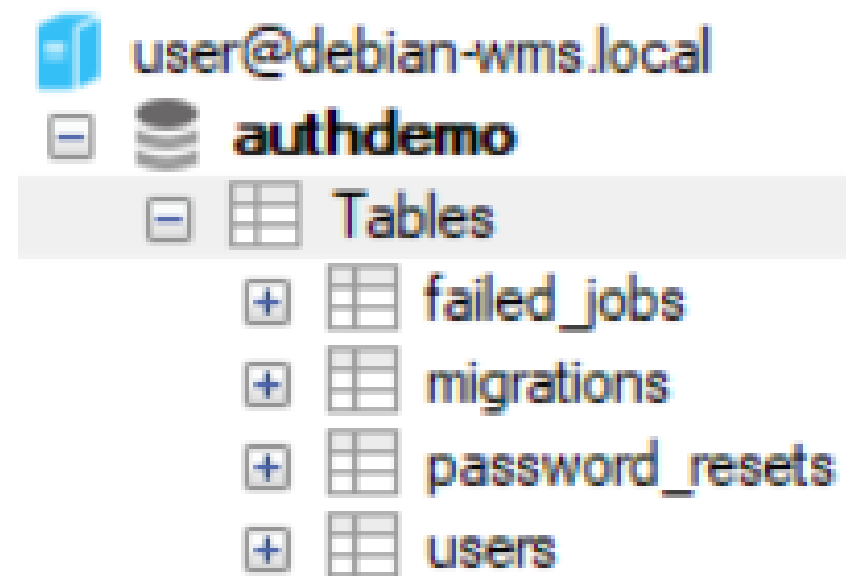
Running the migrations

- First, make sure you have a database associated with your project (.env file)!
- Next, run the migrations **from within your app directory** using the command **php artisan migrate**:

```
user@debian-wms: ~/code/auth-demo3
user@debian-wms:~/code/auth-demo3$
user@debian-wms:~/code/auth-demo3$
user@debian-wms:~/code/auth-demo3$ php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (0.01 seconds)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (0.01 seconds)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (0.01 seconds)
user@debian-wms:~/code/auth-demo3$
```

