# Laboratorio Sistema de opiniones en Laravel 9

[Laboratorio Sistema de opiniones en Laravel 9 1](#_Toc155530304)

[Introducción 2](#_Toc155530305)

[Base de datos inicial 2](#_Toc155530306)

[Datos 2](#_Toc155530307)

[Tabla Posts 2](#_Toc155530308)

[Relaciones entre tablas 3](#_Toc155530309)

[Desarrollo 4](#_Toc155530310)

[Ruta y controlador 4](#_Toc155530311)

[Estructura de un formulario 5](#_Toc155530312)

[Diseño de un formulario 6](#_Toc155530313)

[Middleware Auth 7](#_Toc155530314)

[Guardar en base de datos 8](#_Toc155530315)

[Componentes Blade 10](#_Toc155530316)

[Listado de registros 10](#_Toc155530317)

[Optimización de consulta 12](#_Toc155530318)

[Eliminar de la base de datos 13](#_Toc155530319)

[Políticas de acceso 14](#_Toc155530320)

[Políticas de acceso básico 14](#_Toc155530321)

[Política de acceso centralizado 15](#_Toc155530322)

[Política de acceso estándar 16](#_Toc155530323)

[Mejoras 18](#_Toc155530324)

[Filtrado de registros 18](#_Toc155530325)

[Módulo de idiomas 19](#_Toc155530326)

[Añadir datos fake 20](#_Toc155530327)

[Crear el Archivo PostFactory 21](#_Toc155530328)

[Cómo Sembrar la Base de Datos 23](#_Toc155530329)

[Archivo PostSeeder 23](#_Toc155530330)

[Archivo UserSeeder 24](#_Toc155530331)

[Insertando datos fake 24](#_Toc155530332)

[Restablecer la Base de Datos 24](#_Toc155530333)

[Test propio de Laravel 25](#_Toc155530334)

[Resumen final 25](#_Toc155530335)

[Conclusiones finales 27](#_Toc155530336)

## Introducción

Siguiendo el vídeo no puedo realizar la instalación

Lo consigo instalando Composer y después ejecutando el comando

composer create-project laravel/laravel publicaciones desde la carpeta htdocs.

Ahora sí seguimos las instrucciones del vídeo.

Composer require laravel/breeze --dev

Php artisan breeze:install

Ahora instalamos Node.js para poder continuar con los comandos npm e instalar los elementos necesarios para el proyecto de laravel

Npm install

Npm run dev

Finaliza de forma distinta al vídeo y no crea los css en la carpeta public. Todo lo demás parece que está correcto.

### Base de datos inicial

Creamos una base de datos vacía con el nombre de “publicaciones” con nuestro cliente de mysql que es PhpmyAdmin.

Establecemos en el fichero .env las variables de entorno de la base de datos(nombre, usuario, password)

Php aristan migrate

Y se crean las tablas que se definen en los archivos de la carpeta migrations.

Conclusión

Con algunos problemas al principio de la instalación, que se han solucionado fácilmente, hemos creado el proyecto y la base de datos. Vamos introduciéndonos en el manejo de los comandos de artisan.

## Datos

### Tabla Posts

php artisan make:model Post -m (este comando crea la clase Post y, al añadir el parámetro –m crea también el archivo de migraciones database\migrations\2024\_01\_04\_103911\_create\_posts\_table.php). Se creará la tabla la próxima vez que se ejecute “php aristan migrate”.

En este archivo se crea el esquema para la creación de la tabla “posts”

Añadimos las instrucciones para la creación de un campo user\_id en la tabla posts y establecemos una clave ajena en el campo user\_id que se relaciona con el campo “id” de la tabla “users”.

$table->unsignedBigInteger('user\_id');

            $table->foreign('user\_id')->references('id')->on('users');

            $table->text('body');

Ejecutamos de nuevo php artisan migrate y se crea la tabla en la base de datos.

Comprobamos que se ha creado la tabla y que no nos permite introducir datos si no hay un usuario debido a las restricciones de la clave ajena.

Php artisan migrate:refresh

Genera todas las tablas de la base de datos destruyendo las anteriores.

Añadimos un usuario mediante el formulario Register y comprobamos en la Base de Datos que se puede añadir registros en la tabla “posts” ahora que ya existe un usuario.

en database\migrations\2024\_01\_04\_103911\_create\_posts\_table.php añadimos ->onDelete(‘cascade’) para que al borrar un usuario se borren también todos los registros de otras tablas que estén asociados con una clave ajena.

 $table->foreign('user\_id')->references('id')->on('users')->onDelete('cascade');

Php artisan migrate:refresh

Se crean de nuevo las tablas. Creamos un usuario mediante el formulario register y luego un post en la base de datos. Comprobamos que se puede borrar el usuario y se eliminan los posts asociados a éste.

