



# LARAVEL

## FRUTERIA

### ▼ RUTAS



El archivo de configuración de las rutas es → `/routes/web.php`

## 1. Rutas simples

```
Route::get('ruta', function() {  
    // Acción o respuesta  
});
```

- **Ejemplo:**

```
Route::get('fecha', function() {  
    return date("d/m/y h:i:s");  
});
```

- Accediendo a `http://localhost:8000/fecha` (con `php artisan serve`), veremos la fecha/hora actual.

## 2. Rutas con parámetros

### 2.1 Parámetros obligatorios

```
Route::get('saludo/{nombre}', function($nombre) {  
    return "Hola, " . $nombre;  
});
```

- El parámetro `{nombre}` **debe** aparecer en la URL.
- Si no se incluye, retorna error 404.

## 2.2 Parámetros opcionales

```
Route::get('saludo/{nombre?}', function($nombre = "Invitado") {  
    return "Hola, " . $nombre;  
});
```

- Añadir `?` tras el nombre del parámetro lo vuelve opcional.
- Se recomienda asignar un valor por defecto en la función (e.g. `"Invitado"`).

## Validación de parámetros ( `>where` )

- Permite restringir el formato del parámetro con **expresiones regulares**:

```
Route::get('saludo/{nombre?}', function($nombre = "Invitado") {  
    return "Hola, " . $nombre;  
})->where('nombre', '[A-Za-z]+');
```

- Si la expresión no se cumple, se produce un error 404.

## 3. Rutas con nombre (Named Routes)

- Asignar un **nombre** a una ruta con el método `>name('nombre_ruta')`.

```
Route::get('contacto', function() {  
    return "Página de contacto";  
})->name('ruta_contacto');
```

- **Ventaja:** al enlazar, se utiliza `route('nombre_ruta')` en lugar de la URL fija.

```
<a href="{{ route('ruta_contacto') }}">Contacto</a>
```

- Si la ruta cambia en el futuro, basta con actualizarla en `routes/web.php`, sin tocar los enlaces.

## 4. Combinando parámetros y nombrando rutas

- Se pueden encadenar varias restricciones `>where` y nombrar la ruta con `>name(...)`:

```
Route::get('saludo/{nombre?}/{id?}',  
    function($nombre = "Invitado", $id = 0) {  
        return "Hola $nombre, tu código es el $id";  
    }  
)  
→where('nombre', '[A-Za-z]+')  
→where('id', '[0-9]+')  
→name('saludo');
```

- Importante:** si se omite un parámetro, se deben omitir también los que vayan detrás (p. ej. no puedes pasar solo el `id` sin el `nombre`).

## 5. Otros métodos de Route

- Además de `Route::get`, existen:
  - `Route::post` (para manejar datos enviados por formularios).
  - `Route::put` y `Route::patch` (para actualizar recursos).
  - `Route::delete` (para eliminar recursos).
- Se usan principalmente en rutas de tipo **API** o en formularios con métodos HTTP distintos a GET.

## ▼ Vistas con Blade



Los archivos de las vistas se ubican en → `/resources/views` (dentro de esa carpeta pueden haber subcarpetas de vistas)

## 1. Devolver vistas en una ruta

- Sintaxis básica:

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

- No es necesario especificar la extensión `.blade.php` . Laravel la asume por defecto.
- **Ejemplo:** si tenemos `resources/views/inicio.blade.php` :

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

---

### 3. Pasar valores a las vistas

1. Con `with` :

```
return view('inicio')→with('variable_vista', $valor);
```

2. Con array asociativo:

```
return view('inicio')→with(['variable1' ⇒ $valor1, 'variable2' ⇒ $valor2]);
```

3. Parámetro opcional de `view` :

```
return view('inicio', ['variable1' ⇒ $valor1, ...]);
```

4. `compact` (si la variable local se llama igual): LA QUE DEBEMOS USAR, MAS FACIL

```
return view('inicio', compact('nombre'));
```

5. `Route::view` (atajo si sólo devolvemos una vista sencilla):

```
Route::view('/', 'inicio', ['nombre' ⇒ 'Nacho']);
```

## 4. Mostrar datos con Blade

- **Imprimir variable:**

```
{{ $variable }}
```

- **Evitar la protección XSS** (para HTML embebido):

```
{!! $variable !!}
```

- **Ejemplo:**

```
<p>Bienvenido/a {{ $nombre }}</p>
```

## 5. Estructuras de control en Blade

### 5.1. @if , @elseif , @else , @endif

```
@if ($condicion)
    ...
@endif
@elseif ($otraCondicion)
    ...
@else
    ...
@endif
```

### 5.2. @isset / @endisset

- Comprueba si una variable está definida y no es `null`.

### 5.3. Bucles

#### @foreach , @endforeach

```
@foreach ($array as $elemento)
    {{ $elemento }}
@endforeach
```

**@forelse** , **@empty** , **@endforelse**

- Parecido a **@foreach** , pero incluye una sección **@empty** para el caso de array vacío o no definido.

```
@forelse ($libros as $libro)
    <li>{{ $libro["titulo"] }}</li>
@empty
    <li>No se encontraron libros</li>
@endforelse
```

- **Objeto \$loop** dentro de **@foreach** / **@forelse** :
  - **\$loop->index** , **\$loop->count** , **\$loop->first** , **\$loop->last** , etc.

## 6. Enlazar vistas mediante rutas nombradas

- **Nombrar una ruta** con **>name('nombre\_ruta')** :

```
Route::get('contacto', function() {
    return view('contacto');
})->name('ruta_contacto');
```

- **En Blade:**

```
<a href="{{ route('ruta_contacto') }}">Ir a contacto</a>
```

- Así evitas escribir la URL directa, por si cambia la ruta en el futuro.

