



Tutorial OpenWebinars

Asignatura: Desarrollo Web en Entorno Servidor
Autor: Judit García Canós

Índice

1. Introducción.....	3
1.1. Configuración inicial en Laravel:.....	3
1.2. Base de datos inicial:.....	5
2. Datos:.....	7
2.1. Tabla Posts:.....	7
2.2. Relación de tablas:.....	8
3. Desarrollo:.....	9
3.1. Ruta y controlador:.....	9
3.2. Estructura de un formulario:.....	10
3.3. Diseño de un formulario:.....	12
3.4. Middleware Auth:.....	13
3.5. Guardar en base de datos:.....	15
3.6. Componentes Blade:.....	18
3.7. Listado de registros:.....	19
3.8. Optimización de consulta:.....	21
3.9. Eliminar de la base de datos:.....	22
4. Políticas de acceso:.....	24
4.1. Política de acceso básico:.....	24
4.2. Política de acceso centralizada:.....	25
4.3. Política de acceso estándar:.....	26
5. Mejoras:.....	27
5.1. Filtrado de registros:.....	27
5.2. Módulo de idiomas:.....	29
6. Incluir datos “fake”:.....	30
7. Conclusiones:.....	32

1. Introducción

1.1. Configuración inicial en Laravel:

A lo largo de este tutorial aprenderemos a desarrollar aplicaciones web integrales mediante el uso del framework Laravel. Este recurso nos guiará paso a paso a través del proceso de desarrollo, desde la instalación inicial hasta la comprensión de conceptos fundamentales como rutas, controladores y modelos.

Para que todo funcione correctamente se ha tenido que instalar previamente diferentes herramientas:

- **NodeJs:** en Laravel se emplea principalmente para facilitar el desarrollo frontend y gestionar las dependencias de JavaScript, asegurando una integración eficiente de las tecnologías del lado del cliente con el framework del lado del servidor.
- **Composer:** se utiliza para Gestión de Dependencias de PHP Laravel. Al igual que muchas otras aplicaciones de PHP, utiliza bibliotecas y componentes de terceros para tareas específicas. Composer facilita la instalación, actualización y carga automática de estas dependencias, permitiendo que el desarrollador se centre en la lógica de la aplicación en lugar de gestionar manualmente cada paquete. Además se utiliza para Autocarga de clases, creación de proyectos laravel y administración de versiones.

Para la instalación de Laravel se ha lanzado desde composer: **composer global require laravel/installer**

Laravel es un popular framework de desarrollo web de código abierto, escrito en PHP. Laravel sigue el patrón de diseño de arquitectura Modelo-Vista-Controlador (MVC), que proporciona una estructura organizada para el desarrollo de aplicaciones web.

Las versiones instaladas han sido las siguientes:

```
PS C:\workspace\fp\2\entornos servidor web\laravel> composer --version
Composer version 2.6.6 2023-12-08 18:32:26
PS C:\workspace\fp\2\entornos servidor web\laravel> laravel --version
Laravel Installer 5.2.0
PS C:\workspace\fp\2\entornos servidor web\laravel> █
```

Para la creación del proyecto de cero, se han ido siguiendo los pasos del tutorial del video de “Configuración inicial de Laravel”.

Primero se crea el proyecto con la siguiente instrucción: “**laravel new publicaciones**”

Una vez se ha terminado de ejecutar, instalando la aplicación en xampp dio problemas y no se visualizaba nada. Para ello tuve que investigar y encontré un artículo en el que hacía mención a una parte del manual de xampp <http://localhost/dashboard/howto.html> y dentro de la documentación oficial de xampp a este artículo en concreto <http://localhost/dashboard/docs/configure-vhosts.html>

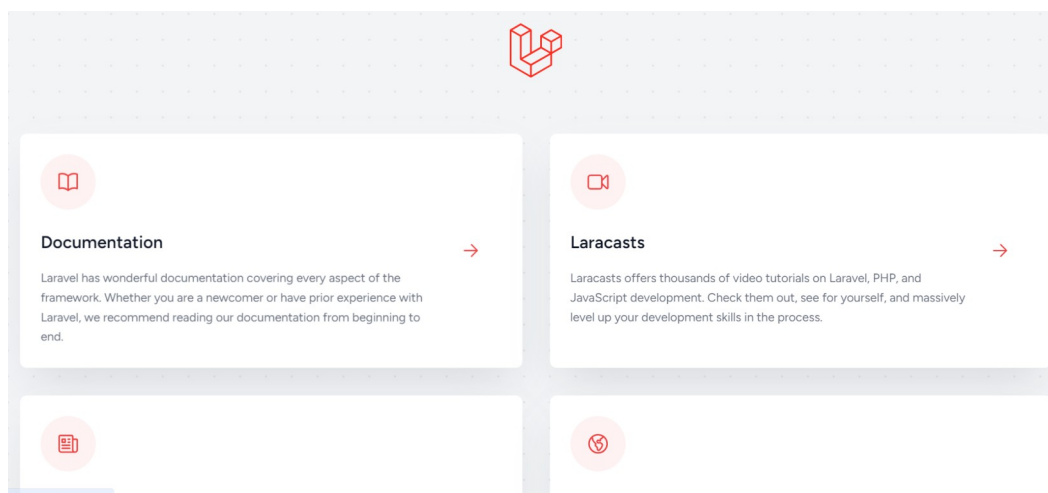
Para poder realiza la configuración y el acceso desde xampp hay que modificar el archivo “httpd-vhosts.conf”

```
<VirtualHost *:80>
    DocumentRoot "C:\xampp\htdocs\publicaciones\public"
    ServerName publicaciones.test
</VirtualHost>
```

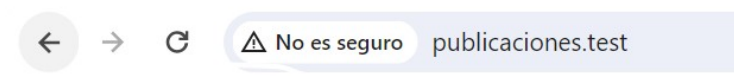
Con esto se configura la ruta principal donde se encuentra el “index” de nuestra aplicación de Laravel siempre y cuando se acceda través de nombre otorgado en el serverName en este caso “publicaciones.test”. Para que desde el navegador no redirija esta url y la busque por internet se modifica el archivo host para que cuando se ponga en el navegador sólo busque dentro de los servidores locales. Esto se hace mediante la modificación de el archivo host de windows.

```
127.0.0.1    publicaciones.test
```

Una vez con la configuración realizada, se puede visualizar el proyecto inicial



Siguiendo el paso a paso del video se modifica la vista “welcome.blade.php”, eliminando los estilos, el código sobrante y añadiendo el h1, con lo que el resultado sería el siguiente:



Welcome

Laravel – Sistemas de opiniones en Laravel 9

Una vez realizado esto, se va a instalar el modulo de autenticación para ello se lanza lo siguiente:

composer require laravel/breeze --dev (Descarga del modulo)

```
PS C:\xampp\htdocs\publicaciones> composer require laravel/breeze --dev
./composer.json has been updated
Running composer update laravel/breeze
Loading composer repositories with package information
Updating dependencies
Lock file operations: 1 install, 0 updates, 0 removals
```

php artisan breeze:install (Instalación del modulo en un proyecto existente)
(seleccionar la opcion blade) Seleccione la acción api anteriormente y me rompió el proyecto y tuve que empezar de 0.

```
PS C:\xampp\htdocs\publicaciones> php artisan breeze:install

Which Breeze stack would you like to install?
Blade with Alpine ..... blade
Livewire (Volt Class API) with Alpine ..... livewire
Livewire (Volt Functional API) with Alpine ..... livewire-functional
React with Inertia ..... react
Vue with Inertia ..... vue
API only ..... api
> blade
```

npm install (Instalación del modulo de autenticación)

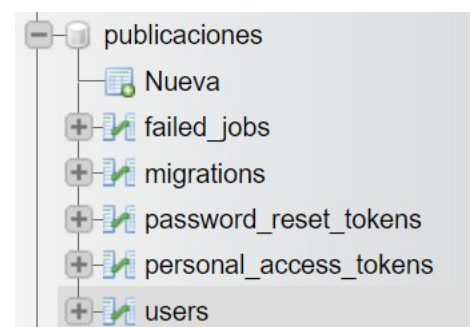
1.2. Base de datos inicial:

Una vez instalado el modulo de autenticación, se procede a conectar con Base de datos y crear el esquema de la aplicación publicaciones y ejecutar la instrucción “php artisan migrate” que realiza la importación del esquema.