### Relaciones entre tablas

Lo siguiente es establecer en Laravel la multiplicidad entre las clases Post y User ya que un usuario pude tener muchos posts pero cada post sólo puede pertenecer a un usuario.

Para ello modificamos las clases User y Post de la siguiente manera:

En la clase User Añadimos la función posts()

public function posts()

    {

        return $this->hasMany(Post::class);

    }

En la clase post la función user

public function user()

    {

        return $this->belongsTo(User::class);

    }

Y añadimos en Post

protected $fillable = ['body'];

para permitir la asignación en masa en el campo body ya que en Laravel no se permite por defecto.

Conclusión:

Hemos visto cómo se cómo se crea un modelo o clase y crean las tablas de la base de datos dese Laravel, así como las relaciones entre éstas. Observamos que una vez creada la base de datos en blanco, todo lo demás se puede hacer desde el framework.

## Desarrollo

### Ruta y controlador

Escribimos el comando php artisan make:controller PostController para crear el archivo del controlador PostController.php

class PostController extends Controller

{

    public function index()

    {

        return 'index';

    }

    public function store()

    {

        //guardar;

    }

    public function destroy()

    {

        //eliminar

    }

}

Añadimos las funciones index, store y destroy y pasamos a crear las rutas en el archivo routes/web.php

Añadimos

use App\Http\Controllers\PostController;

para importar el controlador que hemos definido.

Y añadimos las rutas, una para cada método que hemos creado.

Route::get('posts',[PostController::class, 'index'])->name('posts.index');

Route::post('posts',[PostController::class, 'store'])->name('posts.store');

Route::delete('posts/{post}', [PostController::class, 'destroy'])->name('posts.index.destroy');

Añadimos en la vista una entrada en la barra de navegación que redirige a la ruta posts.index

En el archivo resources/views/layout/navigation.blade.php

Añadimos en Navigation links

  <x-nav-link :href="route('posts.index')" :active="request()->routeIs('posts.index')">

                        {{ \_\_('Post') }}

                    </x-nav-link>

y terminamos añadiendo una entrada para el botón de hamburguesa

  <x-responsive-nav-link :href="route('posts.index')">

                        {{ \_\_('Posts') }}

                    </x-responsive-nav-link>

### Estructura de un formulario

Modificamos la clase PostsController para que el método store() devuelva una vista.

 public function index()

    {

        return view('posts.index');

    }

Creamos el archivo de la vista en resources/views/posts/index.blade.php

Tomamos como plantilla el contenido del archivo resources/views/dashboard.blade.php y lo modificamos.

Quedará de la siguiente forma

<x-app-layout>

    <x-slot name="header">

        <h2 class="font-semibold text-xl text-gray-800 leading-tight">

            {{ \_\_('Posts') }}

        </h2>

    </x-slot>

    <div class="py-12">

        <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">

            <div class="bg-white overflow-hidden shadow-sm sm:rounded-lg">

                <div class="p-6 text-gray-900">

                    Formulario

                </div>

            </div>

        </div>

    </div>

</x-app-layout>

Añadimos una directiva para hacer el formulario seguro

  <form action="{{route('posts.store')}}" method="POST" >

                    @csrf

                    FORMULARIO

                   </form>

La directiva es @csrf y genera un campo oculto con un token.

### Diseño de un formulario

Creamos un archivo en resources/views/components/textarea.blade.php para definir el campo body en el formulario

<textarea

    rows="2"

    {!!

        $attributes->merge([

            'class' => 'border-gray-300 focus:border-indigo-500 focus:ring-indigo-500 rounded-md shadow-sm'

            ])

    !!}>

</textarea>

En el archivo resources/views/index.blade.php modificamos el formulario que teníamos creado añadiéndole un campo textarea.

  <form action="{{route('posts.store')}}" method="POST" >

                    @csrf

                    <div class="mt-4">

                        <x-input-label for="body" :value="\_\_('Body')" />

                        <x-textarea class="block mt-1 w-full" name="body" required />

                    </div>

                   </form>

Procedemos a crear un botón Save para el formulario

 <div class="flex justify-end mt-4">

                        <x-primary-button>

                             {{ \_\_('Save') }}

                        </x-primary-button>

                    </div>

### Middleware Auth

En esta sección vamos a eliminar código innecesario para esta aplicación y a agregar seguridad mediante Middleware Auth

Para ello modificamos el archivo de rutas routes\auth.php y eliminamos las que no se van a usar. Quedará de la siguiente manera.

use App\Http\Controllers\Auth\AuthenticatedSessionController;

use App\Http\Controllers\Auth\RegisteredUserController;

use Illuminate\Support\Facades\Route;

Route::middleware('guest')->group(function () {

Route::get('register', [RegisteredUserController::class, 'create'])

                ->name('register');

Route::post('register', [RegisteredUserController::class, 'store']);

Route::get('login', [AuthenticatedSessionController::class, 'create'])

                ->name('login');

Route::post('login', [AuthenticatedSessionController::class, 'store']);

});

Route::middleware('auth')->group(function () {

Route::post('logout', [AuthenticatedSessionController::class, 'destroy'])

                ->name('logout');

});

Eliminamos la página de bienvenida resources/views/welcome.blade.php

Como ya no existe la vista welcome vamos a establecer una redirección en el archivo routes\web.php

Route::redirect('/', 'dashboard'); redirige todo lo que vaya a la raíz ‘/’ a la ruta dashboard.