## 7. Plantillas (Layouts) y Secciones

### 7.1. Definir la plantilla base ( **@yield** )

- En un archivo como **plantilla.blade.php** :

```
<html>
<head>
    <title>@yield('titulo')</title>
</head>
```

```
<body>
    @yield('contenido')
</body>
</html>
```

## 7.2. Extender la plantilla ( `@extends` )

- En la vista específica ( `inicio.blade.php` , por ejemplo):

```
@extends('plantilla')

@section('titulo', 'Inicio')

@section('contenido')
    <h1>Página de inicio</h1>
    <p>Bienvenido/a {{ $nombre }}</p>
@endsection
```

- Puedes tener varias secciones `@yield('algo')` , `@yield('otra')` , etc., y rellenarlas con `@section` .

## 8. Incluir vistas parciales ( `@include` )

- Útil para menús, cabeceras, pies comunes, etc.
- **Ejemplo** de archivo parcial `resources/views/partials/nav.blade.php` :

```
<nav>
    <a href="{{ route('inicio') }}">Inicio</a> |
    <a href="{{ route('listado_libros') }}">Libros</a>
</nav>
```

- Luego, en tu plantilla principal:

```
@include('partials.nav') <!-- aqui ira el menu definido en nav.blade.
php→
@yield('contenido')
```

## 9. Estructurar las vistas en carpetas

- Para mayor organización, se suelen crear subcarpetas dentro de `resources/views` :
  - `resources/views/libros/listado.blade.php`
  - `resources/views/partials/nav.blade.php`
  - etc.
- Al llamar a la vista, se indica el **path** usando puntos (pero también se pueden utilizar `/` ):

```
return view('libros.listado', compact('libros'));
```

## 10. Vistas para páginas de error

- Laravel busca en `resources/views/errors/` si ocurre un error.
- **Ejemplo** para error 404: crear `resources/views/errors/404.blade.php` :

```
@extends('plantilla')
@section('titulo', 'Error 404')
@section('contenido')
    <h1>Documento no encontrado</h1>
@endsection
```

## ▼ Estilos y JavaScript



**Dónde escribir tus estilos y scripts:**

- `resources/sass/app.scss` .
- `resources/js/app.js` .

## 1. Infraestructura para archivos CSS y JavaScript

### 1.1. Gestión de dependencias en la parte del cliente



- **Archivo `package.json` :**
  - Define las dependencias **JavaScript** que se usarán en el proyecto (por ejemplo, `vite` , `laravel-mix` , bibliotecas como `axios` , `sass` , etc.).
  - Ejecutar `npm install` en la raíz del proyecto para instalar todas las dependencias listadas en `package.json` .
- **Carpeta `node_modules/` :**
  - Se crea al hacer `npm install` .
  - No se sube a Git (igual que la carpeta `vendor/` de PHP).

## 1.2. Ubicación de archivos propios

- **CSS/SASS:**
  - Por convención, se ubican en `resources/css/` o `resources/sass/` .
  - El archivo principal suele ser `resources/css/app.css` o `resources/sass/app.scss` .
- **JavaScript:**
  - El archivo principal suele ser `resources/js/app.js` .

```
/* Ejemplo: resources/sass/app.scss */
body {
  background-color: #CCC;
  font-family: Arial, sans-serif;
  text-align: justify;
}
```

```
// Ejemplo: resources/js/app.js
console.log('Hola desde app.js');
```

## 2. Generación automática de CSS y JS

Una vez que has definido tu CSS y JS en las carpetas `resources/` , necesitas **compilar** (o "empaquetar") estos archivos para que estén disponibles en `public/` , listos para uso en producción.

Hay dos herramientas principales:

1. **Laravel Mix** (usado en versiones anteriores a Laravel 9, o primeras versiones de 9).
2. **Vite** (desde las versiones más recientes de Laravel 9 en adelante, es la opción por defecto).

## 2.1. Generación con Laravel Mix

- Se basa en **Webpack** y un archivo de configuración `webpack.mix.js` en la raíz.

- **Ejemplo** `webpack.mix.js` :

```
mix.js('resources/js/app.js', 'public/js')
  .sass('resources/sass/app.scss', 'public/css');
```

- Indica que tome `resources/js/app.js` y genere `public/js/app.js`.
- Toma `resources/sass/app.scss` y genere `public/css/app.css`.

- **Comando:**

```
npm run dev
```

- Genera las versiones para desarrollo (no minificadas).
- Para producción: `npm run production`.

- **Incluir en vistas:**

```
<link rel="stylesheet" href="{{ asset('css/app.css') }}">
<script src="{{ asset('js/app.js') }}"></script>
```

## 2.2. Generación con Vite

- **Archivo** `vite.config.js` en la raíz del proyecto.
- **Ejemplo** (incluido por defecto en Laravel 9+):

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
```

```
plugins: [
  laravel({
    input: ['resources/css/app.css', 'resources/js/app.js'],
    refresh: true,
  }),
],
});
```

- **Incluir en vistas** con la directiva `@vite` :

```
<!doctype html>
<html>
  <head>
    @vite(['resources/css/app.css', 'resources/js/app.js'])
  </head>
  <body>
    ...
  </body>
</html>
```

- **Compilación:**
  - `npm run dev` (modo desarrollo, con Live Reload).
  - `npm run build` (genera build de producción minificada).

## ▼ Controladores



Los controladores se ubican en → `/app/Http/Controllers`

### 1. Comando de creación

```
php artisan make:controller NombreController
```

### 2. Controladores de recursos ( `r` )

- Se crean con `r`

```
php artisan make:controller -r LibroController
```

- Incluyen automáticamente métodos CRUD típicos para una entidad:
  1. **index()**: lista recursos
  2. **create()**: muestra formulario de creación
  3. **store()**: guarda el nuevo recurso
  4. **show(\$id)**: muestra un recurso concreto
  5. **edit(\$id)**: muestra formulario de edición
  6. **update(\$id)**: guarda los cambios de un recurso
  7. **destroy(\$id)**: elimina un recurso

## 3. Usar Controladores en las Rutas

### 3.1. Sintaxis de rutas que apuntan a un método

```
use App\Http\Controllers\LibroController;  
  
Route::get('libros', [LibroController::class, 'index']);  
Route::get('libros/{id}', [LibroController::class, 'show']);
```

- Permite un control más claro sobre cada método y su ruta.

### 3.2. Generar rutas automáticamente con `Route::resource()`

- **Crea las rutas CRUD** para todos los métodos del controlador de tipo `r`.

