Inés Larrañaga Fernández de Pinedo	CSC Jesuitas - Logroño
	DWES

Table of Contents

Authentication

Before going into the implementation of the controllers functionality, let's include the Authentication functionality. Please make a copy of your `routes/web.php` file.

Laravel brings a lot of functionality related with authentication, such as: login, registration, password reset, email verification, and password confirmation. [Laravel Breeze](Starter Kits - Laravel - The PHP Framework For Web Artisans) libraries gives you minimal, but straight forward, functionalities.

You may install Laravel Breeze using Composer:

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

After Composer has installed the Laravel Breeze package, you may run the `breeze:install` Artisan command. This command publishes the authentication views, routes, controllers, and other resources to your application. Laravel Breeze publishes all of its code to your application

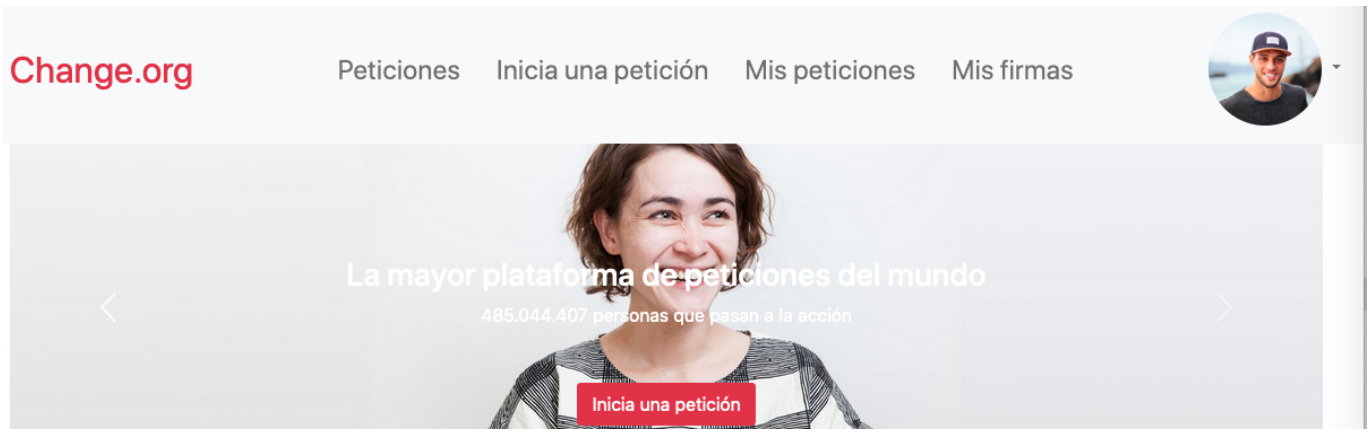
so that you have full control and visibility over its features and implementation. After Breeze is installed, you should also compile your assets so that your application's CSS file is available:

```
php artisan breeze:install

npm install
npm run dev
php artisan migrate
```

Next, you may navigate to your application's `/login` or `/register` URLs in your web browser. All of Breeze's routes are defined within the `routes/auth.php` file.

> Test your register/login/logout functionality.
>
> Adapt the views in order to make the visual integration with your layout. Test how the layout changes in case the user is logged in or not. You can show the username in case the user is logged in.



Peticiones functionalities

We are going to complete different functionalities we need for the public side.

Actions without the need of authentication

The actions of listing `Peticione` and showing the detail of any of them, will not need to an active session (logged in) to be executed. In order to avoid it, inside of the `PeticioneController` file:

```
    public function __construct()
    {
        $this->middleware('auth')->except(['index', 'show']);

    }
```

Now we are ready to make their implementation.

List action

We are going to create the index method in the recently created controller file:

```
public function index(Request $request)
{
    $peticiones = Peticione::all();
    return view('peticiones.index', compact('peticiones'));
}
```

Let's create the corresponding view file `resources/views/peticiones/index.blade.php`. The view should be similar to the original one.