Route::get('dashboard',[PostController::class, 'index'])->middleware(['auth', 'verified'])->name('dashboard'); indica que dashboard es la ruta para index. Además queremos que el dashboard sea directamente la vista de publicaciones. Para ello modificamos el archivo routes\web.php

Route::redirect('/', 'dashboard');

Route::middleware('auth')->group(function () {

    Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');

    Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');

    Route::delete('/profile', [ProfileController::class, 'destroy'])->name('profile.destroy');

});

Route::get('dashboard',[PostController::class, 'index'])->middleware(['auth', 'verified'])->name('dashboard');

Route::post('posts',[PostController::class, 'store'])->middleware(['auth', 'verified'])->name('posts.store');

Route::delete('posts/{post}', [PostController::class, 'destroy'])->name('posts.index.destroy');

require \_\_DIR\_\_.'/auth.php';

En las rutas hemos añadido el método Middleware Auth para protegerlas con autenticación, es decir que solo permite el acceso a usuarios logueados. Si no lo está se redirige a la página de login.

Ahora la ruta que no funciona es posts.index a la que hacíamos referencia en resources\views\layouts\navigation.blade.php. En ésta habíamos incorporado una entrada en la barra de navegación y otra en el botón hamburguesa.

Las eliminamos y ahora en la página de dashboard aparece directamente la página index.blade.php que habíamos creado con el formulario.

### Guardar en base de datos

Lo primero que vamos a establecer son sistemas de validación de datos. En el formulario hemos configurado el parámetro required para que sea obligatorio rellenarlo. Esto solo es una validación en html, para hacer una segunda validación en php modificaremos el archivo app\Http\Controllers\PostController.php que es el controlador que habíamos creado al principio.

public function store(Request $request)

    {

        $request->validate(['body'=>'required']);

        return $request->all();

    }

Añadimos $request->validate(['body'=>'required']); con lo que es obligatorio rellenar el campo antes de enviar el formulario esté o no esté el parámetro required en el textarea del formulario.

Ahora en el vídeo hay que utilizar un componente auth-validation-errors.php que no existe en mi sistema.

Como en el vídeo se ve claro el código creo el componente auth-validation-errors.php

@props(['errors'])

@if ($errors->any())

    <div {{ $attributes }}>

        <div class="font-medium text-red-600">

            {{ \_\_('Whoops! Something went wrong.')}}

        </div>

        <ul class="mt-3 list-disc list-inside text-sm text-red-600">

            @foreach ($errors->all() as $error)

            <li>{{ $error }}</li>

            @endforeach

        </ul>

    </div>

@endif

Y lo llamamos desde el index

    <x-auth-validation-errors class="mb-4" :errors="$errors" />

También añadimos

<x-auth-session-status class="mb-4" :status="session('status')" />

Entre los dos nos devolverán mensajes tanto de error como de status desde el formulario

Modificamos la función store() que hemos creado en el controlador app\Http\Controllers\PostController.php

 public function store(Request $request)

    {

        $request->validate(['body'=>'required']);

        $request->user()->posts()->create($request->only('body'));

        return back()->with('status', 'Publicación guardada exitosamente');

    }

Donde podemos ver

$request->validate(['body'=>'required']); Se requiere que el campo body contenga algo.

$request->user()->posts()->create($request->only('body'));

Este es el que indica que se cree en la base de datos un post perteneciente a un usuario, tomando solamente el contenido del campo “body”.

return back()->with('status', 'Publicación guardada exitosamente');

En el return se añade el mensaje que se almacena en la variable de sesión ”status”

Una vez realizados estos cambios comprobamos con la aplicación que se guardan correctamente los posts en la base de datos.

### Componentes Blade

Primero cambiamos los nombres de los dos componentes auth que hemos utilizado quedando de la siguiente manera:

resources\views\components\session-status.blade.php

resources\views\components\validation-errors.blade.php

Asimismo modificamos las llamadas a estos componentes

<x-session-status class="mb-4" :status="session('status')" />

<x-validation-errors class="mb-4" :errors="$errors" />

En el vídeo aparecen en todos los archivos de la carpeta auth pero en mi sistema no.

Procedo a añadirlo en todos los archivos de esta carpeta.