```
Route::resource('libros', LibroController::class);
```

- Equivale a definir manualmente las rutas para `index`, `show`, `create`, `edit`, `store`, `update`, `destroy`.

### Filtrar rutas con `only()` y `except()`

- `only()`: limita las rutas a los métodos especificados. (solo utilizara las indicadas)

```
Route::resource('libros', LibroController::class)→only(['index','show']);
```

- **except()** : crea todas las rutas salvo las indicadas.

```
Route::resource('libros', LibroController::class)→except(['destroy']);
```

## 4. Estructura de Vistas asociadas a un Controlador

- **Convención:** crear subcarpeta en `resources/views` con el nombre de la entidad.
  - Ej.: `resources/views/libros/`
    - `index.blade.php` , `show.blade.php` , `edit.blade.php` , etc.
- **Ejemplo:** si en `LibroController@show` queremos mostrar la vista `show` :

```
public function show($id) {  
    return view('libros.show', compact('id'));  
}
```

- De este modo se organizan mejor las vistas para cada controlador.

## 5. Renombrar las rutas de un **resource** (traducción de URLs)

- Por defecto, `Route::resource('libros', LibroController::class)` crea rutas como:
  - `/libros/create`
  - `/libros/{id}/edit`
- Para traducir `create` y `edit` , se puede editar `AppServiceProvider` :

```
use Illuminate\Support\Facades\Route;  
  
public function boot()  
{  
    Route::resourceVerbs([
```

```

        'create' ⇒ 'crear',
        'edit' ⇒ 'editar',
    ]);
}

```

- Así, las rutas generadas por `resource` usarán `/libros/crear` y `/libros/{id}/editar`, en lugar de `create` / `edit`.

## ▼ Inyección de dependencias

### 1. Inyectando la petición del usuario ( `Request` )

- Cuando un método de **controlador** necesita la información de la petición (formularios, parámetros, cabeceras...), se le **inyecta** automáticamente un objeto de tipo `Illuminate\Http\Request`.
- **Ejemplo:**

```

use Illuminate\Http\Request;

public function store(Request $request) {
    // Aquí usamos $request para leer datos de la petición
}

```

- Laravel detecta el **tipo** de dato en el parámetro y pasa automáticamente el objeto `Request` adecuado.

### 2. Inyectando la respuesta del servidor ( `Response` )

#### 2.1. Método `response()`

- Permite generar **respuestas personalizadas**.
- **Sintaxis general:**

```

response('Contenido', 201, ['Cabecera1' ⇒ 'Valor1']);

```

- O bien de forma encadenada:

```
response("Mensaje", 201)
    →header('Nombre-Cabecera', 'Valor');
```

## 2.2. Devolver JSON

- Con el método `>json()` podemos devolver **objetos** en formato JSON, indicando también un código de estado HTTP:

```
return response()→json(['dato' ⇒ 'valor'], 201)
    →header('X-Extra', 'algo');
```

## 2.3. Redirecciones con `redirect()`

- **Redirigir** a otra URL:

```
redirect('/');
```

- **Redirigir** a ruta con nombre:

```
redirect()→route('nombre_ruta');
```

- Para que la redirección surta efecto en un controlador, se usa `return redirect()...`

```
public function store() {
    // Acciones...
    return redirect()→route('libros.index');
}
```

- **Enviar datos a la siguiente vista** con `>with()` (se guardan en la sesión y duran sólo la siguiente petición):

```
return redirect()→route('inicio')→with('mensaje', 'Guardado con éxito');
```

- **En la vista**, se accede a esos datos con:

```
@if(session()→has('mensaje'))
    {{ session('mensaje') }}
@endif
```

## 3. Helpers en Laravel

### 3.1. ¿Qué son?

- **Funciones globales** de utilidad que no pertenecen a ninguna clase.
- Ejemplo: queremos una función `setActivo($ruta)` que devuelva la clase CSS `'activo'` si la ruta actual coincide con `$ruta`.

### 3.2. Definir un helper

1. **Crear archivo** `app/helpers.php` :

```
<?php
function setActivo($nombreRuta) {
    return request()→routes($nombreRuta) ? 'activo' : '';
}
```

2. **Registrar el archivo** en `composer.json` (sección `"autoload"` → `"files"`):

```
"autoload": {
    "files": [
        "app/helpers.php"
    ]
}
```

3. **Reconstruir el autoloader:**

```
composer dump-autoload
```

4. **Usar en las vistas** (o controladores):

```
<li class="{{ setActivo('inicio') }}">
    <a href="{{ route('inicio') }}">Inicio</a>
```



</li>

- Si necesitas que coincidan subrutinas, usa `'*': setActivo('libros.*')`

## ▼ Acceso a la base de datos



El archivo se encuentra → `/.env`

### 1. Configuración de la conexión en el archivo `.env`

- **Variables principales:**
  - `DB_CONNECTION` : SGBD a usar (mysql, pgsql, sqlsrv, etc.).
  - `DB_HOST` : Dirección o IP del servidor (e.g., `127.0.0.1`).
  - `DB_PORT` : Puerto de escucha (por defecto, MySQL usa `3306`).
  - `DB_DATABASE` : Nombre de la base de datos.
  - `DB_USERNAME` : Usuario para conectar.
  - `DB_PASSWORD` : Contraseña.
- **Ejemplo ( `.env` ):**

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

- El archivo `config/database.php` define valores por defecto si alguna variable de entorno no está definida:

```
'default' => env('DB_CONNECTION', 'mysql'),
```

### 2. Creación de la base de datos

1. **Crear la base de datos** antes de trabajar con ella en Laravel.
2. Se puede usar un **gestor** como **phpMyAdmin** (en XAMPP) o comandos SQL directos.
3. En el caso de MySQL con phpMyAdmin:
  - Iniciar Apache y MySQL desde XAMPP.
  - Acceder a `http://localhost/phpmyadmin` .
  - Crear la base de datos (p.ej. `biblioteca` ).

## ▼ Las migraciones

### 1. Estructura general de las migraciones

- Se ubican en la carpeta `database/migrations` .
- **Nombre de archivo:** incluye la fecha + descripción (ej. `2023_01_01_000000_crear_tabla_usuarios.php` ).
- Cada migración contiene **dos métodos**:
  1. `up()` : Crea o modifica tablas, columnas, índices, etc.
  2. `down()` : Revierte lo realizado por `up()` .
- **Ejemplo** de método `up()` para crear la tabla `usuarios` :