Pantallazo de .env

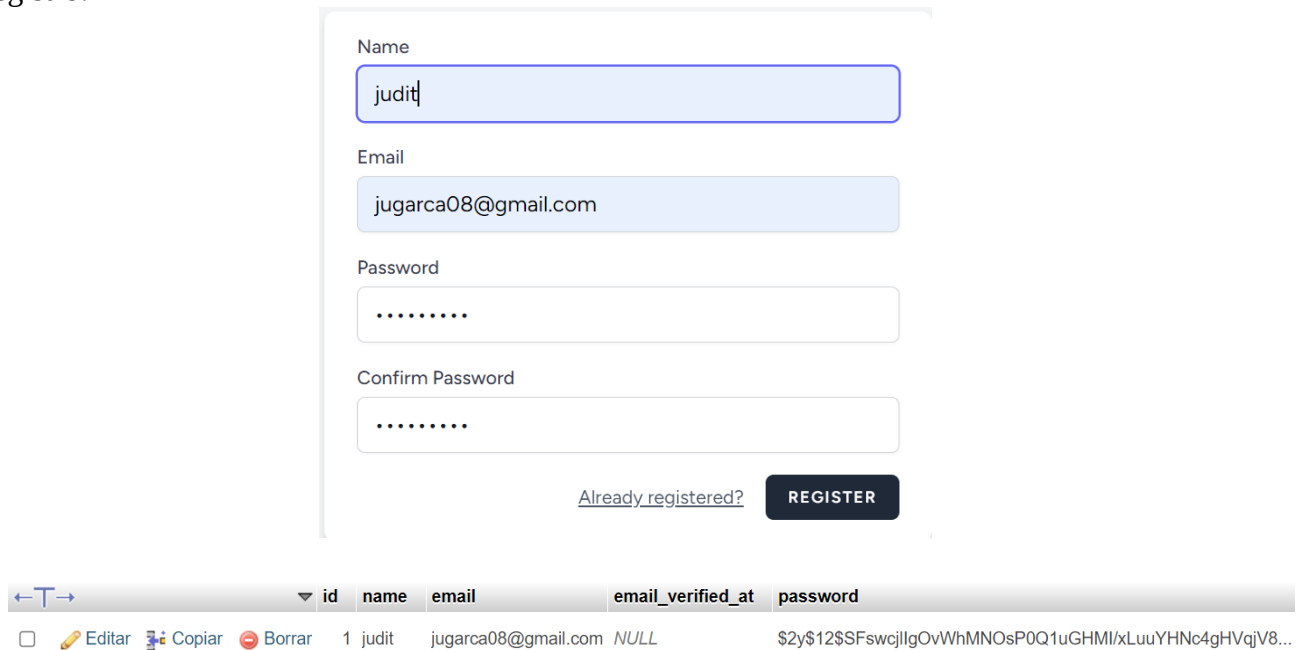
```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=publicaciones
DB_USERNAME=root
DB_PASSWORD=
```

Pantallazo de el esquema creado desde **phpmyadmin** (mysql)



Laravel – Sistemas de opiniones en Laravel 9

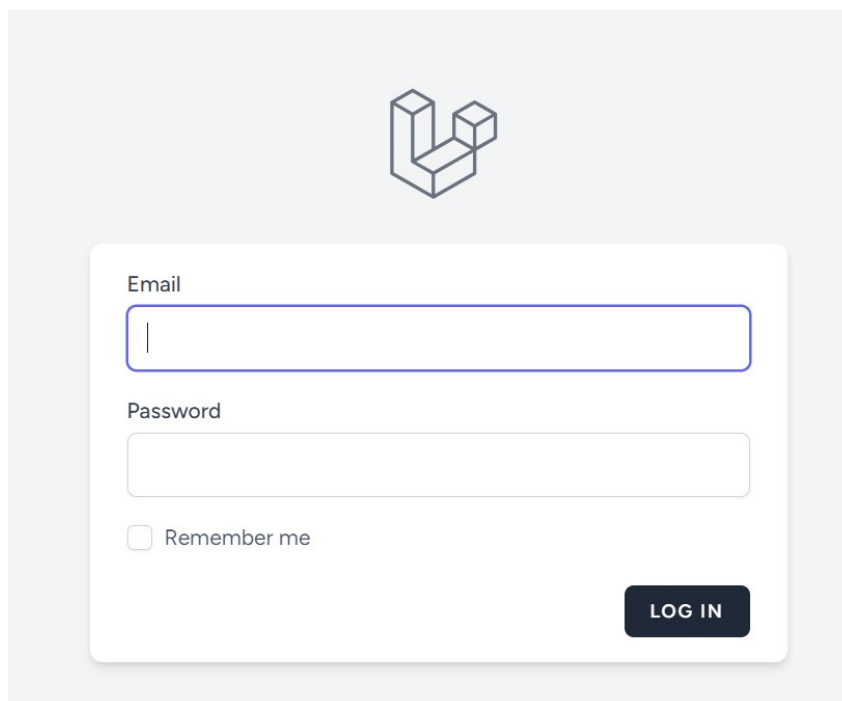
Damos de alta a un usuario y comprobamos que se ha introducido en la tabla users como nuevo registro:



The image shows a registration form and a database table view. The form has fields for Name, Email, Password, and Confirm Password. The Name field contains 'judit', the Email field contains 'jugarca08@gmail.com', and the Password and Confirm Password fields are masked with dots. There is a 'REGISTER' button and a link 'Already registered?'. Below the form is a table view showing the user's data in the 'users' table.

id	name	email	email_verified_at	password
1	judit	jugarca08@gmail.com	NULL	\$2y\$12\$SFswcjlIgOvWhMNOsP0Q1uGHMI/xLuuYHNc4gHVqjV8...

Una vez creado el usuario se puede entrar en la aplicación desde la pantalla de login:



The image shows a login form with fields for Email and Password. There is a 'Remember me' checkbox and a 'LOG IN' button. The form is centered on a light gray background with a 3D cube icon above it.

2. Datos:

2.1. Tabla Posts:

Para crear la entidad Post y que además se haga la migración(-m) escribimos el siguiente comando en el terminal: **php artisan make:model Post -m**

```
PS C:\xampp\htdocs\publicaciones> php artisan make:model Post -m

INFO Model [C:\xampp\htdocs\publicaciones\app\Models\Post.php] created successfully.

INFO Migration [C:\xampp\htdocs\publicaciones\database\migrations\2024_01_09_143635_create_posts_table.php]
created successfully.

PS C:\xampp\htdocs\publicaciones>
```

Añadimos a dicha tabla:

- Una columna 'user_id' que es una foreign key , que hace referencia a la columna 'id' de la tabla 'users'. Esto hace que únicamente los usuarios registrados, con un 'id' propio en la tabla 'users' puedan realizar publicaciones.
- Una columna llamada 'body' en la que podremos introducir texto.

```
public function up(): void
{
    Schema::create('posts', function (Blueprint $table) {
        $table->id();

        $table->unsignedBigInteger('user_id');
        $table->foreign('user_id')->references('id')->on('users');

        $table->text('body');

        $table->timestamps();
    });
}
```

- Se realiza la migración con la siguiente instrucción y visualizamos el resultado: **php artisan migrate**

```
PS C:\xampp\htdocs\publicaciones> php artisan migrate

INFO Running migrations.

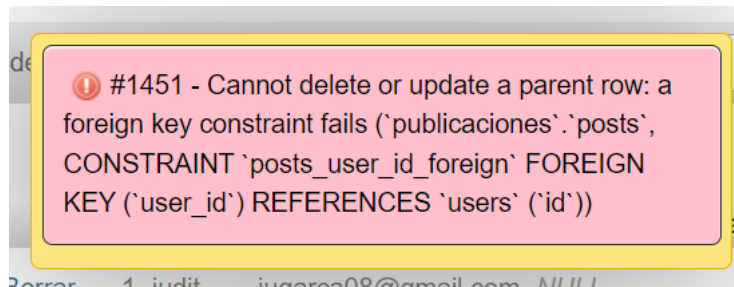
2024_01_09_143635_create_posts_table ..... 107ms DONE
```

```
SELECT * FROM `posts`
```

id	user_id	body	created_at	updated_at
----	---------	------	------------	------------

- Se crea una publicación para el usuario existente (con 'id' 1) e intentamos eliminar dicho usuario.

	id	user_id	body	created_at	updated_at
<input type="checkbox"/> Editar <input type="button" value="Copiar"/> <input type="button" value="Borrar"/>	1	1	publicación	NULL	NULL



Observamos cómo no nos deja borrar el usuario, ya que éste tiene una publicación.