Now that you have the controllers method created, add the corresponding routing inside the `routes/web.php`:

```
Route::resource('peticiones',\App\Http\Controllers\PeticioneController::class);
```

The resource method generates all the basic routes required for the `Peticiones` entity. In case you want to specify one by one you would do it like this:

```
Route::get('/peticiones/',[\App\Http\Controllers\PeticioneController::class,
'index']);
```

In order to now, which endpoints you have available, run this command on your command line:

```
 php artisan route:list
```

```
GET|HEAD   admin/pages/{id}/edit ... voyager.pages.edit › TCG\Voyager › VoyagerBaseController@edit
GET|HEAD   admin/pages/{id}/restore voyager.pages.restore › TCG\Voyager › VoyagerBaseController@r…
GET|HEAD   admin/peticiones . voyager.peticiones.index › TCG\Voyager › VoyagerBaseController@index
POST       admin/peticiones . voyager.peticiones.store › TCG\Voyager › VoyagerBaseController@store
POST       admin/peticiones/action voyager.peticiones.action › TCG\Voyager › VoyagerBaseControlle…
GET|HEAD   admin/peticiones/create voyager.peticiones.create › TCG\Voyager › VoyagerBaseControlle…
GET|HEAD   admin/peticiones/order voyager.peticiones.order › TCG\Voyager › VoyagerBaseController@…
POST       admin/peticiones/order voyager.peticiones.update_order › TCG\Voyager › VoyagerBaseCont…
GET|HEAD   admin/peticiones/relation voyager.peticiones.relation › TCG\Voyager › VoyagerBaseContr…
POST       admin/peticiones/remove voyager.peticiones.media.remove › TCG\Voyager › VoyagerBaseCon…
```

Sometimes, it is necessary to launch this command to update the routes of your application:

```
php artisan route:cache
```

> Now, on your own, modify the previous query in order to list all `Peticiones` with `estado="aceptada"` order by the creation date. Add the corresponding [link](URL Generation - Laravel - The PHP Framework For Web Artisans) in the layout using the corresponding method.
>
> For instance:
>
> ```
> <a class="nav-link fs-4 m-2" href="
> {{route('peticiones.index')}}">Peticiones</a>
> ```

View the detail of a peticion

In order to click on a `peticione` and view its detail. Create the corresponding `show` method in the controller and the view.

```
public function show(Request $request, $id)
{
    $peticion = Peticione::findOrFail($id);
    return view('peticiones.show', compact('peticion'));
}
```

> Please create a view with the same layout as before, add the corresponding links inside of the related views and check the routing files.