Ahora vamos a añadir un helper en el controlador que hemos creado PostController como ejemplo de una herramienta que nos ayuda en la depuración: dd()

dd($request->only('body'));

Nos muestra un array con el índice ‘body’ y el valor del campo;

dd($request->body);

Nos muestra una cadena con el valor del campo ‘body’

dd(['body'=>$request->body]);

Nos muestra un array como el del primer helper

Los dejamos comentados en el controlador como información útil en el futuro.

        //dd($request->only('body'));

        //dd($request->body);

        //dd(['body'=>$request->body]);

### Listado de registros

Creamos una carpeta dentro de views con el nombre inc

resources\views\inc

Creamos dentro de ella dos archivos

resources\views\inc\form.blade.php

resources\views\inc\list.blade.php

en form.blade.php introducimos el formulario creado en index.blade.php

<x-session-status class="mb-4" :status="session('status')" />

<x-validation-errors class="mb-4" :errors="$errors" />

    <form action="{{route('posts.store')}}" method="POST" >

     @csrf

        <div class="mt-4">

            <x-input-label for="body" :value="\_\_('Body')" />

            <x-textarea class="block mt-1 w-full" id="body" name="body"  />

        </div>

        <div class="flex justify-end mt-4">

           <x-primary-button>

               {{ \_\_('Save') }}

           </x-primary-button>

        </div>

    </form>

En index.blade.php borramos el formulario y lo sustituimos por una llamada a form.blade.php

@include('posts.inc.form')

Como lo hemos dirigido hacia la carpeta posts/inc movemos la carpeta inc que hemos creado y la ponemos dentro de la carpeta posts. En mi caso lo he hecho arrastrando directamente la carpeta en VisualStudioCode.

Añadimos otro div en index.blade.php para mostrar el listado

<div class="p-6 text-gray-900">

                    @include('posts.inc.form')

                </div>

                <div class="p-6">

                    @include('posts.inc.list')

                </div>

En el archivo resources\views\posts\inc\list.blade.php ponemos simplemente “Listado” para comprobar su funcionamiento

Listado

Comprobamos y funciona.

Para poder obtener el listado de datos lo primero es crear la consulta en el archivo del controlador.

Añadimos al principio la importación de la clase Post

use App\Models\Post;

Ahora ya podemos generar la consulta modificando la función index.

 public function index()

    {

        return view('posts.index', [

            'posts'=>Post::latest()->paginate()

        ]);

    }

Hemos añadido después de la ruta

['posts'=>Post::latest()->paginate()]);

Para que devuelva un array con los últimos posts.

La vista del listado la hacemos en el archivo resources\views\posts\inc\list.blade.php

@foreach ($posts as $post)

<div class="mt-4">

    <a href="#" class="text-lg font-semibold">{{$post->user->name}}</a>

    <p class="mt-1 text-xs"></p>

    <em>

        {{ $post->created\_at->format('d,m,Y')}}

    </em>

    {{ $post->body }}

</div>

<div class="mt-4">

    {{ $posts->links() }}

</div>

@endforeach

Por cada post formatea la salida en html mostrando el nombre del usuario, la fecha del post y el contenido del mismo. Al final se crea un div para los enlaces de la paginación.

### Optimización de consulta

Vamos a instalar un componente para poder evaluar el rendimiento de las consultas en el sistema.

Escribimos en el terminal.

Composer require barryvdh/laravel-debugbar -dev

Una vez instalado el componente podemos ver en el navegador que aparece una barra inferior con distintas informaciones.

Al mirar en la pestaña queries vemos que se está generando una consulta por cada una de los posts que son del mismo usuario. Esto es porque estas consultas se están generando en las vistas y no en el controlador, lo que supone una carga extra de trabajo al sistema.

Para evitarlo añadimos una nueva propiedad a la clase Post

protected $with=['user'];

para que cuando el controlador nos devuelva los datos de los posts éstos incluyan al usuario.

Comprobamos en el navegador gracias a la barra instalada que las consultas se han reducido a 4, que no aumentan aunque añadamos más posts, y que todas se generan desde el controlador.

### Eliminar de la base de datos

Vamos a crear un formulario que contenga un botón Delete para que aparezca debajo de cada post

 <form action="{{ route('posts.destroy', $post->id) }}" method="POST">

    @csrf

    @method('DELETE')

    <button class="text-indigo-600 text-xs">{{ \_\_('Delete')}}</button>

    </form>

Comprobamos que el botón aparece pero nos dirige a una página en blanco, ya que estamos llamando a la función delete del controlador que está definida pero no tiene código.

Procedemos a escribir el código para que la función elimine un post de la base de datos

Hemos añadido las directivas

@csrf para la seguridad de autenticación y @method('DELETE') para especificar el método que se usará.