Para solucionar este punto deberemos hacer una eliminación en cascada, arrastrando también todas las publicaciones del usuario eliminado. Para ello llevaremos a cabo los siguientes pasos:

- Se ejecuta el siguiente comando: **php artisan migrate:refresh** para borrarlo absolutamente todo y volverlo a crear.
- Se debe indicar que el sistema de eliminación sea en cascada:

```
Schema::create('posts', function (Blueprint $table) {
    $table->id();

    $table->unsignedBigInteger('user_id');
    $table->foreign('user_id')->references('id')->on('users')->onDelete('cascade');

    $table->text('body');

    $table->timestamps();
});
```

- Volvemos a lanzar el comando para lanzar los archivos de nuevo: **php artisan migrate:refresh**
- Ahora ya podemos eliminar el usuario en nuestra base de datos y eliminaremos con él todas las publicaciones que haya realizado.

2.2. Relación de tablas:

Para crear la relación entre tablas hay que modificar las clases User y Post. En la clase User añadiremos la relación con post para definir que un usuario puede tener muchas publicaciones. Lo definimos con el siguiente código:

```
/* Relacion entre la tabla de usuarios y la tabla de post
| donde indica que un usuario tiene muchas publicaciones*/
public function posts()
{
    return $this->hasMany(Post::class);
}
```


En la clase post se definirá que una publicación sólo puede tener asignado un usuario.

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

Además, en la clase post, queda por definir las columnas en las que podemos realizar transacciones sobre la base de datos:

```
/**
 * The attributes that are mass assignable.
 *
 * @var array<int, string>
 */
protected $fillable = ['body'];
```

3. Desarrollo:

3.1. Ruta y controlador:

Creamos un controlador llamado PostController:

```
PS C:\xampp\htdocs\publicaciones> php artisan make:controller PostController

INFO Controller [C:\xampp\htdocs\publicaciones\app\Http\Controllers\PostController.php] created successfully.

PS C:\xampp\htdocs\publicaciones>
```

Modificamos la clase del controlador para definir los métodos necesarios para que nos permita mostrar, guardar y borrar información:

```
class PostController extends Controller
{
    public function index(){
        return 'index';
    }

    public function store(){
        //guardar
    }

    public function destroy(){
        //eliminar
    }
}
```

Una vez tenemos el sistema de control, debemos hacer el registro de las rutas adecuadas. Para ello vamos a nuestro archivo `web.php` y hacemos las siguientes modificaciones:

- Importación del controlador:

```
use App\Http\Controllers\PostController;
```

- Registrar mis rutas. Cada una de las siguientes rutas hace referencia a cada uno de los métodos creados en el punto anterior:

```
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.destroy');
```

- Hacemos una modificación en las vistas para comprobar que se visualiza correctamente:

```
<x-nav-link :href="route('posts.index')" :active="request()->routeIs('posts.index')">
|   {{ __('Post') }}
</x-nav-link>
```

```
<x-responsive-nav-link :href="route('posts.index')">
|   {{ __('Posts') }}
</x-responsive-nav-link>
```

3.2. Estructura de un formulario:

En este punto del tutorial vamos a crear una estructura básica de formulario que utilizaremos para que nuestro controlador retorne no un simple texto, como hacía hasta ahora, sino una vista real.

Para ello modificaremos nuestro controlador 'PostController', diciéndole que retorne una vista llamada 'index' que se encuentra dentro de una carpeta 'posts':

```
public function index(){
|   return view('posts.index');
| }
}
```

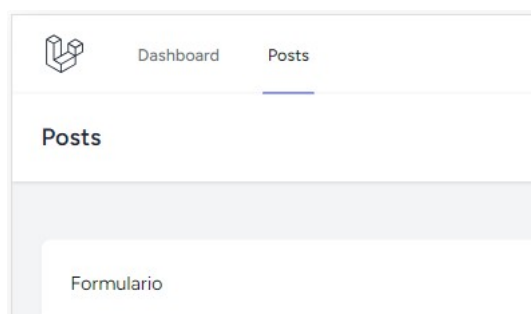
Para que esto funcione es necesario crear dicha vista. Por ello en '**views**' creamos una nueva carpeta llamada 'posts' y dentro de ella un nuevo archivo llamado '**index.blade.php**', donde vamos a crear nuestro formulario.

```
<x-app-layout>
  <x-slot name="header">
    <h2 class="font-semibold text-xl text-gray-800 dark:text-gray-200 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 dark:bg-gray-800 overflow-hidden shadow-sm sm:rounded-lg">
        <div class="p-6 text-gray-900 dark:text-gray-100">
          Formulario
        </div>
      </div>
    </div>
  </div>
</x-app-layout>
```

```
<x-app-layout>
  <x-slot name="header">
    <h2 class="font-semibold text-xl text-gray-800 dark:text-gray-200 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 dark:bg-gray-800 overflow-hidden shadow-sm sm:rounded-lg">
        <div class="p-6 text-gray-900 dark:text-gray-100">
          <form action="{{route('posts.store')}}" method="post">
            @csrf
            Formulario
          </form>
        </div>
      </div>
    </div>
  </div>
</x-app-layout>
```



```
<form action="http://publicaciones.test/posts" method="post"> == $0
  <input type="hidden" name="_token" value="INRlqrXNEU4iX731r6s76cutbeQD0AJJESf:
  YBx" autocomplete="off">
  " Formulario "
</form>
```

3.3. Diseño de un formulario:

Creamos un nuevo componente llamado ‘textarea’: **textarea.blade.php** y a partir de ahí le crearemos la diferentes configuraciones.

```
<textarea
  rows="2"
  {!!
    $attributes->merge([
      'class' => 'border-gray-300 dark:border-gray-700 dark:bg-gray-900 dark:text-gray-300 focus:border-indigo-500 dark:fo
    ])
  !!}
></textarea>
```

Ahora ya podemos crear nuestro formulario haciendo uso de este componente. Para ello nos vamos a nuestra vista ‘index’ y comenzamos a hacer las modificaciones correspondientes para crear una etiqueta, un área de texto y un botón de guardado (‘Save’).

```
<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>
  <div class="flex justify-end mt-4">
    <x-primary-button>
      {{ __('Save')}}
    </x-primary-button>
  </div>
</form>
```

Body

publicación

SAVE

En este punto, debemos comprobar que al pulsar el botón ‘Save’ se realiza el guardado correctamente. Para ello vamos a nuestro controlador ‘PostController’ y modificamos el método store() añadiéndole el request, que es la petición enviada, para poder mostrar la información enviada (en este punto aún no se guarda).

```
public function store(Request $request){
    return $request->all();
}
```

← → ↻ ⚠ No es seguro publicaciones.test/posts

```
{"_token":"6Em1PGR2Kdm12bhuHpaetKs0L6bLYh22esIVkya5","body":"Publicaci\u00f3n1"}
```

3.4. Middleware Auth:

En este punto del tutorial, vamos a realizar una mejora de nuestro código, eliminando aquello que no estamos utilizando, optimizando así la manera en que está funcionando nuestro sistema, y añadiremos también seguridad a las publicaciones, para que nadie pueda acceder a estas rutas sin haber iniciado sesión.

```
<?php

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');
});
```

Queremos también eliminar nuestra vista welcome.blade.php. Para ello hemos de hacer las modificaciones necesarias en web.php para que al hacer esto no falle nuestro sistema. Ahora cuando vayamos a la raíz de nuestro proyecto no queremos que retorne a welcome, que no existe, sino que nos redirija a dashboard. (**dashboard requiere de inicio de sesión**)

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

Cuando se intenta acceder al dashboard esta página utiliza el user y un objeto Auth que se inicializa en el inicio de sesión, para intentar mostrar el nombre de un usuario. Esto nos falla porque esta ruta no está protegida.