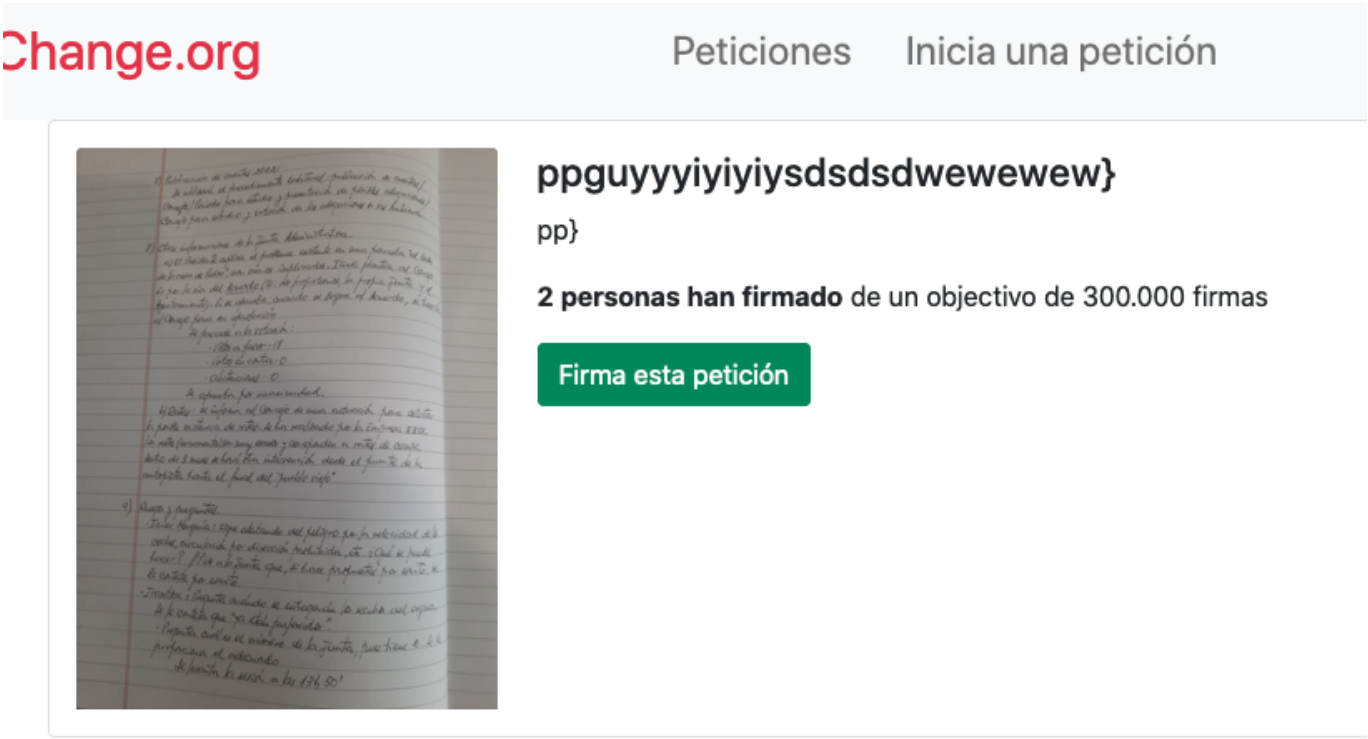
Actions with authentication

List mine action

> Now, following the previous examples, you have to create a new method where only the `Peticiones` of and specificic user are loaded. That means, the ones that were created by an specific user. The query will be very similar to the previous one but you will have to pass the `Auth::user()->id` to the query as parameter. That is the id of the user logged in.
>
> Use the same view as before, with the corresponding link in the layout and do not forget to include the method into the routing file.

Sign a Peticion

In order to sign a `Peticione`, you should generate the corresponding url like `.../peticiones/firmar/X`, where x should be replaced by the id of the `Peticione` a logged in user wants to sign. That link should be included in the `show` view of `Peticiones`.



Another thing you should check/create is the N-M relationship between `Peticiones` and `Users` through 'peticione_user' table:

```
public function firmas(){
     return $this->belongsToMany(User::class,'peticione_user');
}
```

After that, you should create the corresponding controller method:

```
    public function firmar(Request $request, $id)
    {
        try {
            $peticion = Peticione::findOrFail($id);
            $user = Auth::user();
            $firmas = $peticion->firmas;
            foreach ($firmas as $firma) {
                if ($firma->id == $user->id) {
                    return back()->withError( "Ya has firmado esta petición")-
>withInput();
                }
            }
            $user_id = [$user->id];
            $peticion->firmas()->attach($user_id);
            $peticion->firmantes = $peticion->firmantes + 1;
            $peticion->save();
        }catch (\Exception $exception){
            return back()->withError( $exception->getMessage())->withInput();
        }
        return redirect()->back();

    }
```

As you can see, one of the things we are doing is checking if that logged in user has already signed that peticion before. And we are obtaining the `Peticione` a logged in user has signed thanks to the relationship defined on the `User` model through the `firmas` method. After, in case that user has never signed that `Peticione` before, we associate that `Peticione` to the user through the relationship again. We increase in one unit the number of `firmantes` and update the object.