Modificamos la función delete

public function destroy(Post $post)

    {

        $post->delete();

        return back();

    }

Añadiendo parámetro de entrada Post $post, la consulta $post->delete() y un return que nos dirige a la vista anterior.

Comprobamos en el navegador eu efectivamente se eliminan los posts al hacer click en el botón delete.

Con el fin de mejorar el código con archivos lo más pequeño posibles vamos a pasar todo el formulario que hemos creado a un fichero resources/views/posts/inc/ítem.blade.php.

De esta forma el archivo index.blade.php queda sólo con la llamada a ítem

@foreach ($posts as $post)

<div class="mt-4">

    @include('posts.inc.item')

</div>

@endforeach

<div class="mt-4">

    {{ $posts->links() }}

</div>

Conclusión:

En este apartado vemos cómo funciona el sistema controlador=>ruta=>vista que es la base sobre la que funciona todo.

El controlador es el que genera la consulta, la ruta especifica el tipo (get, post, delete, etc..) y lo asocia a un método del controlador y le asigna un nombre de ruta. Las vistas se construyen llamando a las rutas. Laravel proporciona métodos como Middleware Auth que hemos usado para validar la autenticación, es decir que solo permite el acceso a usuarios logueado, o funciones como @csrf que genera un token oculto en los campos de un formulario. Esto facilita mucho la tarea del desarrollador.

Hemos visto cómo crear y usar componentes que nos han permitido mostrar el usuario mensajes tanto de error como de confirmación. Asimismo hemos visto herramientas como el debug-bar de barryvdh que nos ayuda a ver cómo está funcionando todo el sistema y detectar en qué puntos se puede optimizar. En este caso hemos detectado que se producían demasiadas consultas al hacerse desde las vistas y se ha corregido haciendo que las consultas vengan todas desde el controlador

## Políticas de acceso

En este momento, cualquier usuario puede eliminar posts de otros usuarios lo que es un fallo de seguridad que resolveremos mediante las políticas de acceso.

### Políticas de acceso básico

Vamos a establecer una restricción para que sólo puedan eliminarse los posts propios de un usuario

Modificamos la función destroy() del controlador

public function destroy(Request $request, Post $post)

    {

        if($request->user()->id != $post->user\_id){

            abort(403);

        }

        $post->delete();

        return back();

    }

Indicamos que se aborte la petición si el usuario que está con la sesión activa no es el autor del post. Además se redirigirá a una página de error 403 Forbidden.

### Política de acceso centralizado

Ahora vamos a tratar de centralizar la política de acceso que hemos creado. Para hacerlo usaremos el archivo app\Providers\AuthServiceProvider.php que contiene el modelo manejador de las autenicaciones.

Importamos las clases User y Post

use App\Models\User;

use App\Models\Post;

Modificamos la función boot()

 public function boot()

    {

        $this->registerPolicies();

        Gate::define('destroy-posts',function(User $user, Post $post){

            return $user->id===$post->user\_id;

        });

    }

En ésta función definimos la política ‘destroy-posts’ como una función que nos devolverá verdadero si el usuario activo en la sesión es el usuario autor del post , o falso si no lo es.

Ahora vamos a sustituir la política que habíamos incluido en el archivo del controlador

 public function destroy(Request $request, Post $post)

    {

        if($request->user()->id != $post->user\_id){

            abort(403);

        }

        $post->delete();

        return back();

    }

Por una llamada directa a la nueva política creada

 public function destroy(Request $request, Post $post)

    {

        $this->authorize('destroy-post',$post);

        $post->delete();

        return back();

    }

También deberemos hacer cambios en la vista para que no salga el botón delete si el post no es del usuario activo.

El archivo resources\views\posts\inc\item.blade.php queda así:

Encerrando el formulario en un @can(‘destroy-post, $post) @endcan que evalúa la política en cada impresión de un post.

    <a href="#" class="text-lg font-semibold">{{$post->user->name}}</a>

    <p class="mt-1 text-xs">

    <em>

        {{ $post->created\_at->format('d,m,Y')}}

    </em>

    {{ $post->body }}

    </p>

    @can('destroy-post', $post)

    <form action="{{ route('posts.destroy', $post->id) }}" method="POST">

    @csrf

    @method('DELETE')

    <button class="text-indigo-600 text-xs">{{ \_\_('Delete')}}</button>

    </form>

    @endcan

### Política de acceso estándar

Creamos un archivo de política con el nombre PostPolicy escribiendo en el terminal el comando: php artisan make:policy PostPolicy

Se crea el archivo app\Policies\PostPolicy.php

En este archivo establecemos la política de acceso a delete

Importamos las clases User y Post y escribimos la función delete

<?php

namespace App\Policies;

use App\Models\Post;

use App\Models\User;

use Illuminate\Auth\Access\HandlesAuthorization;

class PostPolicy

{

    use HandlesAuthorization;

   public function delete(User $user, Post $post){

    return $user->id===$post->user\_id;

   }

}

?>

que nos devolverá verdadero si el usuario activo en la sesión es el usuario autor del post , o falso si no lo es.