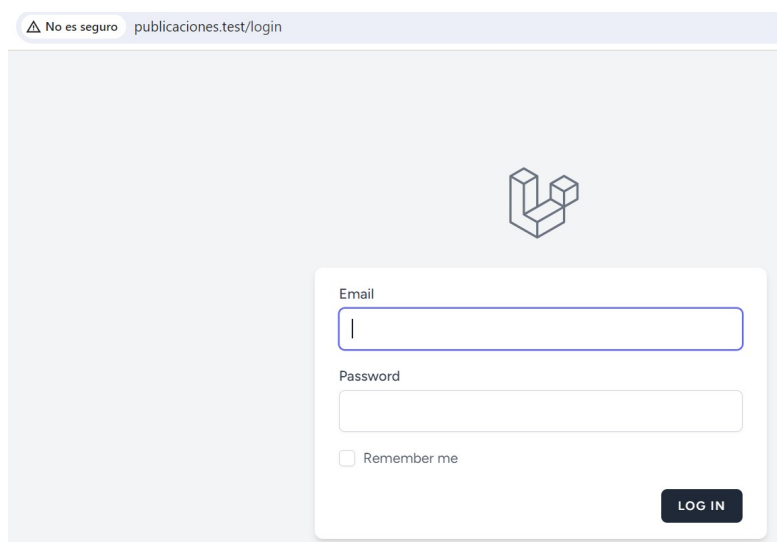
Attempt to read property "name" on null

```
<div>{{ Auth::user()->name }}</div>
```

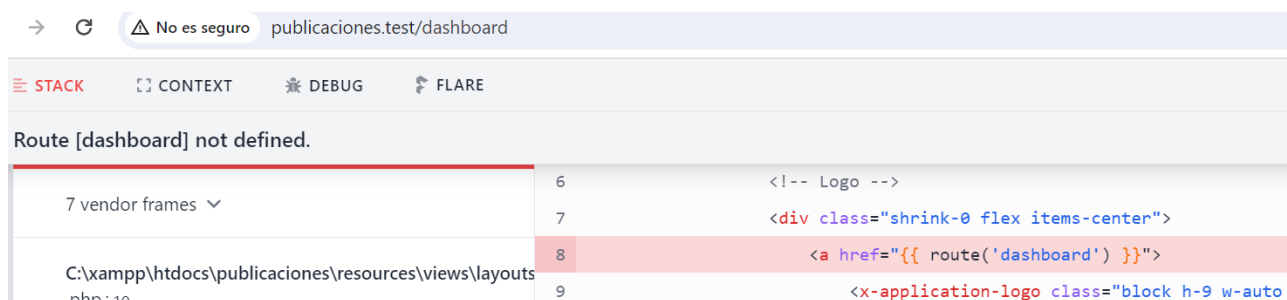
Laravel – Sistemas de opiniones en Laravel 9

Para resolver este problema hemos de proteger nuestras rutas (index, store, destroy) y dashboard con un middleware de autenticación. De este modo, si intentamos acceder directamente a <http://publicaciones.test/posts>, el sistema detecta que no me he logueado y me retorna directamente a la vista de login.

```
Route::get('posts',[PostController::class, 'index'])->middleware(['auth', 'verified'])->name('posts.index');
Route::post('posts',[PostController::class, 'store'])->middleware(['auth', 'verified'])->name('posts.store');
Route::delete('posts/{post}',[PostController::class, 'destroy'])->middleware(['auth', 'verified'])->name('posts.destroy');
```



Haremos otra modificación, puesto queremos que en dashboard sea directamente nuestra lista de publicaciones. Para ello vamos a web.php e indicamos que el posts.index es directamente el dashboard.



Para que no falle, al no encontrar la ruta dashboard; puesto que la hemos eliminado, podemos hacer que el index se renombre a dashboard. A todo lo que haga referencia a dashboard va a significar que quiero mostrar el index de mis publicaciones.

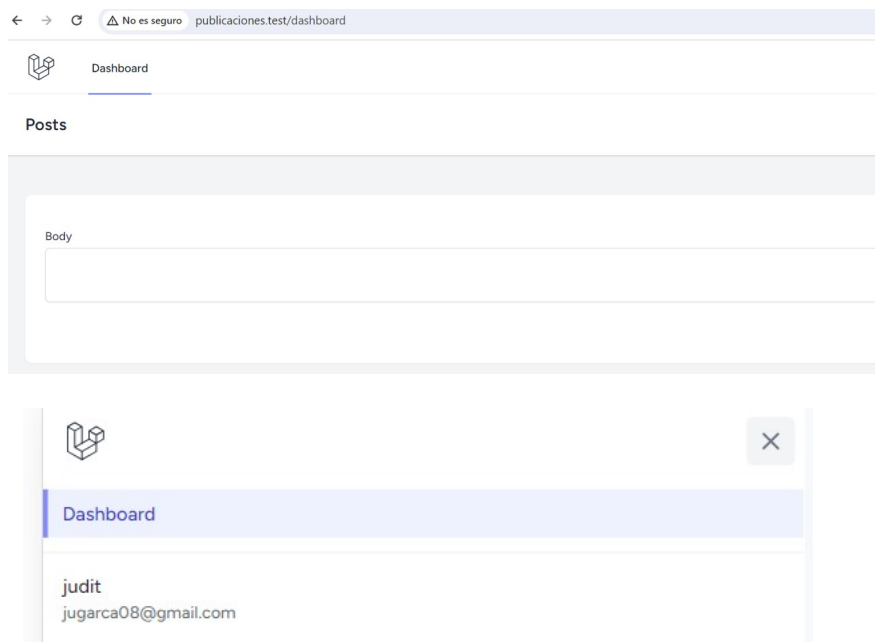
```
Route::get('dashboard',[PostController::class, 'index'])->middleware(['auth', 'verified'])->name('dashboard');
```

Ahora esa ruta ya existe pero me va a fallar el posts.index. Lo podemos solucionar en la plantilla de navegación eliminando.

```
<x-nav-link :href="route('posts.index')" :active="request()->routeIs('posts.index')">
    {{ __('Posts') }}
</x-nav-link>
```

```
<x-responsive-nav-link :href="route('posts.index')">
    {{ __('Posts') }}
</x-responsive-nav-link>
```

En este momento, cuando pulsamos el botón Dashboard, éste representa al índice de mi sistema



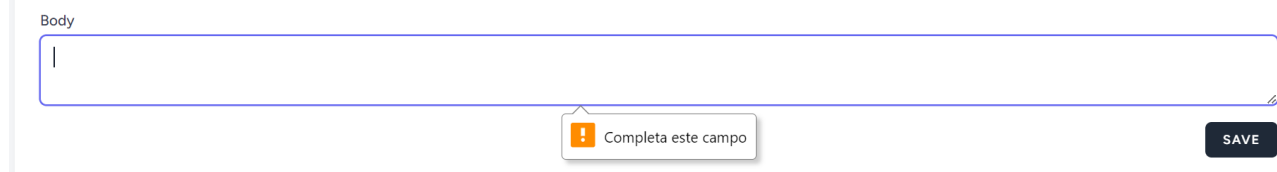
3.5. Guardar en base de datos:

El objetivo de este punto del tutorial es aprender a guardar nuestros datos en una tabla. Para ello construiremos paso a paso las validaciones tanto del lado cliente como del servidor.