Do not forget to add the method to the routing file and test it:

```
Route::get('/peticiones/firmar/{id}',
[\App\Http\Controllers\PeticioneController::class, 'firmar']);
```

Add a new Peticione

Attending to Laravel convention, in case a user press an action that goes into a form that he/she will fill afterwards, the action that loads the view will be loaded thanks to a `create` method on the controller (GET method), and the action where the user sends to server the information filled when he press the submit button (POST action) will be processed thanks to the `store` action of the same controller

On the layout, we need to create a link that goes through the GET method and loads the view:

```
    public function create(Request $request)
    {
        return view('peticiones.edit-add');
    }
```

> Do not forget creating the corresponding view. The method should be added to the routing list in case you used the resource method. Test and check if the form is properly loaded.
>
> Apart from that we need to create an `store` method inside of `PeticioneController`, that will be in charge of the POST request. Here you have a first version of the ir, where we are not expecting to arrive the file yet:

```
    public function store(Request $request)
    {
        $this->validate($request, [
            'titulo' => 'required|max:255',
            'descripcion' => 'required',
            'destinatario' => 'required',
            //'file' => 'required',
        ]);

        $input = $request->all();

        $category = Category::findOrFail($input['category']);
        $user = Auth::user(); //asociarlo al usuario authenticado
        $peticion = new Peticione($input);
        $peticion->user()->associate($user);
        $peticion->category()->associate($category);

        $peticion->firmantes = 0;
        $peticion->estado = 'pendiente';
        $peticion->save();
        return redirect('/mispeticiones');

    }
```

As you can see, the incoming request is validated against some rules. Those rules are always needed from a server side point of view. **YOU CANNOT MAKE ANY UPDATE OR INSERT IN THE DATABASE WITHOUT VALIDATING IT FIRST, EVEN IF YOU ALREADY DID IT ON THE CLIENT**.

Check this Validation - Laravel - The PHP Framework For Web Artisans, on how these validation rules are defined in laravel.

Before testing it, check that inside your `App\Models\Peticione.php` model, you have the corresponding relationships defined:

```
    public function user()
    {
        return $this->belongsTo(User::class);
    }

    public function category()
    {
        return $this->belongsTo(Category::class);
    }

    public function firmas()
    {
        return $this->belongsToMany(User::class, 'peticione_user');
    }
```

And the other way round, inside `App\Models\User.php`:

```
    public function peticiones()
    {
        return $this->hasMany(Peticione::class);
    }

    public function firmas()
    {
        return $this->belongsToMany(Peticione::class, 'peticione_user');
    }
```

> Exercise1:Test the store action properly filling the form and sending the information to the server.
>
> Exercise2: On your own modify the view in order to send a file. Moreover, on the store method you should validate if the incoming file is valid(required, image type and 5MB as its maxsize) and in case it is valid, save it on the database (File model) with the related peticione_id and the path and name of the file. The file should be saved on the system storage, that is why you will just save the path and name of the file in the database.

List of signed Peticiones by an specific user

This method will be similar to the previous list methods. We need the logged in user, and once you have it take its `Peticiones` from the belongstomany relationship.

In order to do so, crete the `VoyagerUsersController` class using artisan:

```
php artisan make:controller VoyagerUsersController
```

And inside include this code (where `$id` will be replaced by the authenticated user in the near future:

```
    public function peticionesFirmadas(Request $request)
    {
        $id = Auth::id();
        $usuario = User::findOrFail($id);
        $peticiones = $usuario->firmas;
        return view('peticiones.index', compact('peticiones'));

    }
```

As you can see, making use of `$usuario->firmas` we are calling to the belongsToMany method defined in the `User` model.

```
    public function firmas()
    {
        return $this->belongsToMany(User::class, 'peticione_user');
    }
```

Create a link to this action in the layouts menu.Finally, include the method into the routing file and test it.