Ahora debemos registrar esta política en el archivo app\Providers\AuthServiceProvider.php

Importamos la nueva política

use App\Policies\PostPolicy;

y la registramos

 protected $policies = [

      Post::class => PostPolicy::class,

    ];

Ahora ya no es necesario el código que habíamos añadido en la función boot. Lo dejamos comentado.

 public function boot()

    {

        $this->registerPolicies();

       /\* Gate::define('destroy-post',function(User $user, Post $post){

            return $user->id===$post->user\_id;

        });\*/

    }

Por último actualizamos el controlador y las vistas para que se conecten con el nuevo método

En el controlador

 public function destroy(Request $request, Post $post)

    {

        $this->authorize('delete',$post);

        $post->delete();

        return back();

    }

En la vista

 @can('delete', $post)

    <form action="{{ route('posts.destroy', $post->id) }}" method="POST">

    @csrf

    @method('DELETE')

    <button class="text-indigo-600 text-xs">{{ \_\_('Delete')}}</button>

    </form>

    @endcan

Conclusión:

Hemos visto cómo crear políticas de acceso de tres formas:

Directamente en el método que requiere la política

De forma centralizada en el archivo app\Providers\AuthServiceProvider.php que es el proveedor de servicios de autenticación de Laravel. (Al hacerlo así se puede llamar a la política de acceso desde cualquier parte sin repetir el código)

La forma estándar. Aquí se crea un archivo con la política de acceso, se registra en el proveedor de autenticación y ya es posible implementarla en cualquier método del controlador, en las rutas y en las vistas.

## Mejoras

### Filtrado de registros

Vamos a crear un filtro para que aparezcan solo los posts del usuario seleccionado,

Creamos un controlador app\Http\Controllers\UserController.php con el comando:

Php artisan make:controller UserController

Importamos la clase User

use App\Models\User;

Creamos la función show

public function show(User $user){

        return view('users.show', [

            'posts'=>$user-posts()->latest()->paginate()

        ]);

    }

Creamos la ruta correspondiente en el archivo routes\web.php

Importamos el nuevo controlador

use App\Http\Controllers\UserController;

Añadimos la ruta

Route::get('user/{user}',[UserController::class, 'show'])->middleware(['auth', 'verified'])->name('users.show');

Ahora en el archivo resources\views\posts\inc\item.blade.php modificamos el primer elemento <a> quedando así:

<a href="{{route('users.show', $post->user->id)}}" class="text-lg font-semibold">{{$post->user->name}}

    </a>

Ahora solo queda aplicarlo en las vistas.

Creamos el archivo de vista: resources\views\users\show.blade.php

En éste insertamos el contenido de index.blade.php y lo modificamos quedando así:

<x-app-layout>

    <x-slot name="header">

        <h2 class="font-semibold text-xl text-gray-800 leading-tight">

            {{ \_\_('Filter') }}

        </h2>

    </x-slot>

    <div class="py-12">

        <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">

            <div class="bg-white overflow-hidden shadow-sm sm:rounded-lg">

                <div class="p-6 text-gray-900">

                    @include('posts.inc.list')

                </div>

            </div>

        </div>

    </div>

</x-app-layout>

### Módulo de idiomas

Vamos a crear un diccionario para que el idioma de los elementos que aparecen en el navegador lo hagan en español.

Creamos el archivo lang\es.json en el que pondremos la traducción de cada palabra.

{

    "Save": "Guardar",

    "Dashboard" : "Panel administrativo",

    "Posts": "Publicaciones",

    "Body" : "Publicación",

    "User": "Usuario",

    "Log Out": "Cerrar sesión",

    "Profile": "Perfil"

    }

En el archivo config\app.php modificamos ‘locale’ => ‘en’ por ‘locale’=>’es’

 'locale' => 'es',

Y comprobamos que, en el navegador, ahora los elementos se nombran en Español.

Conclusión:

Hemos visto cómo se pueden ir añadiendo mejoras al proyecto como un filtro de datos. Para hacerlo usamos el mismo sistema controlador=>ruta=>vistas creando un método en el controlador, usa ruta asociada a dicho método y una vista para mostrar en el navegador.

Una vez comprendidas las bases del funcionamiento del framework, se requiere muy poco código para ir haciendo las ampliaciones y mejoras que se deseen.

También hemos visto cómo cambiar el idioma de los mensajes que salen en el navegador simplemente creando un diccionario e informando al sistema que lo use.

## Añadir datos fake

php artisan make:Factory Post genera el archivo database\factories\PostFactory.php

<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

use App\Models\Post;

/\*\*

 \* @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Post>

 \*/

class PostFactory extends Factory

{

    /\*\*

     \* Define the model's default state.