Lo primero que hacemos es realizar la validación en nuestro controlador 'PostController' (validación del lado del servidor) para que el campo body sea requerido. De este modo indicamos que la publicación es obligatoria, por lo tanto, si está vacía, no se insertará en base de datos:

```
public function store(Request $request){
    //Validar la petición que llega
    //Validar que el body es requerido
    $request -> validate(['body' => 'required']);
    return $request->all();
}
```

En este momento, queremos poder visualizar el mensaje de error del servidor. Para lograrlo, lo primero que hemos de hacer es desactivar la validación del lado del cliente, ya que de lo contrario, este sería el mensaje que se mostraría:



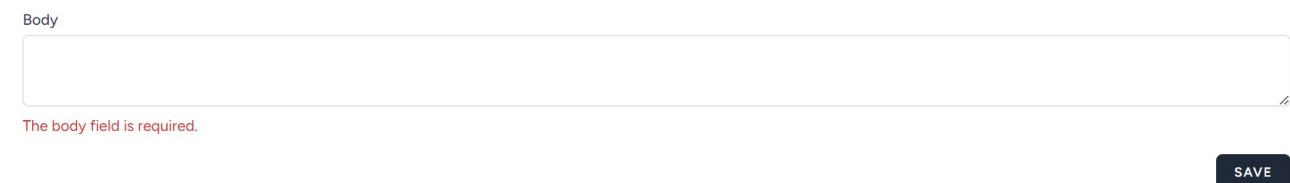
A screenshot of a web form. At the top, the label 'Body' is visible. Below it is a large, empty text area. A red error message 'Completa este campo' (Complete this field) is displayed below the text area. To the right of the text area is a dark blue button labeled 'SAVE'.

Para lograrlo, accedemos a nuestro archivo `index.blade.php` y eliminamos la directiva `'required'` del textarea. Posteriormente, en el vídeo se proporcionan instrucciones sobre cómo mostrar los errores utilizando del componente `'auth-validation-errors'`.

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

Este es un componente que no aparece en mi caso, por lo tanto, he revisado el código autogenerated y he optado por una alternativa de mostrarlo. Consiste en añadir una etiqueta `'x-input-error'` y obtener, del array de errores `$errors`, el mensaje correspondiente al campo solicitado, en este caso, `'body'`.

```
<x-textarea class="block mt-1 w-full" name="body" />
<x-input-error class="mt-2" :messages="$errors->get('body')" />
```



A screenshot of a web form. At the top, the label 'Body' is visible. Below it is a large, empty text area. A red error message 'The body field is required.' is displayed below the text area. To the right of the text area is a dark blue button labeled 'SAVE'.

Finalmente, reincorporamos el atributo `'required'` al textarea, estableciendo así una validación doble.

Para guardar los datos en nuestro sistema hacemos la siguiente modificación en el método store:

```
public function store(Request $request){  
    //Validar la petición que llega  
    //Validar que el body es requerido  
    $request -> validate(['body' => 'required']);  
    //Realiza el guardado en Base de datos  
    $request->user()->posts()->create($request->only('body'));  
    //Devuelve a la misma página  
    return back()->with('status','publicacion guardada correctamente');  
}
```

```
//Realiza el guardado en Base de datos  
$request->user()->posts()->create($request->only('body'));
```

En esta línea lo que estamos indicando es que creamos una nueva publicación asociada al usuario actual utilizando la relación 'posts' del modelo de usuario y el método create, que extrae el contenido del campo 'body' de la request para utilizarlo como el contenido de la nueva publicación.

```
//Devuelve a la misma página  
return back()->with('status','publicacion guardada correctamente');
```

Una vez que la publicación ha sido guardada con éxito, empleamos return back() para redirigir al usuario de vuelta a la página anterior. Además, mediante el método with(), adjuntamos un mensaje de éxito ('publicación guardada correctamente') que se encuentra disponible en la vista a través de una variable de sesión llamada 'status'.

Para que este mensaje se muestre es necesario que hagamos uso del componente 'auth-session-status' en nuestro index.blade.php. Con ello le estamos diciendo que, si existe un estatus se muestre. Si tenemos un mensaje de sesión llamado estatus se envía y se muestra.

```

<div class="p-6 text-gray-900 dark:text-gray-100">
<x-auth-session-status class="mb-4" :status="session('status')"/>
  <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 />
      <x-input-error class="mt-2" :messages="$errors->get('body')"/>
    </div>
    <div class="flex justify-end mt-4">
      <x-primary-button>
        {{ __('Save') }}
      </x-primary-button>
    </div>
  </form>
</div>

```

publicacion guardada correctamente

Body

SAVE

Y aquí podemos observar nuestras publicaciones guardadas correctamente en nuestra base de datos:

Opciones extra								
		id	user_id	body	created_at	updated_at		
<input type="checkbox"/>	Editar	2	2	publicacion 1	2024-01-11 13:48:01	2024-01-11 13:48:01		
<input type="checkbox"/>	Editar	3	2	publicacion3	2024-01-11 13:53:55	2024-01-11 13:53:55		
<input type="checkbox"/>	Editar	4	2	publicacion5	2024-01-11 13:55:28	2024-01-11 13:55:28		
<input type="checkbox"/>	Editar	5	2	publicacion6	2024-01-11 13:56:11	2024-01-11 13:56:11		
<input type="checkbox"/>	Editar	6	2	2	2024-01-12 11:05:36	2024-01-12 11:05:36		
<input type="checkbox"/>	Editar	7	2	publicación 8	2024-01-12 13:15:00	2024-01-12 13:15:00		

3.6. Componentes Blade:

En este punto del tutorial nos centraremos en renombrar el componente utilizado anteriormente, para un mejor ajuste a su función, dándole un nombre genérico. Para ello deberemos renombrar también en todos los puntos de la aplicación dónde se está utilizando dicho componente, ya que si no hiciésemos ésto fallaría.

Otra opción habría sido crear un componente nuevo para la nueva utilidad que le íbamos a dar, para ello duplicaríamos el componente y le cambiaríamos el nombre.

3.7. Listado de registros:

En este momento, vamos a organizar nuestro código de manera que sea más mantenible y reutilizable. Para lograrlo, llevaremos a cabo la extracción del código de nuestro formulario, incorporándolo en el archivo ‘index’ mediante un ‘include’. De esta manera, en el futuro podrá ser utilizado en 1 o n pantallas.

Con este propósito, creamos una nueva carpeta llamada ‘inc’, la cual contendrá todos los archivos que deseamos incluir. Dentro de esta carpeta, creamos un nuevo archivo llamado ‘form.blade.php’, donde ubicamos tanto el formulario como el estado (‘status’), aislando así la lógica de mi formulario.

```
resources > views > posts > inc > form.blade.php
1  <!--Se modifica el x-auth-session-status que es un componente del auth por uno cuyo nombre no
2  <!--refleje lo del auth.-->
3  <x-session-status class="mb-4" :status="session('status')" />
4  <form action="{{route('posts.store')}}" method="post">
5      @csrf
6      <div class="mt-4">
7          <x-input-label for="body" :value="__('Body')" />
8          <x-textarea class="block mt-1 w-full" name="body" required />
9          <x-input-error class="mt-2" :messages="$errors->get('body')" />
10     </div>
11     <div class="flex justify-end mt-4">
12         <x-primary-button>
13             {{ __('Save') }}
14         </x-primary-button>
15     </div>
16 </form>
```

En el vídeo tutorial, el profesor nos guía a través de la corrección de un error relacionado con la ubicación de la carpeta ‘inc’, que inicialmente estaba fuera de la carpeta ‘posts’:

```
mv resources/views/inc resources/views/posts
```

Simultáneamente, el tutorial aborda la visualización de las diferentes publicaciones bajo del formulario. Para lograr esto, seguimos los siguientes pasos:

- 1 Creamos un nuevo archivo llamado ‘list.blade.php’ en la carpeta ‘inc’, dónde incluimos el código necesario para personalizar la presentación de nuestro listado de publicaciones.
- 2 Realizamos un ‘include’ de este nuevo archivo en nuestro ‘index’.
- 3 Creamos en ‘PostController’ la consulta:

3.1 Importamos la clase ‘Post’ del modelo para poder hacer uso de ella:

```
use App\Models\Post;
```

3.2 Generamos la consulta:

Dentro de la función `index()` creamos la consulta que nos permite obtener las publicaciones más recientes de la base de datos:

- `Post::latest()` nos indica que queremos obtener las publicaciones ordenadas por la fecha de creación, de la más reciente a la más antigua (orden descendente).
- `Paginate()` se utiliza para dividir los resultados en varias páginas, lo que es útil cuando hay muchas publicaciones y queremos mostrar solo un número limitado por página.

```
//      SELECT * FROM POSTS ORDER BY ID DESC; Despues lo pagina el sistema
public function index(){
    return view('posts.index',[
        'posts' => Post::latest()->paginate()
    ]);
}
```