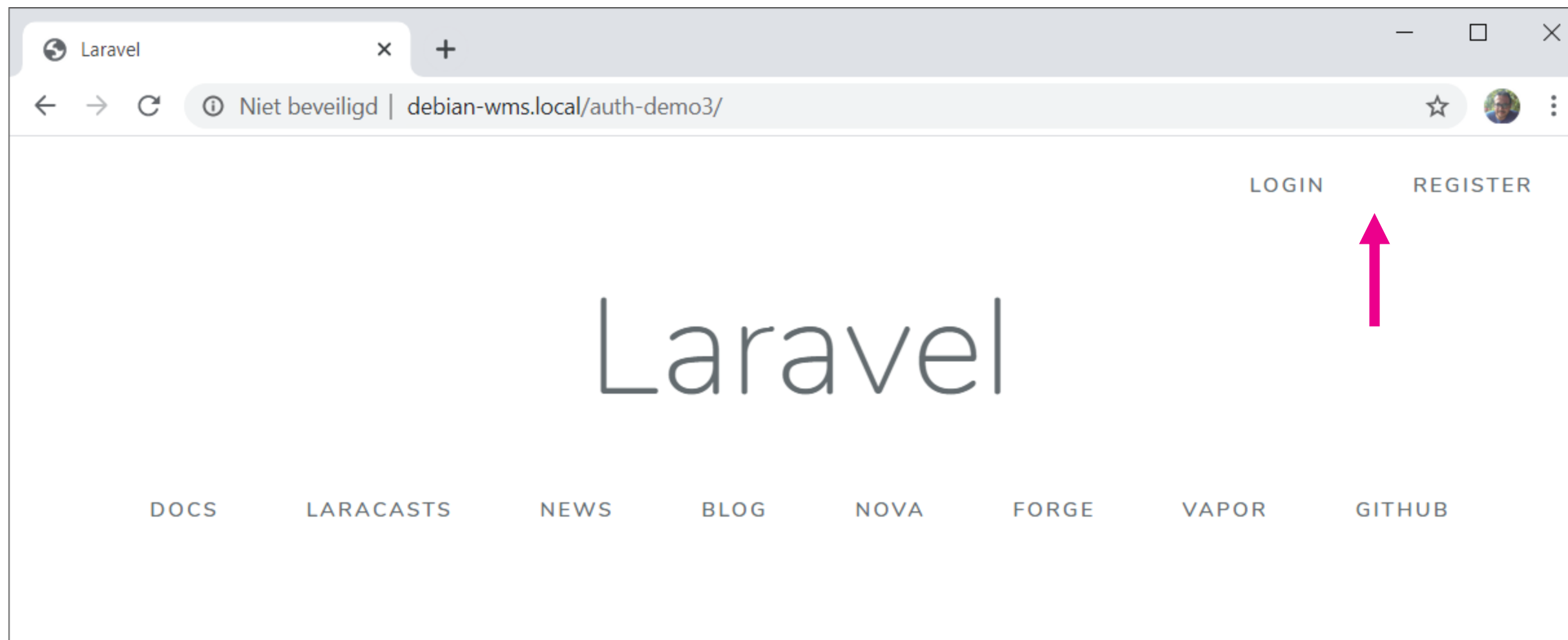
Running the migrations

- The tables with their corresponding columns have now been generated:



Using the generated code

- When you now access your site, you have **Login** and **Register** functionality available!



Studying the generated code

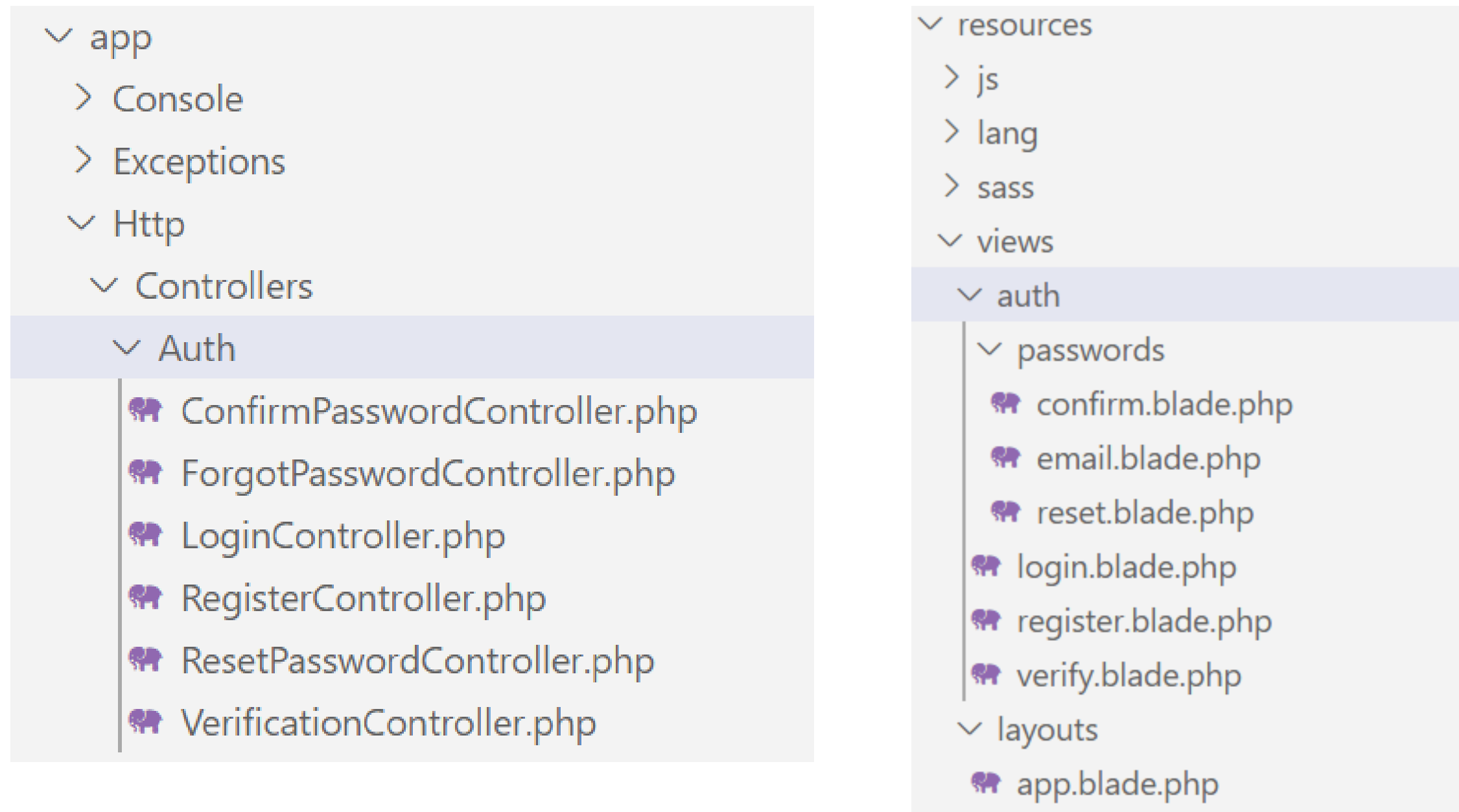
- Additional routes for authentication have been added in **./routes/web.php**:

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

```
Auth::routes(); ←
```

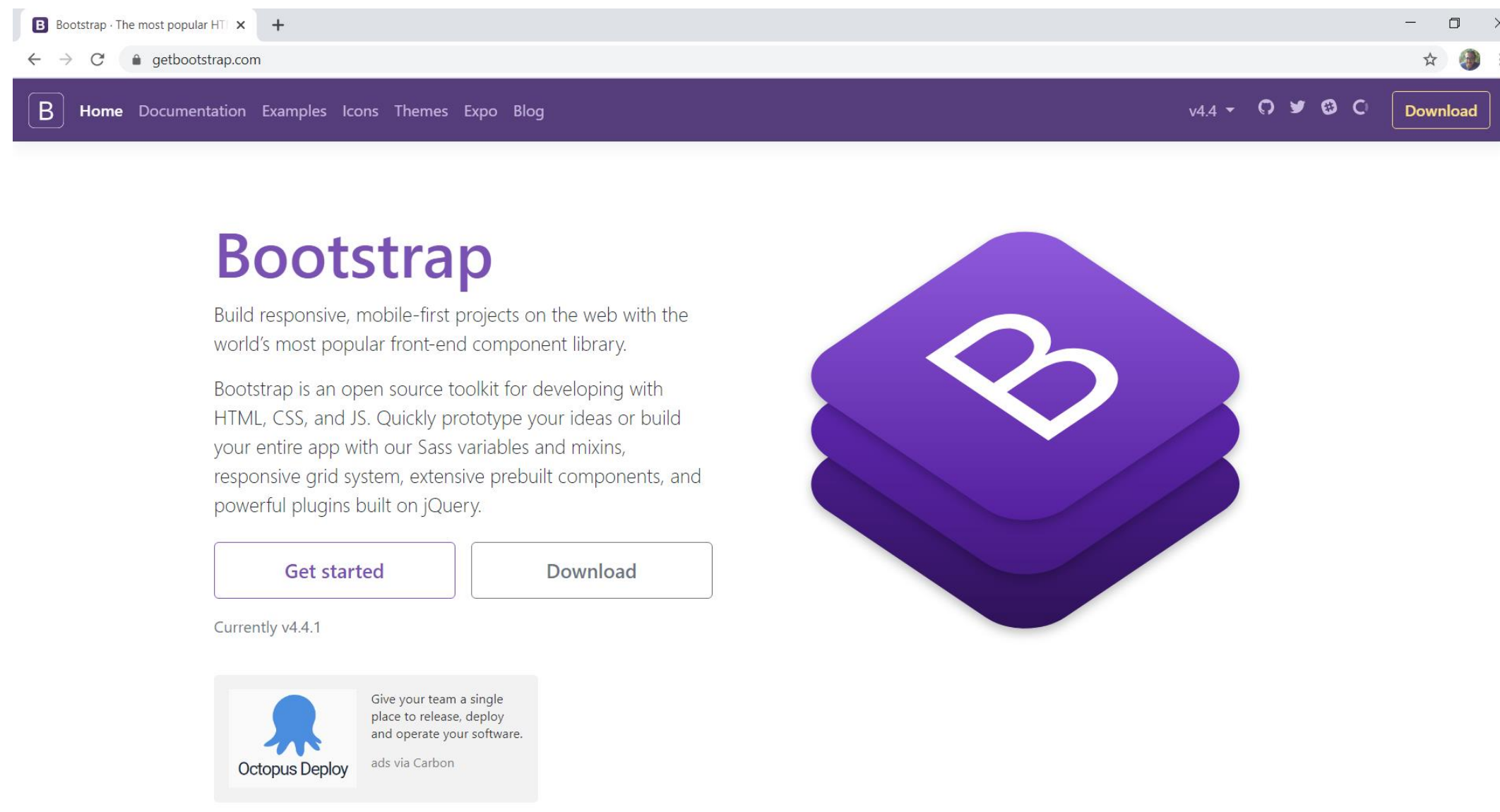
Studying the generated code

- Additional controllers and views have been generated:



Note...

- The generated views use **Bootstrap**



Note...

- The generated views use **Bootstrap**
- You can kick out the Bootstrap CSS and implement your own CSS

Note...

- The generated views use **Bootstrap**
- You can kick out the Bootstrap CSS and implement your own CSS

Views

As mentioned in the previous section, the `laravel/ui` package's `php artisan ui vue --auth` command will create all of the views you need for authentication and place them in the `resources/views/auth` directory.

The `ui` command will also create a `resources/views/layouts` directory containing a base layout for your application. All of these views use the Bootstrap CSS framework, but you are free to customize them however you wish.

Path customization

- After successful authentication, user is redirected to **/home**
- If you want another post-authentication redirect path, change the HOME constant in **./app/Providers/RouteServiceProvider.php**

```
protected $namespace = 'App\Http\Controllers';

/**
 * The path to the "home" route for your application.
 *
 * @var string
 */
public const HOME = '/home';
```

Useful methods for your controllers

- First, make sure to import the **Auth** facade:
use Illuminate\Support\Facades\Auth;
- Retrieve the currently authenticated user:
\$user = Auth::user();
- Retrieve the currently authenticated user's ID:
\$userId = Auth::id();
- Determine whether a user is authenticated:
\$isAuthenticated = Auth::check();
- Logging out a user:
Auth::logout();
- More info: <https://laravel.com/docs/master/authentication#retrieving-the-authenticated-user>

Protecting your routes

- Compare the following routes:

```
Route::get('/howest1', "HowestController@howest1") → middleware("auth");
```

```
Route::get('/howest2', "HowestController@howest2");
```

- Route **/howest1** will only be accessible if you are authenticated
- Route **/howest2** is also available to unauthenticated users

Displaying information in view only to logged in users

```
@auth
    <p>Welcome, {{ Auth::user() → email }}.</p>
@else
    <p>
        You need to <a href="{{ route('login') }}">login</a> to write a review.
        If you do not have an account, <a href="{{ route('register') }}">register here</a>.
    </p>
@endauth
```

or

```
@guest
    <p>
        You need to <a href="{{ route('login') }}">login</a> to write a review.
        If you do not have an account, <a href="{{ route('register') }}">register here</a>.
    </p>
@else
    <p>Welcome, {{ Auth::user() → email }}.</p>
@endguest
```

Questions?