     \*

     \* @return array<string, mixed>

     \*/

    protected $model=Post::class;

    public function definition()

    {

        return [

            'body'=>$this->faker->sentence(45)

        ];

    }

}

php artisan make:seeder PostSeeder genera el archivo database\seeders\PostSeeder.php

<?php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;

use Illuminate\Database\Seeder;

use App\Models\Post;

class PostSeeder extends Seeder

{

    /\*\*

     \* Run the database seeds.

     \*

     \* @return void

     \*/

    public function run()

    {

        Post::factory()->count(50)->create();

    }

}

### Crear el Archivo PostFactory

Crea un archivo factory que contenga tu función de definición. Para esta demostración, crearás un factory utilizando Artisan:

php artisan make:factory Post

Este código crea un archivo database\factories\PostFactory.php.

La función definition dentro de PostFactory define cómo Faker genera datos falsos.

En nuestro caso queda así

<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

use App\Models\Post;

use App\Models\User;

/\*\*

 \* @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Post>

 \*/

class PostFactory extends Factory

{

    /\*\*

     \* Define the model's default state.

     \*

     \* @return array<string, mixed>

     \*/

    protected $model=Post::class;

    public function definition()

    {

        return [

            'body'=>$this->faker->sentence(45),

            'user\_id'=>$this->faker->numberBetween(20,User::count())

        ];

    }

}

Lo primero es añadir la importación de la clase Post

use App\Models\Post;

Y en la función definition

Este código le dice a Faker que genere lo siguiente:

Un body con hasta 45 frases

Un user\_id que corresponda a usuarios existentes en este caso entre el 20 y el total que hayan registrados

Para conectar la clase Post a PostFactory declaramos una variable protegida $model antes de la definición:

protected $model = Post::class;

### Cómo Sembrar la Base de Datos

En programación, sembrar (seeder) significa generar datos aleatorios falsos para una base de datos con fines de prueba.

### Archivo PostSeeder

Creamos un archivo seeder que utilice factory para generar los datos con el comando:

php artisan make:seeder PostSeeder.php

y se crea un archivo database\seeders\PostSeeder.php

Edita la Función Run

Conecta la clase Post al PostSeeder.

En la función run añadimos

Post::factory()->count(50)->create();

Este código indica al PostSeeder que utilice la clase Post y la función de definición de PostFactory para generar 50 comentarios dentro de la base de datos.

Añadimos también el use App\Models\Post y queda así

<?php

namespace Database\Seeders;

use Illuminate\Database\Console\Seeds\WithoutModelEvents;

use Illuminate\Database\Seeder;

use App\Models\Post;

class PostSeeder extends Seeder

{

    /\*\*

     \* Run the database seeds.

     \*

     \* @return void

     \*/

    public function run()

    {

        Post::factory()->count(50)->create();

    }

}

Nombres españoles

Se puede configurar Faker para que cree datos locales generando nombres españoles en lugar de nombres aleatorios .

Para ello establecemos faker\_locale dentro del archivo app/config.php en es\_ES.

 \*/

    'faker\_locale' => 'es\_ES',

    /\*

### Archivo UserSeeder

El archivo UserSeeder ya existe en el sistema junto a su Factory correspondiente por lo que podemos pasar a inertar datos falsos.

### Insertando datos fake

Primero los usuarios

php artisan db:seed --class=UserSeeder

y luego los posts

php artisan db:seed --class=PostSeeder

y comprobamos en el navegador que se han añadido 50 posts. Mirando en la base de datos también se han añadido 50 usuarios con nombre españoles.

### Restablecer la Base de Datos

Si se desea restablecer la base de datos eliminando todos los registros de todas las tablas utiliza RefreshDatabase dentro del archivo de prueba tests\Feature\ExampleTest.php.

En este archivo quitamos el comentario en

use Illuminate\Foundation\Testing\RefreshDatabase;

y añadimos la siguiente línea de código encima de la función test\_the\_application\_returns\_a\_successful\_response:

Debe quedar así:

<?php

namespace Tests\Feature;

use Illuminate\Foundation\Testing\RefreshDatabase;

use Tests\TestCase;

class ExampleTest extends TestCase

{

    /\*\*

     \* A basic test example.

     \*

     \* @return void

     \*/

    use RefreshDatabase;

    public function test\_the\_application\_returns\_a\_successful\_response()

    {

        $response = $this->get('/');

        $response->assertStatus(200);

    }

}

### Test propio de Laravel

php artisan test

Este código ejecuta todas las pruebas de la aplicación y actualiza la base de datos después de las pruebas.

## Resumen final

Hemos visto la forma de trabajar con la base de datos en esta framework creando una base de datos, un controlador, tablas y vistas para interactuar con el usuario a través del navegador.

