## Introducción

Siguiendo el vídeo no puede realizar la instalación

Lo consigo instalando Composer y después ejecutando el comando

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

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.

## Datos

php artisan make:model Post

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.

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.

 $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.

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 la clase User de la siguiente manera:

Añadimos la función posts()

public function posts()

    {

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

    }

En la clase post

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.

# 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 ls 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>

## 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 clasesUser 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

## 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.

### 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

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.

### Conclusiones

## Añadir datos fake