```
public function up() {  
    Schema::create('usuarios', function(Blueprint $table) {  
        $table->id();  
        $table->string('nombre');  
        $table->string('email')->unique();  
        $table->timestamps(); // crea campos created_at y updated_at  
    });  
}
```

- **Claves principales y composición:**
  - El método `$table->id()` crea un autoincrement `id` como **primary key**.
  - Para claves compuestas: `$table->primary(['campo1', 'campo2']);`
- **Tipos de columna:** `string` , `text` , `longText` , `float` , `integer` , `boolean` , etc.

- **Modificadores:**

- `>unique()` (clave alternativa),
- `>nullable()` (permite `NULL` ),
- `>default($valor)` ,
- `>after('columna')` , etc.

## 2. Creación de migraciones

- **Comando:**

```
php artisan make:migration nombre_migracion
```

- Laravel añade automáticamente la fecha al principio del archivo.
- **Patrones de nombre:** si detecta `create` y `table` , infiere que es para crear una tabla; si detecta `to ... table` , infiere que es para modificarla.
- **Parámetros adicionales:**

```
# Crea una migración para crear la tabla "pedidos"
php artisan make:migration crear_tabla_pedidos --create=pedidos

# Crea una migración para modificar la tabla "usuarios"
php artisan make:migration nuevo_campo_usuario --table=usuarios
```

- **Ejemplo:** Añadir un campo `telefono` en `usuarios` (migración de modificación):

```
public function up() {
    Schema::table('usuarios', function (Blueprint $table) {
        $table->string('telefono')->nullable();
    });
}

public function down() {
    Schema::table('usuarios', function (Blueprint $table) {
        $table->dropColumn('telefono');
    });
}
```

```
});  
}
```

## 3. Ejecutar o revertir migraciones

### 3.1. Ejecutar migraciones

```
php artisan migrate
```

- Lee todas las migraciones **pendientes** (no registradas en la tabla `migrations`) y ejecuta sus métodos `up()`.
- Crea automáticamente la tabla `migrations` para controlar cuáles se han ejecutado.

### 3.2. Revertir migraciones ( `rollback` )

```
php artisan migrate:rollback
```

- Ejecuta los métodos `down()` del **último lote** de migraciones realizadas.
- **Parámetro `-step`** :

Deshace 2 migraciones del último lote, empezando por las más recientes.

```
php artisan migrate:rollback --step=2
```

### 3.3. `migrate:fresh`

```
php artisan migrate:fresh
```

- **Elimina** todas las tablas (incluidas en migraciones) y **las vuelve a crear** desde cero.
- Útil en entornos de **desarrollo** (¡destrutivo en producción!).

## 4. Ejemplo práctico en el proyecto "biblioteca"

1. **Configurar** `.env` con los datos de conexión a la BD y crear la base de datos `biblioteca`.
2. **Eliminar migraciones innecesarias** en `database/migrations`.
3. **Editar la migración de usuarios** (`create_users_table`, por ejemplo):

```
public function up()
{
    Schema::create('usuarios', function (Blueprint $table) {
        $table->id();
        $table->string('login')->unique();
        $table->string('password');
        $table->timestamps();
    });
}
```

1. **Crear migración para la tabla libros:**

```
php artisan make:migration crear_tabla_libros --create=libros
```

Luego, en el método `up()`:

```
public function up()
{
    Schema::create('libros', function (Blueprint $table) {
        $table->id();
        $table->string('titulo');
        $table->string('editorial')->nullable();
        $table->float('precio');
        $table->timestamps();
    });
}
```

1. **Ejecutar migraciones:**

```
php artisan migrate
```

- Como resultado, en la BD `biblioteca` aparecerán:

- `usuarios`
- `libros`
- `migrations` (de control interno de Laravel)
- (opcional) `personal_access_tokens`, si no se ignora Sanctum.

## ▼ El modelo de datos



Los archivos de los modelos se ubican en → `/app/models`

### 1. Creación del modelo

#### 1. Comando:

```
php artisan make:model NombreModelo
```

- Crea la clase del modelo (por defecto en `app/Models` ).
- Por convención, el nombre del modelo va **en singular y en mayúscula**.
- El **modelo** se asocia automáticamente a una tabla en **plural y minúscula** (por ejemplo, `Libro` → `libros` ).

#### 2. Ejemplo:

```
// app/Models/Libro.php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Libro extends Model
{
    // Podemos configurar la tabla si no cumple la convención:
    // protected $table = 'mislibros';
}
```

#### 3. Otras opciones:

- `m`: crea migración junto al modelo.
- `c`: crea controlador vacío.
- `r`: crea controlador de recursos (CRUD).
- **Ejemplo:**

Crearé:

```
php artisan make:model Pelicula -mcr
```

- Modelo `Pelicula`
- Migración `create_peliculas_table`
- `PeliculaController` con métodos `index`, `show`, `create`, etc.

## 2. Estructura y nomenclatura

- **Convenio:**
  - Modelo singular: `Libro`
  - Controlador: `LibroController`
  - Vistas en `resources/views/libros/`
  - Tablas en BD en plural: `libros`
- **Opcional:** si deseas que el modelo use una **tabla distinta** a la del convenio, define:

```
protected $table = 'nombre_tabla';
```

## 3. Eloquent: Primeros pasos

- **Eloquent** es el **ORM** por defecto de Laravel.
- **Patrón Active Record:** las instancias de la clase modelo representan registros de la tabla, con métodos como `save()`, `update()`, `delete()`, etc.

### 3.1. Obtención de datos (Lectura)

1. **Obtener todos los registros:**

```
use App\Models\Libro;

$libros = Libro::get(); // o Libro::all();
```

## 2. Filtrar con where:

```
$libros = Libro::where('precio','<',10)→get();
// Más condiciones:
$libros = Libro::where('precio','<',10)
    →where('precio','>',5)
    →get();
```

## 3. Ordenar con orderBy:

```
$libros = Libro::orderBy('titulo')→get(); // ascendente por defecto
$libros = Libro::orderBy('titulo','DESC')→get();
```

## 4. Paginación:

```
$libros = Libro::paginate(5); // 5 registros por página
```

- En la vista Blade, se añade:

```
{{ $libros→links() }}
```

- Si quieres orden antes de paginar:

```
$libros = Libro::orderBy('titulo')→paginate(5);
```

## Ajuste de paginación con Bootstrap / Tailwind

- Desde **Laravel 8**, por defecto se usa **Tailwind**.
- Para usar **Bootstrap**, en `App\Providers\AppServiceProvider` :

```
use Illuminate\Pagination\Paginator;

public function boot()
```



```
{
    Paginator::useBootstrapFive(); // o Paginator::useBootstrap()
}
```

## 3.2. Mostrar detalle de un registro (show)

1. **Linkear al detalle** en la vista de listado:

```
<a href="{{ route('libros.show', $libro) }}">
    {{ $libro->titulo }}
</a>
```

- Laravel convierte automáticamente `$libro` a su `id` en la URL (si usas `Route::resource('libros', ...)`).

2. **En el controlador:**

```
public function show($id)
{
    $libro = Libro::findOrFail($id);
    return view('libros.show', compact('libro'));
}
```

- `findOrFail($id)` lanza **404** si no encuentra ese registro.
- `find($id)` retorna `null` si no existe.

## 3.3. Inserciones (CREATE)

1. **Instanciar y `save()`** :

```
$libro = new Libro();
$libro->titulo = "El juego de Ender";
$libro->editorial = "Ediciones B";
$libro->precio = 8.95;
$libro->save();
```

2. **Uso de `create()`** :

```
Libro::create($request→all());
```

- **Requisitos:**

- Definir `$fillable` en el modelo:

```
class Libro extends Model
{
    protected $fillable = ['titulo','editorial','precio'];
}
```

- Así, Eloquent **acepta** solo esos campos (evita inserciones maliciosas).

3. **Dónde se hace:** en el método `store` del **controlador**:

```
public function store(Request $request)
{
    Libro::create($request→all());
    // luego redireccionamos o mostramos vista...
}
```

### 3.4. Modificaciones (UPDATE)

1. **Patrón con** `save()` :

```
$libro = Libro::findOrFail($id);
$libro→titulo = "Otro título";
$libro→save();
```

2. **Uso de** `update()` :

```
Libro::findOrFail($id)→update($request→all());
```

- También requiere `$fillable` en el modelo.

3. **Dónde se hace:** en el método `update` del **controlador**.

### 3.5. Borrados (DELETE)

```
Libro::findOrFail($id)→delete();
```

- Normalmente en el método `destroy` del **controlador**.

## Borrado desde la vista

- No se debe usar un simple enlace con `GET`.
- Se usa un **formulario** con método `POST` y directiva `@method('DELETE')`:

```
<form action="{{ route('libros.destroy', $libro) }}" method="POST">
    @csrf
    @method('DELETE')
    <button>Borrar</button>
</form>
```

- Esto envía una **petición DELETE** según las convenciones de Laravel.

**Formularios:** En Laravel, para acciones `PUT`, `PATCH`, o `DELETE`, se usan directivas `@method` y `@csrf` para cumplir los métodos HTTP correctos y proteger la aplicación.

## ▼ Relaciones entre modelos

### 1. Relaciones Uno a Uno (One to One)

#### 1.1. Concepto

- Un registro de una tabla se asocia exactamente con un **solo** registro de otra tabla, y viceversa.
- Ejemplo: Un `Usuario` tiene un `Telefono`.

#### 1.2. Implementación

1. **Migración:** añadir un campo foráneo en una de las tablas.

- Por convención, se añade en la tabla de la entidad "que tiene" la otra.
- Ejemplo: en la tabla `usuarios`, añadir `telefono_id`.

## 2. Modelo principal (Usuario):

```
class Usuario extends Model
{
    public function telefono()
    {
        return $this->hasOne(Telefono::class);
        // Por convención, buscará 'telefono_id' en 'usuarios'
        // y 'id' en 'telefonos'.
    }
}
```

- Si los nombres de los campos difieren de la convención, se indican adicionalmente:

```
return $this->hasOne(Telefono::class, 'claveForanea', 'claveLocal');
```

## 3. Modelo inverso (Telefono):

```
class Telefono extends Model
{
    public function usuario()
    {
        return $this->belongsTo(Usuario::class);
    }
}
```

## 4. Uso:

- Obtener teléfono de un usuario:

```
$telefono = Usuario::findOrFail($id)->telefono;
```

- Obtener usuario desde el teléfono:

```
$usuario = Telefono::findOrFail($idTel)->usuario;
```

## 1.3. Guardar datos relacionados

- **Asignar manualmente la clave foránea:**

```
$usuario→telefono_id = $telefono→id;  
$usuario→save();
```

- **Usar método `associate()` :**

```
$usuario→telefono()→associate($telefono);  
$usuario→save();
```

## 2. Relaciones Uno a Muchos (One to Many)

### 2.1. Concepto

- Un registro de una tabla se asocia con **varios** registros de otra tabla, pero cada registro de la segunda tabla pertenece a un solo registro de la primera.
- Ejemplo: Un `Autor` tiene muchos `Libros`.

### 2.2. Implementación

1. **Migración:** en la tabla hija (ej. `libros`), añadir campo `autor_id`.
2. **Modelo padre (Autor):**

```
class Autor extends Model  
{  
    public function libros()  
    {  
        return $this→hasMany(Libro::class);  
        // Buscará 'autor_id' en 'libros' y 'id' en 'autores'.  
    }  
}
```

3. **Modelo hijo (Libro):**

```
class Libro extends Model
{
    public function autor()
    {
        return $this->belongsTo(Autor::class);
    }
}
```

#### 4. Uso:

- Obtener libros de un autor:

```
$libros = Autor::findOrFail($id)→libros;
```

- Obtener el autor de un libro:

```
$autor = Libro::findOrFail($idLibro)→autor;
```

## 2.3. Ejemplo Práctico (Biblioteca)

- **Migraciones:**

- Crear tabla `autores` con id, nombre, nacimiento, etc.
- En `libros` añadir campo `autor_id`.

- **Modelos:**

```
// Autor.php
class Autor extends Model
{
    protected $table = 'autores';

    public function libros()
    {
        return $this->hasMany(Libro::class);
    }
}

// Libro.php
```

```
class Libro extends Model
{
    public function autor()
    {
        return $this->belongsTo(Autor::class);
    }
}
```

- **Crear un libro asociado:**

```
$autor = Autor::findOrFail(1);
$libro = new Libro();
$libro->titulo = "Ejemplo";
$libro->autor()->associate($autor);
$libro->save();
```

- **Mostrar autor en la vista:**

```
@foreach($libros as $libro)
    <li>{{ $libro->titulo }} ({{ $libro->autor->nombre }})</li>
@endforeach
```

## 2.4. Eager Loading (Carga prematura)

- Para evitar **múltiples** consultas al acceder a la relación, usamos `with()` :

```
$libros = Libro::with('autor')->get();
// Esto carga de una vez todos los autores de la colección de libros
```

- **Uso en Blade:**

Esto no generará consultas adicionales por cada iteración.

```
@foreach($libros as $libro)
    {{ $libro->autor->nombre }}
@endforeach
```

## 3. Relaciones Muchos a Muchos (Many to Many)

### 3.1. Concepto

- Un registro de la tabla A puede estar asociado a varios registros de la tabla B, y viceversa.
- Ejemplo: Un `Usuario` puede tener varios `Rol`, y un `Rol` puede tener varios usuarios.

### 3.2. Tabla intermedia

- Laravel asume la existencia de una **tercera tabla** (pivot) que contiene las referencias de las dos tablas principales.
- Por convención, se nombra en orden alfabético: `rol_usuario` (con campos `rol_id`, `usuario_id`).

### 3.3. Implementación

#### 1. Modelo A (Usuario):

```
class Usuario extends Model
{
    public function roles()
    {
        return $this->belongsToMany(Rol::class);
        // Asume tabla 'rol_usuario' con 'usuario_id' y 'rol_id'.
    }
}
```

#### 2. Modelo B (Rol):

```
class Rol extends Model
{
    public function usuarios()
    {
        return $this->belongsToMany(Usuario::class);
    }
}
```



### 3. Uso:

- Obtener roles de un usuario:

```
$roles = Usuario::findOrFail($id)→roles;
```

- Obtener usuarios de un rol:

```
$usuarios = Rol::findOrFail($idRol)→usuarios;
```

### 4. Si las columnas o el nombre de la tabla pivot difieren:

```
return $this→belongsToMany(Rol::class, 'nombre_tabla_pivot', 'usuario_id', 'rol_id');
```

### 3.4. Acceder a campos de la tabla pivot

- Si la tabla pivot tiene columnas adicionales (p. ej. `created_at`, `algo_extra`), se accede con `$rol→pivot→columna`.

```
$usuario = Usuario::findOrFail($id);  
foreach($usuario→roles as $rol) {  
    echo $rol→pivot→created_at;  
}
```

## ▼ SEEDERS Y FACTORIES (NO ENTRA)

### 1. Seeders

#### 1.1. ¿Qué son?

- **Seeders:** clases que permiten **sembrar** datos iniciales o de prueba en la base de datos.
- Se ubican en `database/seeders` (Laravel 8+).

#### 1.2. Creación de un seeder

```
php artisan make:seeder NombreSeeder
```

- Genera un archivo `NombreSeeder.php` en `database/seeder` .

### 1.3. Definir contenido en el método `run()`

```
public function run()
{
    // Ejemplo creamos un autor y un libro
    $autor = new Autor();
    $autor->nombre = "Juan Seeder";
    $autor->nacimiento = 1960;
    $autor->save();

    $libro = new Libro();
    $libro->titulo = "El libro del Seeder";
    $libro->editorial = "Seeder S.A.";
    $libro->precio = 10;
    $libro->autor()->associate($autor); // si la relación está definida
    $libro->save();
}
```

- Importa ( `use` ) los modelos que vayas a usar (ej. `App\Models\Autor` ).

### 1.4. Añadir seeders en el seeder principal `DatabaseSeeder`

- Ubicado en la misma carpeta `database/seeder` .
- Incluir llamadas a tus seeders:

```
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        $this->call([
            AutoresSeeder::class,
            LibrosSeeder::class,
        ]);
    }
}
```

- **Importante** el **orden** si un seeder depende de datos creados en otro.

## 1.5. Ejecutar seeders

```
php artisan db:seed
```

- Ejecuta todos los seeders registrados en `DatabaseSeeder` .
- Para un seeder concreto:

```
php artisan db:seed --class=LibrosSeeder
```

- Para **reiniciar** la base de datos y sembrarla otra vez:

```
php artisan migrate:fresh --seed
```

- Borra todas las tablas y vuelve a ejecutar las migraciones y seeders.

## 2. Factories

### 2.1. ¿Qué son?

- **Factories**: clases para **generar datos de prueba** de manera rápida y masiva (generalmente se combinan con seeders).
- Se ubican en `database/factories` .

### 2.2. Creación de un factory

```
php artisan make:factory NombreFactory --model=NombreModelo
```

- Genera `NombreFactory.php` en `database/factories/` .
- Asocia el factory al **modelo** ( `-model=NombreModelo` ).

### 2.3. Uso del trait `HasFactory`

- Desde Laravel 8, los modelos generados suelen incluir el trait `HasFactory` .
- Ejemplo:

```
use Illuminate\Database\Eloquent\Factories\HasFactory;
```

```
class Autor extends Model
{
    use HasFactory;
    ...
}
```

- Esto permite hacer: `Autor::factory()→...`

## 2.4. Método `definition()` dentro del factory