Error handling

We need to review how the errors are displayed to the user in case they occur. But that depends on the kind of error:

Functional errors

Imagine there is an error on a field in a form, we need to return back to the same page giving back some information to the user in order he/she corrects it. **REMEMBER, we make**

validations not only on the frontend, also on the BACKEND, and it is even more important.

For instance, inside of our store method from the controller:

```
public function store(Request $request)
    {
        $this->validate($request, [
            'titulo' => 'required|max:255',
            'descripcion' => 'required',
            'destinatario' => 'required',
            'category'=>'required',
            'file' => 'required',
        ]);

        $input = $request->all();

        try {
            $category = Categoria::findOrFail($input['category']);
            $user = Auth::user(); //asociarlo al usuario authenticado
            $peticion = new Peticione($input);
            $peticion->categoria()->associate($category);
            $peticion->user()->associate($user);

            $peticion->firmantes = 0;
            $peticion->estado = 'pendiente';

            $res=$peticion->save();
            if ($res) {
                $res_file = $this->fileUpload($request, $peticion->id);
                if ($res_file) {
                    return redirect('/mispeticiones');
                }
                return back()->withError( 'Error creando la peticion')-
>withInput();
            }
        }catch (\Exception $exception){
            return back()->withError( $exception->getMessage())->withInput();
        }
    }
```

And inside of the `edit-add.blade.php` we can give information for every wrong field:

```
<div class="col-md-8">
     <label for="validationServer01" class="form-label">Titulo</label>
     <input type="text" name="titulo" class="form-control  @error('titulo') is-
invalid @enderror" id="validationServer01" >
     @error('titulo')
     <div class="alert alert-danger">{{ $message }}</div>
     @enderror
</div>
```

Technical errors

This could be the case where the DB server is down, or in case we try to find out an identified that does not exist on the DB. In order to control those kind of errors, we will need to modify an some controller methods like this:

```
public function show(Request $request, $id)
{
        try{
            $peticion = Peticione::findOrFail($id);
        }catch (\Exception $exception){
            return back()->withError( $exception->getMessage())->withInput();
        }
        return view('peticiones.show', compact('peticion'));
}
```

And inside of the views, we will print a flash error (an error saved in session that is displayed only once):

```
@if (session('error'))
    <div class="alert alert-danger">{{ session('error') }}</div>
@endif
```

> Now, on your own, refactor EVERY CONTROLLER METHOD to catch possible errors, and make the corresponding Flash error display on the views

Custom error pages

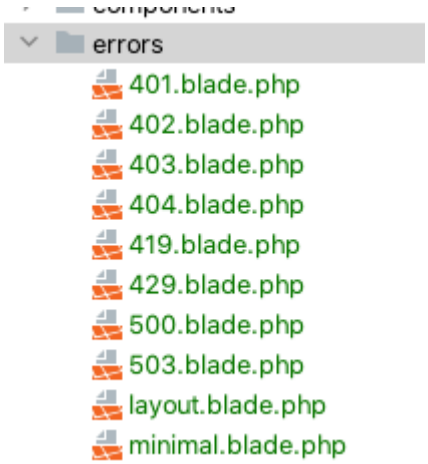
Imagine that you try writing a url that does not exist, it could be a good idea to customize the error pages that appear to the user.