- 4 Finalmente, escribimos el código correspondiente en `'list.blade.php'` para que nuestro listado se muestre con éxito:

```
resources > views > posts > inc > list.blade.php
1  @foreach ($posts as $post)
2  <div class="mt-4">
3      <a href="#" class="text-lg font-semibold">{{$post->user->name}}</a>
4      <p class="met-1 text-xs">
5          <em>
6              {{$post->created_at->format('d/m/Y')}}
7          </em>
8          {{$post->body}}
9      </p>
10 </div>
11 @endforeach
12
13 <div class="mt-4">
14     {{$posts->links()}}
15 </div>
```

Utilizamos un bucle `@foreach` para iterar sobre cada publicación (`$post`) en la variable `$posts`. Para cada una de las publicaciones mostramos:

- el nombre del usuario: `{{$post->user->name}}`
- fecha de creación formateada para que se muestre de la manera deseada ('d/m/Y'): `{{$post->created_at->format('d/m/Y')}}`
- Cuerpo de la publicación: `{{$post->body}}`

Al final del listado, se incluyen enlaces de paginación generados automáticamente por Laravel, permitiendo la navegación entre las páginas si hay más publicaciones de las que se pueden mostrar en una sola página: `{{ $posts->links() }}`

Body

SAVE

judit

12/01/2024 publicación 8

judit

12/01/2024 2

judit

11/01/2024 publicación6

judit

11/01/2024 publicación5

judit

11/01/2024 publicación3

3.8. Optimización de consulta:

Hasta este punto, hemos logrado generar nuestra consulta de manera adecuada. Sin embargo, en el proceso de desarrollo, es crucial enfocarnos en crear crear código de alta calidad, buscando la optimización y mejorando el rendimiento. Esto asegura que no nos enfrentemos a problemas de rendimiento cuando trabajemos con un número elevado de registros.

Para ayudarnos a lograr esto, instalamos un nuevo componente desde el terminal que nos permite visualizar los problemas de rendimiento:

```
PS C:\xampp\htdocs\publicaciones> composer require barryvdh/laravel-debugbar --dev
```

Si revisamos en nuestro navegador, observamos que ahora nos aparece una pequeña barra en la que podemos leer el número de queries generadas (en este caso, 9). Podemos ver que tenemos una consulta generada en el controlador, pero el resto de consultas (consulta del usuario) se están realizando en la vista.

Messages	Timeline	Exceptions	Views 19	Route	Queries 9	Models 13	Gate	Session	Request	GET dashboard	21MB	1.52s	8.2.4	
9 statements were executed, 6 of which were duplicated, 3 unique. Show only duplicated 6.43ms														
select * from `users` where `id` = 2 limit 1						2.81ms	vendor\laravel\framework\src\Illuminate\Auth\EloquentUserProvider.php:59 publicaciones							
select count(*) as aggregate from `posts`						560µs	app\Http\Controllers\PostController.php:15 publicaciones							
select * from `posts` order by `created_at` desc limit 15 offset 0						610µs	app\Http\Controllers\PostController.php:15 publicaciones							
select * from `users` where `users`.`id` = 2 limit 1						430µs	view::posts.inc.list:3 publicaciones							
select * from `users` where `users`.`id` = 2 limit 1						420µs	view::posts.inc.list:3 publicaciones							
select * from `users` where `users`.`id` = 2 limit 1						580µs	view::posts.inc.list:3 publicaciones							
select * from `users` where `users`.`id` = 2 limit 1						360µs	view::posts.inc.list:3 publicaciones							
select * from `users` where `users`.`id` = 2 limit 1						340µs	view::posts.inc.list:3 publicaciones							

Para solucionar este problema vamos a nuestro modelo ‘Post’ y creamos una nueva propiedad protegida (\$with): `protected $with = ['user'];`.

Al agregar la propiedad \$with al modelo ‘Post’ con el valor ['user'], garantizamos que cada vez que se realice una consulta de las publicaciones, la relación ‘user’ se incluirá automáticamente. Esto significa que al tratar con entidades de tipo ‘Post’, los usuarios asociados se recuperarán simultáneamente durante las consultas desde el controlador.

Si revisamos ahora en el navegador, observamos cómo ya no tenemos consultas desde las vistas y si probamos a añadir una nueva publicación observamos como el valor de las queries no se altera, ya que trae de una vez toda la información de los usuarios.

11/01/2024 publicacion5

Messages

Timeline

Exceptions

Views19

Route

Queries4

Models8

Gate

Session

Request

GET dashboard

21MB

406ms

</> 8.2.4

4 statements were executed16.35ms

select * from `users` where `id` = 2 limit 114.84ms📄\vendor\laravel\framework\src\Illuminate\Auth\EloquentUserProvider.php:59publicaciones

select count(*) as aggregate from `posts`470µs📄\app\Http\Controllers\PostController.php:15publicaciones

select * from `posts` order by `created_at` desc limit 15 offset 0560µs📄\app\Http\Controllers\PostController.php:15publicaciones

select * from `users` where `users`.`id` in (2)480µs📄\app\Http\Controllers\PostController.php:15publicaciones

3.9. Eliminar de la base de datos:

En este punto del tutorial, abordaremos la funcionalidad de eliminar, una parte importante y necesaria para cerrar nuestro ciclo de desarrollo.

Para implementar la eliminación de una publicación, desarrollaremos dentro del listado de publicaciones ('list.blade.php') un formulario:

```
<form action="{{route('posts.destroy', $post->id)}}" method="post">
  @csrf
  @method('delete')
  <button class="text-indigo-600 text-xs">{{__('Delete')}}</button>
</form>
```

- **Action del formulario:** indicamos la ruta que lleva a cabo la eliminación y le pasamos el id del post que deseamos eliminar `route('posts.destroy', $post->id)`.
- **Method:** dado que estamos alterando un elemento en nuestra base de datos, especificamos que el formulario trabaje con el método ‘post’
- **Token CSRF:** incluimos `@csrf` para agregar un campo oculto que contiene el token CSRF necesario para proteger nuestra aplicación.
- **Delete():** usamos `@method('delete')` para establecer el método HTTP que referencia la función de eliminación en el controlador.
- **Botón de eliminación:** Finalmente, incorporamos un botón ‘Delete’, que cuando se presiona, inicia el proceso de eliminación.

```
<form action="http://publicaciones.test/posts/9" method="post"> == $0
  <input type="hidden" name="_token" value="n4qx7EcbdgyTBxiXuRKoxTcbX4JPT0S7LfnCuAZh" autocomplete="off">
  <input type="hidden" name="_method" value="delete">
  <button class="text-indigo-600 text-xs">Delete</button>
</form>
```

judit

16/01/2024 nueva publicación

Delete

En este momento, al pulsar el botón eliminar que hemos creado, nos dirige a una ruta con una pantalla en blanco, ya que a que dentro de nuestro método ‘destroy()’ no tenemos ninguna acción implementada. Por lo tanto, es el momento de desarrollar nuestro método para que realice la tarea que deseamos:

```
public function destroy(Post $post){
    $post->delete();
    return back();
}
```

En este método, recibimos un post por prámetro (‘\$post’), establecemos que vamos a eliminar el post recibido (‘\$post → delete()’) y finalmente retornamos a la vista anterior con ‘return back()’.