Creamos una base de datos vacía con el nombre de “publicaciones” con nuestro cliente de mysql que es PhpmyAdmin. Establecemos en el fichero .env las variables de entorno de la base de datos(nombre, usuario, password).

Ejecutamos el comando Php aristan migrate

Y se crean las tablas que se definen en los archivos de la carpeta migrations.

php artisan make:model Post -m (este comando crea la clase Post y, al añadir el parámetro –m crea también el archivo de migraciones database\migrations\2024\_01\_04\_103911\_create\_posts\_table.php)

Con esto estamos creando el esquema para la creación de la tabla “posts”

Añadimos las instrucciones para la creación de un campo user\_id en la tabla posts y establecemos una clave ajena en el campo user\_id que se relaciona con el campo “id” de la tabla “users”.

$table->unsignedBigInteger('user\_id');

            $table->foreign('user\_id')->references('id')->on('users');

            $table->text('body');

Ejecutamos de nuevo php artisan migrate y se crea la tabla en la base de datos.

Relaciones:

Establecemos la multiplicidad entre las clases Post y User ya que un usuario pude tener muchos posts pero cada post sólo puede pertenecer a un usuario modificando las clases User y Post de la siguiente manera:

En la clase User Añadimos la función posts()

public function posts()

    {

        return $this->hasMany(Post::class);

    }

En la clase post la función user

public function user()

    {

        return $this->belongsTo(User::class);

    }

Y añadimos en Post

Consultas

Las consultas las realizamos desde el archivo controlador app\Http\Controllers\PostController.php. En éste se crean las funciones index,store y destroy.

Al ser ua metodología orientada a objetos entiendo que las consultas propiamente dichas se hacen en las clases de las que se hereda. Siguiendo hacia arriba la herencia de clases creo que las consultas vienen de:

vendor\laravel\framework\src\Illuminate\Database\Connection.php donde se crean multitud de mátodos para las consultas y la conexión.

Ante la pregunta de si hay consultas creadas por defecto yo diría que todas las consultas se crean por defecto y lo único que hemos hecho es llamarlas desde el controlador.

Hemos visto cómo funciona el sistema controlador=>ruta=>vista que es la base sobre la que funciona todo. El controlador es el que genera la consulta, la ruta especifica el tipo (get, post, delete, etc..) y lo asocia a un método del controlador y le asigna un nombre de ruta. Las vistas se construyen llamando a las rutas.

Hemos visto como Laravel proporciona sistemas de seguridad propios (Middleware Auth y @csrf ). Hemos visto cómo crear y usar componentes para aumentar el feedback con el usuario. Hemos visto herramientas de optimización como el debug-bar de barryvdh.

Hemos visto cómo crear políticas de acceso de tres formas: directa, centralizada y estándar.

Hemos visto cómo crear y aplicar un módulo de idiomas

Hemos visto cómo se pueden ir añadiendo mejoras al proyecto con el sistema controlador=>ruta=>vistas creando un método en el controlador, usa ruta asociada a dicho método y una vista para mostrar en el navegador.

Por último hemos utilizado herramientas propias de Laravel para crear datos falsos y probar la aplicación.

## Conclusiones finales

Este laboratorio con Laravel me ha permitido introducirme en el concepto de un framework. Realmente me parece un punto de partida excelente para desarrollar proyectos con bases de datos ya que proporciona muchísimas herramientas y métodos predefinidos , además de generar automáticamente los archivos de modelos, creación de tablas, políticas de acceso, etc. Proporciona un entorno visual actual y agradable que, por supuesto se puede modificar.

Todo ello permite desarrollar aplicaciones con un resultado profesional partiendo de una base bien estructurada. Una vez creada la base de datos en blanco, todo lo demás se puede hacer desde el framework.

El ciclo controlador => ruta => vista es la base sobre la que se fundamenta todo el proyecto.

El controlador es el que genera la consulta, la ruta especifica el tipo (get, post, delete, etc..) y lo asocia a un método del controlador y le asigna un nombre de ruta. Las vistas se construyen llamando a las rutas.

Métodos como Middleware Auth o funciones como @csrf facilitan mucho la tarea del desarrollador en cuestiones de seguridad.

Una vez comprendidas las bases del funcionamiento del framework, se requiere muy poco código para ir construyendo el proyecto y hacer las ampliaciones y mejoras que se deseen.

Es importante también recordar la filosofía de utilizar archivos pequeños con un código limpio que se puedan interpretar fácilmente. También es muy importante buscar siempre la máxima eficiencia en la ejecución de los programas, a fin de obtener los resultados deseados con el mínimo consumo de recursos.

El uso de herramientas como los factories y los seeders ayuda mucho a la hora de verificar el funcionamiento de la aplicación al poder introducir una gran cantidad de datos de una forma rápida y sencilla.