- **Publish Error Page Default Files:** In this step, we will run laravel command to create default error page blade file. when you run below command then laravel will create "errors"

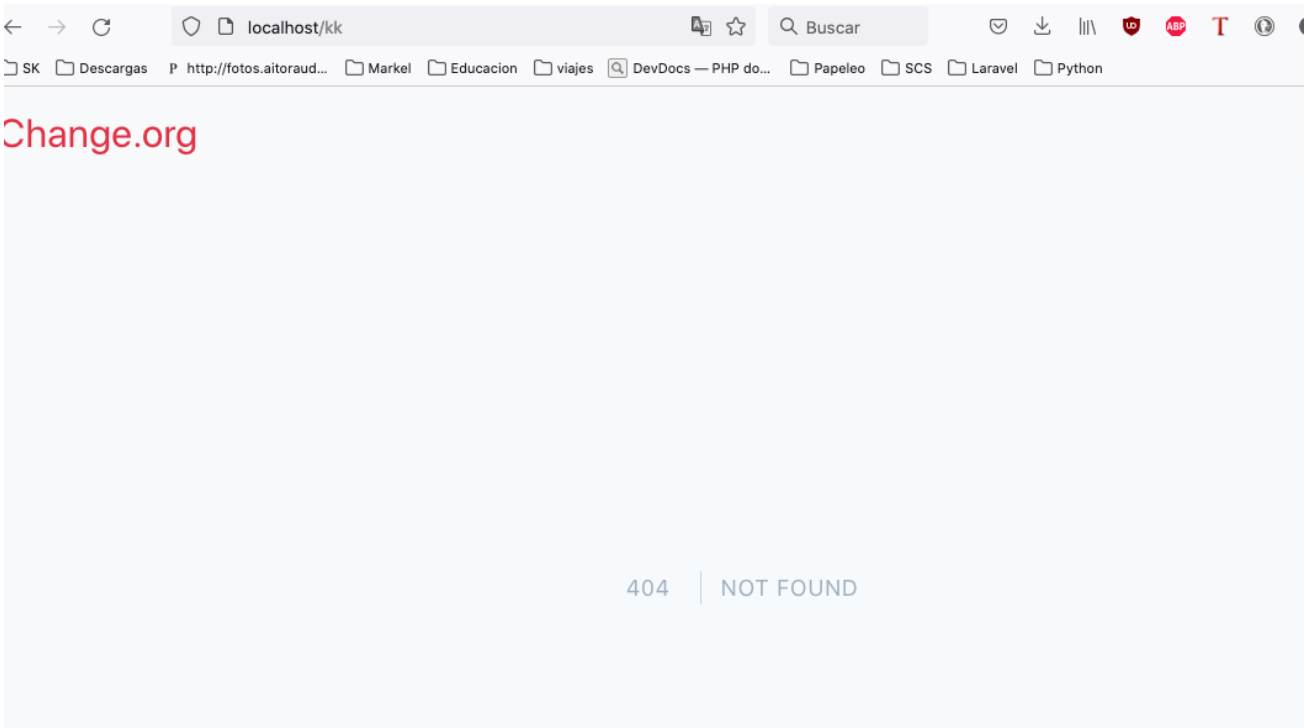
directory with all error pages inside views folder. so, let's run below command:

```
php artisan vendor:publish --tag=laravel-errors
```

In your Laravel project folder, in your `/resources/views/errors` folder you will see the default error pages created:



- **Update Error Pages Design**: You can update 404 error page design with de layout of your page or any similar design you could have. This error occurs when a resource is not found, if you try accessing a url that does not exist, this page should be prompted:



Pagination

Pagination is a way of showing extensive data in smaller chunks. In Laravel is very easy to use; it is integrated with the query builder and Eloquent ORM. Laravel pagination covers limit and offset automatically.

Let's change the `index` method of our `PeticioneController`:

```
    public function index(Request $request)
    {
        $peticiones = Peticione::paginate(5);
        return view('peticiones.index', compact('peticiones'));
    }
```

We have remove `all()` and use `paginate()` and it takes a number as an argument, that number defines the number of results to be shown to the user.

To display the pagination component on the view, we need to add the following code in the corresponding view:

```
    {!! $peticiones->links() !!}
```

> Exercise: on your own change also the `listMine` and `peticionesFirmadas` methods and related views, in order the result to be paginated.