Como último paso para concluir este punto del tutorial y aprovechando que hemos implementado con éxito la funcionalidad de eliminación en nuestra aplicación, y considerando que el código relacionado con la lista ha crecido, organizaremos nuestro código de manera más eficiente. Para ello, creamos una nueva vista llamada ‘item.blade.php’, donde colocaremos todo lo relacionado con el item.

```
resources > views > posts > inc > item.blade.php
1  <a href="#" class="text-lg font-semibold">{{ $post->user->name }}</a>
2  <p class="met-1 text-xs">
3      <em>
4          {{ $post->created_at->format('d/m/Y')}}
5      </em>
6      {{ $post->body }}
7  </p>
8  <form action="{{ route('posts.destroy', $post->id) }}" method="post">
9      @csrf
10     @method('delete')
11     <button class="text-indigo-600 text-xs">{{ __('Delete')}}</button>
12 </form>
```

No debemos olvidar incluir esta nueva vista en ‘list.blade.php’.

```
resources > views > posts > inc > list.blade.php
1  @foreach ($posts as $post)
2      <div class="mt-4">
3          @include('posts.inc.item')
4      </div>
5  @endforeach
6
7  <div class="mt-4">
8      {{ $posts->links() }}
9  </div>
```

4. Políticas de acceso:

4.1. Política de acceso básico:

En el siguiente paso del tutorial, abordaremos un fallo de seguridad que hemos identificado en la aplicación. Para ilustrar este problema, nos dirigiremos a la ruta de registro, crearemos un nuevo usuario y generaremos una nueva publicación. En este escenario, observaremos un fallo de seguridad evidente: la posibilidad de visualizar y utilizar el botón ‘eliminar’ (‘Delete’) para una publicación que no pertenece al usuario actual. Es decir, aunque mi usuario sea ‘Usuario’, tengo la capacidad de borrar una publicación de otro usuario, como ‘Judith’.

Para abordar este fallo de seguridad, implementaremos la técnica de políticas de acceso. Una política de acceso establece reglas para determinar si un usuario tiene permiso para realizar ciertas acciones. En este caso, la regla que implementaremos será clara: “Si una publicación no pertenece al usuario actual, no se le permitirá eliminarla”. Esto asegurará que cada usuario solo pueda eliminar sus propias publicaciones.

La implementación de esta política se llevará a cabo directamente en el método `delete()` de nuestro controlador:

```
public function destroy(Request $request, Post $post){
    //dd($request->user()->id);
    if($request->user()->id !== $post->user_id){
        abort(403);
    }
    $post->delete();
    return back();
}
```

Observamos como el método `destroy` acepta 2 parámetros:

- `$request`: se utiliza para acceder a los datos de la solicitud, como el usuario autenticado.
- `$post`: es una instancia del modelo `Post` que hace referencia a la publicación que se intenta eliminar.

La validación del usuario actual se hace de la siguiente manera:

- `$request → user() → id`: accede al id del usuario que hace la solicitud.
- `$post → user_id`: accede al id del usuario que creó la publicación que se intenta eliminar.
- La condición `'if($request → user() → id !== $post → user_id)'` verifica si el usuario actual es el propietario de la publicación.
 - Si no lo es, se ejecuta `'abort(403)'`, generando un error HTTP 403 (Prohibido), indicando que el usuario no tiene permisos para realizar la acción.
 - En caso de que el usuario actual sea el propietario de la publicación, entonces `'$post → delete()'` elimina la publicación de la base de datos y `'return back()'` nos redirige a la página anterior (página desde la que se realizó la solicitud).

Si vamos a nuestro navegador e intentamos eliminar una publicación que no sea nuestra, como hacíamos antes, vemos que ahora no nos lo permite y nos muestra el error http 403

403 | FORBIDDEN

4.2. Política de acceso centralizada:

En este punto aprenderemos a hacer una técnica de política de acceso más limpia. Vamos a aislar nuestro código hacia un archivo central.

Existe un archivo central, que se encuentra en la carpeta Providers y ahí dentro tenemos un archivo que hace referencia a la autenticación AuthServiceProvider.php. Lo podemos aislar ahí. Ahí debemos implementar el método de arranque boot(), método de definición de políticas de acceso. Como vamos a estar trabajando con los usuarios y las publicaciones necesitamos importarlos:

```
use App\Models\Post;  
use App\Models\User;
```

La modificación en el archivo centralizado “AuthProvider.php” sería lo siguiente:

```
public function boot(): void  
{  
    //Metodo/s que se necesiten para validar las acciones del usuario.  
    //En este caso se validan  
    Gate::define('destroy-posts',function(User $user, Post $post){  
        return $user->id === $post->user_id;  
    });  
}
```

EN dicho método se definen: un nombre para la autorización ('destroy-posts') y se crea una función en la que se le pasa el \$user y la clase que vayamos a validar, en este caso Post.

A continuación mostramos 2 usos de la autorización: una es en el postController y otra en la vista directamente.

```
public function destroy(Request $request, Post $post){  
    //En caso de que devuelva true continua y en caso de que devuelva false da un error controlado  
    $this->authorize('destroy-posts', $post);  
  
    $post->delete();  
    return back();  
}
```

```
@can('destroy-posts', $post)  
<form action="{{route('posts.destroy', $post->id)}}" method="post">  
    @csrf  
    @method('delete')  
    <button class="text-indigo-600 text-xs">{{__('Delete')}}</button>  
</form>  
@endcan
```

4.3. Política de acceso estándar:

A continuación vamos a realizar la política de acceso estándar, más adecuada cuando estamos realizando proyectos muy grandes.

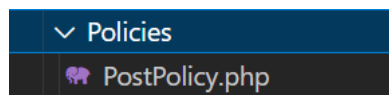
Para emplear este método se va a lanzar la siguiente instrucción:

php artisan make:policy “NombreModel”+policy, en nuestro caso para las publicaciones.

php artisan make:policy PostPolicy

```
PS C:\xampp\htdocs\publicaciones> php artisan make:policy PostPolicy
INFO Policy [C:\xampp\htdocs\publicaciones\app\Policies\PostPolicy.php] created successfully.
```

Se ha creado correctamente el nuevo archivo.



En este fichero se crea el método de validación:

```
class PostPolicy
{
    public function delete(User $user, Post $post){
        return $user->id === $post->user_id;
    }
}
```

Y se modifica el “AuthServiceProvider.php” y se añade lo siguiente, además de comentar el método de “destroy-posts”:

```
/**
 * The model to policy mappings for the application.
 *
 * @var array<class-string, class-string>
 */
protected $policies = [
    //
    Post::class => PostPolicy::class,
];
```

Después hay que modificar los archivos que hacen uso del anterior método de política de acceso, que son la vista de “item” y “PostController”

```
@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
```

```
public function destroy(Request $request, Post $post){
    //dd($request->user()->id);
    /*if($request->user()->id !== $post->user_id){
        abort(403);
    }*/
    //En caso de que devuelva true continua y en caso de que devuelva false da un error controlado
    $this->authorize('delete',$post);

    $post->delete();
    return back();
}
```

5. Mejoras:

5.1. Filtrado de registros:

Aunque el proyecto esencialmente está ya completo, nos interesa aprender a hacer un filtrado por usuario, optimizando la manera en la que el usuario pueda navegar.

Abrimos el terminal y creamos un nuevo controlador y dentro de este crearemos todo lo necesario:

```
php artisan make:controller UserController
```

- Importamos de manera directa a la entidad de los usuarios: `use App\Models\User;`
- Dentro de “UserController” creamos un método llamado ‘show’ que nos permita visualizar todas las publicaciones de un usuario:
 - Se recibe como parámetro al usuario que quiero visualizar.

```
class UserController extends Controller
{
    public function show(User $user){
        return view('users.show',[
            'posts' => $user->post()->latest()->paginate()
        ]);
    }
}
```

Para que esto funcione se necesita definir la ruta: `routes/web.php`:

- Se importa el controlador de usuarios:

```
use App\Http\Controllers\UserController;
```

- Se define la ruta de navegación:

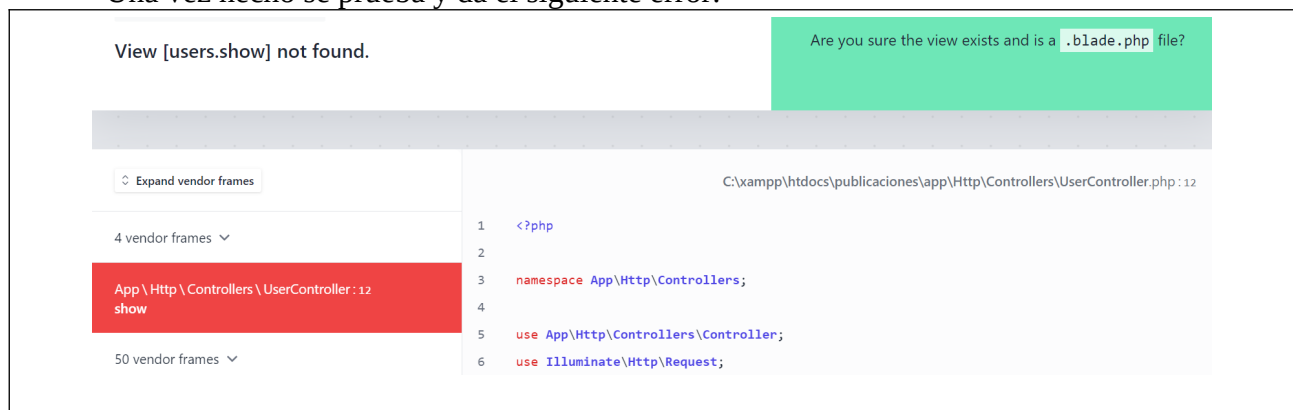
```
Route::get('user/{user}', [UserController::class, 'show'])->middleware(['auth', 'verified'])->name('users.show');
```

Para configurar la navegación desde cada “publicación” se necesita definir la url en el archivo “**item.blade.php**” que se encuentra en la siguiente carpeta “**resources/views/posts/inc/**”.

- Para ello se rellena el href del item, aquí introducimos la llamada a la url anteriormente preparada en el “web.php” y le pasamos como parámetro el id del usuario:

```
href="{{ route('users.show', $post->user->id) }}"
```

- Una vez hecho se prueba y da el siguiente error:



Para subsanar dicho error se debe crear una vista dentro de la siguiente ruta “**resources/views/users**” dicha vista se llamará “**show.blade.php**”.

- Se creará la vista y en la que se mostrará la lista de publicaciones en la que solo se mostrará la información de un usuario:

```
<div class="py-12">
  <div class="max-w-7xl mx-auto sm:px-6 lg:px-8">
    <div class="bg-white dark:bg-gray-800 overflow-hidden shadow-sm sm:rounded-lg">
      <div class="p-6 text-gray-900 dark:text-gray-100">
        @include('posts.inc.list')
      </div>
    </div>
  </div>
</div>
```

- Con ello se mostrará la información a través de la vista que ya tenemos creada, como la variable “posts” viene rellena en el “UserController.php”. En pantalla quedaría así según el usuario conectado.

<div> <p>judit 12/01/2024 publicación 8 Delete</p> <p>judit 11/01/2024 publicacion6 Delete</p> <p>judit 11/01/2024 publicacion5 Delete</p> </div>	<div> <p>Usuario 16/01/2024 B Delete</p> <p>Usuario 16/01/2024 Mi publicación Delete</p> </div>
--	---

5.2. Módulo de idiomas:

Para crear el multi-idioma hay que ir a la estructura de carpeta “/lang” y crear un fichero “es.json”. En este módulo ha surgido un problema, ya que no existía esta estructura creada. Para corregir dicho error cree la estructura “/lang” y dentro 2 archivos “es.json” y “en.js”.

Para hacer las pruebas se crea la siguiente información dentro de los archivos:

<pre>{ "Save": "Guardar" }</pre>	<pre>{ "Save": "Save" }</pre>
------------------------------------	---------------------------------

Y así el sistema es capaz de recuperar la etiqueta que se crea desde los formularios como en los botones:

```
<div class="flex justify-end mt-4">
  <x-primary-button>
    {{ __('Save') }}
  </x-primary-button>
</div>
```

Laravel – Sistemas de opiniones en Laravel 9

En caso de que no exista la definición en el archivo de idioma, pone el literal que se ha definido en el propio botón.

En este punto, nos encontramos con que no funcionaba el multidioma. Al repasar el videotutorial observé que en mi archivo 'app.php' no se encontraba definida la variable 'locale', por lo que tuve que crearla para que cogiera el idioma.

Modificación en el archivo app.php:

```
/*
|-----
|  Locale
|-----
|
| Configuración de locale para poder generar el multiidioma.
|
*/

'locale' => 'es',
```

Aplicación con el multi-idioma funcionando:

<div><input type="text"/></div> <div>GUARDAR</div>	<div><input type="text"/></div> <div>SAVE</div>
--	---

6. Incluir datos “fake”:

Leyendo información y viendo videos he llegado a este <https://www.youtube.com/watch?v=jdeBaEsdV9Q>, en el que se explica correctamente la forma de generarlo. Para ello lo primero que realizaremos será crear una clase Factory y configurarla tal y como se desea que se genere la información.

```
PS C:\xampp\htdocs\publicaciones> php artisan make:factory PostFactory --model=Post
```

```
INFO Factory [C:\xampp\htdocs\publicaciones\database\factories\PostFactory.php] created successfully.
```

Después dentro del Factory se configura la función 'definition()' para indicar cómo se genera la información ficticia para cada modelo ('PostFactory').

```
class PostFactory extends Factory
{
    protected $model = Post::class;
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition(): array
    {
        return [
            'body' => $this->faker->text(100),
            'user_id' => 1
        ];
    }
}
```

Seguidamente vamos al fichero de configuración DatabaseSeeder para configurar la carga de info en la Base de datos que se desea realizar.

```
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     */
    public function run(): void
    {
        User::factory(10)->create();
        Post::factory(100)->create();

        // \App\Models\User::factory()->create([
        //     'name' => 'Test User',
        //     'email' => 'test@example.com',
        // ]);
    }
}
```

Por último se lanza la instrucción para lanzar la creación de Base de datos pero además le añadimos como parametro “--seed” que es lo que hace lanzar la creación de datos fake.

```
PS C:\xampp\htdocs\publicaciones> php artisan migrate:refresh --seed
```

A continuación se muestran un ejemplo de los datos generados automáticamente:

<div><div><div></div><div></div></div></div>				id	user_id	body	created_at	updated_at
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	1	1	Voluptatem laudantium fugiat impedit. Nihil est de...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	2	1	Sit qui ducimus qui pariat. Perferendis expedita...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	3	1	Voluptatum animi autem esse. Error laudantium quib...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	4	1	Aspernatur omnis delectus ullam sed. Enim harum do...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	5	1	Aliquid sit vero voluptates ut. Lure nam ut aut. O...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	6	1	Autem nostrum voluptas autem porro. Et est rerum a...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	7	1	Minus qui id iure dolorem voluptas aliquid quia. A...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	8	1	Corporis placeat dignissimos cupiditate rerum volu...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	9	1	Ab sed et molestiae quia sunt voluptatem natus. Al...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	10	1	Sint placeat ut placeat omnis. Pariatur rerum non ...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	11	1	Hic tempore ut velit et aut maxime. Est labore et ...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	12	1	Illum aut officia quia reiciendis nemo. Ipsum tene...	2024-01-19 21:25:49	2024-01-19 21:25:49
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	13	1	Atque hic dolorum omnis. Nulla voluptas perferendi...	2024-01-19 21:25:50	2024-01-19 21:25:50
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	14	1	Dolor omnis quidem sunt quia vero quam. Officia et...	2024-01-19 21:25:50	2024-01-19 21:25:50
<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	<div><div><div></div><div></div></div></div>	15	1	Neque reiciendis hic officia. Laboriosam amet erro...	2024-01-19 21:25:50	2024-01-19 21:25:50

7. Conclusiones:

Este curso de Laravel ha proporcionado una comprensión integral de la configuración inicial, la estructura MVC y el uso de ‘blade’ para las plantillas. Se resalta la eficiencia al utilizar módulos predefinidos, como el de autenticación, para agilizar el desarrollo. Además, se abordaron aspectos fundamentales como el enrutamiento, controladores, relaciones entre tablas y la implementación de formularios con y sin seguridad.

El curso ha servido para sentar bases sólidas para abordar proyectos profesionales desde 0. La explicación del profesor ha sido un punto destacado, pero cabe señalar que algunas funcionalidades no se autogeneran con la versión actual de Laravel. La complejidad inicial radicó en la instalación de herramientas adicionales, como Composer y Node, además de Laravel.

En definitiva, el curso me parece que ha sido completo y didáctico, proporcionando una buena base para embarcarse en futuros proyectos profesionales.