```
public function definition()
{
    return [
        // Campos del modelo:
        'nombre' => $this->faker->name,
        'nacimiento' => $this->faker->numberBetween(1950, 1990),
    ];
}
```

- Se usa `$this->faker` para generar datos realistas (ejemplo: `name`, `text`, `email`, `randomNumber`, etc.).

## 2.5. Invocar el factory en un seeder

```
class AutoresSeeder extends Seeder
{
    public function run()
    {
        Autor::factory()
            ->count(5) // Generar 5 autores
            ->create();
    }
}
```

- Esto creará 5 autores con datos pseudo-aleatorios.

## 3. Ejemplo Completo

### 3.1. Modelos

Supongamos que tenemos dos modelos: `Autor` y `Libro`, cada uno con su `HasFactory`:

```
class Autor extends Model
{
    use HasFactory;

    // Relación: un autor puede tener muchos libros
    public function libros()
    {
        return $this->hasMany(Libro::class);
    }
}
```

```
class Libro extends Model
{
    use HasFactory;

    // Relación: un libro pertenece a un autor
    public function autor()
    {
        return $this->belongsTo(Autor::class);
    }
}
```

### 3.2. Factories

**AutorFactory** ( `database/factories/AutorFactory.php` ):

```
public function definition()
{
    return [
        'nombre' => $this->faker->name,
        'nacimiento' => $this->faker->numberBetween(1950, 1990),
    ];
}
```

**LibroFactory** ( `database/factories/LibroFactory.php` ):

```
public function definition()
{
    return [
        'titulo' => $this->faker->sentence,
        'editorial' => $this->faker->sentence(2),
        'precio' => $this->faker->randomFloat(2, 5, 20),
        'autor_id' => null, // se rellenará al crear
    ];
}
```

### 3.3. Seeders

#### **AutoresSeeder**

```
class AutoresSeeder extends Seeder
{
    public function run()
    {
        // Crear 5 autores aleatorios
        Autor::factory()
            ->count(5)
            ->create();
    }
}
```

#### **LibrosSeeder**

```
class LibrosSeeder extends Seeder
{
    public function run()
    {
        $autores = Autor::all();

        // Para cada autor, crear 2 libros
        $autores->each(function($autor) {
            Libro::factory()->count(2)->create([
```

```

        'autor_id' => $autor->id
    });
});
}
}

```

### 3.4. Registrar en DatabaseSeeder

```

class DatabaseSeeder extends Seeder
{
    public function run()
    {
        $this->call([
            AutoresSeeder::class,
            LibrosSeeder::class,
        ]);
    }
}

```

- El orden es importante: primero se crean los autores, luego los libros.

### 3.5. Ejecutar

```

# Si quieres arranque limpio:
php artisan migrate:fresh --seed

# O solo sembrar datos sin resetear:
php artisan db:seed

```

## ▼ Definición y uso de formularios

### 1. Creación de formularios

#### 1.1. Estructura básica en Blade

- Un formulario en Laravel utiliza la misma **sintaxis de HTML**, con la particularidad de poder aprovechar las directivas de **Blade**:

```

<form action="{{ route('ruta.store') }}" method="POST">
    @csrf <!-- Protección CSRF -->
    <!-- Campos input -->
    <button type="submit">Enviar</button>
</form>

```

- `@csrf` : evita errores 419 (protección contra ataques CSRF).

## 1.2. Ejemplo de formulario para crear un recurso (Libros)

```

@extends('plantilla')
@section('titulo', 'Nuevo libro')

@section('contenido')
    <h1>Nuevo libro</h1>
    <form action="{{ route('libros.store') }}" method="POST">
        @csrf
        <div>
            <label for="titulo">Título:</label>
            <input type="text" name="titulo" id="titulo">
        </div>
        <div>
            <label for="editorial">Editorial:</label>
            <input type="text" name="editorial" id="editorial">
        </div>
        <div>
            <label for="precio">Precio:</label>
            <input type="text" name="precio" id="precio">
        </div>
        <div>
            <label for="autor">Autor:</label>
            <select name="autor" id="autor">
                @foreach($autores as $autor)
                    <option value="{{ $autor->id }}">{{ $autor->nombre }}</option>
                @endforeach
            </select>
        </div>
        <button type="submit">Crear libro</button>
    </form>

```



```
</form>
@endsection
```

- Observa el uso de `@csrf`.
- Se envía a la ruta `libros.store` definida en `Route::resource('libros', LibroController::class)`.

## 2. Rutas y Controlador asociados

### 2.1. Ruta de tipo resource

Si hemos definido:

```
Route::resource('libros', LibroController::class);
```

Entonces, se crean automáticamente rutas como:

- `libros.create` (GET `/libros/create`) → Muestra el formulario.
- `libros.store` (POST `/libros`) → Procesa la inserción.

### 2.2. Método `create()` en el controlador

- Carga la vista del formulario:

```
public function create()
{
    $autores = Autor::all();
    return view('libros.create', compact('autores'));
}
```

- Pasa datos adicionales (p. ej. lista de autores) a la vista.

### 2.3. Método `store()` en el controlador

- Recibe los datos del formulario a través de un objeto `Request`.
- **Ejemplo:**

```
public function store(Request $request)
{
    $libro = new Libro();
```

```

$libro→titulo = $request→get('titulo');
$libro→editorial = $request→get('editorial');
$libro→precio = $request→get('precio');

// Asociar el autor (belongsTo)
$autor = Autor::findOrFail($request→get('autor'));
$libro→autor()→associate($autor);

$libro→save();

return redirect()→route('libros.index');
}

```

- **Otros métodos de `Request` :**

- `$request→has('campo')` (comprueba si existe un campo).
- `$request→only(['campo1','campo2'])` , `$request→except([...])` .

## 3. Actualización (UPDATE) y borrado (DELETE)

### 3.1. Método HTTP en formularios

- HTML sólo soporta `GET` y `POST` .
- Para usar `PUT` , `PATCH` o `DELETE` , en **Laravel**:

```

<form action="{ route('ruta.update', $id) }" method="POST">
  @csrf
  @method('PUT')
  <!-- Campos -->
  <button type="submit">Actualizar</button>
</form>

```

- Laravel interpretará esta petición como `PUT` o `DELETE` , etc., gracias a la directiva `@method` .

### 3.2. Ejemplo: Formulario para borrar un libro

- En la vista `show.blade.php` (detalle del libro):

```
<form action="{{ route('libros.destroy', $libro->id) }}" method="POST">
    @csrf
    @method('DELETE')
    <button type="submit" class="btn btn-danger">Borrar libro</button>
</form>
```

- Laravel enviará una **petición DELETE** a `/libros/{id}`.

### 3.3. Método `destroy()` en el controlador

- Se encarga de **borrar** el recurso y hacer una redirección o retornar una vista:

```
public function destroy($id)
{
    $libro = Libro::findOrFail($id);
    $libro->delete();

    return redirect()->route('libros.index')
        ->with('mensaje', 'Libro borrado con éxito');
}
```

- Añade un `>with()` si quieres enviar un mensaje a la próxima vista.

## 4. Añadir enlaces a formularios

- Para mostrar el formulario de creación o edición, se suele poner un enlace en algún menú o en la lista de recursos:

```
<a href="{{ route('libros.create') }}">Nuevo libro</a>
```

- Para edición, sería:

```
<a href="{{ route('libros.edit', $libro->id) }}">Editar</a>
```

- Para borrado, se prefiere un **formulario** en lugar de un simple enlace con GET.

## ▼ Validación de formularios

### 1. Validación básica con `validate()`

1. **Uso sencillo** en el controlador, dentro de `store()` o `update()` :

```
public function store()
{
    request()→validate([
        'titulo' ⇒ 'required|min:3',
        'editorial'⇒ 'required',
        'precio' ⇒ 'required|numeric|min:0'
    ]);

    // Si pasa la validación, continúa la lógica:
    // ...
}
```

2. **Reglas de validación** se separan con `|` (p.ej. `'titulo' ⇒ 'required|min:3'` ).

## 2. Form Requests para validaciones más complejas

1. **Creación:**

```
php artisan make:request LibroPost
```

- Genera una clase en `app/Http/Requests/` .

2. **Contenido** de la clase (ejemplo `LibroPost` ):

```
namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class LibroPost extends FormRequest
```

```

{
    public function authorize()
    {
        // true si no requerimos lógica de autorización
        return true;
    }

    public function rules()
    {
        return [
            'titulo' => 'required|min:3',
            'editorial' => 'required',
            'precio' => 'required|numeric|min:0',
        ];
    }

    public function messages()
    {
        return [
            'titulo.required' => 'El título es obligatorio',
            // ...
        ];
    }
}

```

### 3. Uso en el controlador:

```

public function store(LibroPost $request)
{
    // Si estamos aquí, la validación ha pasado
    // ...
}

```

- No necesitas llamar a `validate()` dentro del método; se hace automáticamente.

## 3. Mostrar mensajes de error

1. **Retorno a la vista:** Si falla la validación, Laravel redirige de nuevo al formulario y pasa un objeto con **errores**.
2. **Listar errores:**

```
@if ($errors->any())
    <ul>
        @foreach ($errors->all() as $error)
            <li>{{ $error }}</li>
        @endforeach
    </ul>
@endif
```

3. **Errores por campo:**

```
<input type="text" name="titulo" id="titulo">
@if($errors->has('titulo'))
    <div class="text-danger">{{ $errors->first('titulo') }}</div>
@endif
```

4. **Mensajes personalizados:**

- **En form request** ( `messages()` method).
- **En controlador** como segundo parámetro en `validate()` :

```
request()->validate([
    'titulo' => 'required|min:3'
], [
    'titulo.required' => 'El título es obligatorio'
]);
```

---

## 4. Recordar valores enviados (old inputs)

1. **Mantener valores tras un error:**

```
<input type="text" name="titulo" value="{{ old('titulo') }}">
```

- Si el usuario vuelve al formulario tras error, se mostrará el valor anterior.

2. En `<textarea>` :

```
<textarea name="comentario">{{ old('comentario') }}</textarea>
```

## ▼ Autenticación basada en sesiones

### 1. Configuración de la Autenticación

#### 1.1. Archivo `config/auth.php`

- Define cómo se autentican los usuarios y de dónde se obtienen (guards y providers).
- **Guards**: mecanismo de autenticación (sesiones, tokens, etc.).
- **Providers**: define **cómo** obtenemos los usuarios (Eloquent, Query Builder...).
- **Ejemplo** con `Eloquent` y modelo `Usuario` :

```
'providers' => [
    'users' => [
        'driver' => 'eloquent',
        'model' => App\Models\Usuario::class,
    ],
    // ...
],
```

- Aquí indicamos el **modelo** de usuario ( `App\Models\Usuario` ).

#### 1.2. El modelo (o tabla) de usuarios

- Si usamos `Eloquent` , debemos tener un **modelo** que represente la tabla de usuarios (p. ej. `Usuario` ).
- Si usamos `driver => 'database'` , necesitamos especificar la **tabla** directamente.

#### 1.3. Creación de un usuario (ej. con `bcrypt` )

- Se recomienda almacenar passwords **encriptados** con `bcrypt`:

```
$usuario = new Usuario();  
$usuario->login = 'admin';  
$usuario->password = bcrypt('admin');  
$usuario->save();
```

- Podemos usar un **seeder** para crear usuarios iniciales:

y dentro del  
`run()` creamos el usuario.

```
php artisan make:seeder UsuariosSeeder
```

## 2. Añadir autenticación a un proyecto

### 2.1. Formulario de login

- Crear vista `resources/views/auth/login.blade.php` con un formulario:

```
@extends('plantilla')  
@section('titulo', 'Login')  
  
@section('contenido')  
<h1>Login</h1>  
@if (!empty($error))  
    <div class="text-danger">  
        {{ $error }}  
    </div>  
@endif  
  
<form action="{{ route('login') }}" method="POST">  
    @csrf  
    <div>  
        <label for="login">Login:</label>  
        <input type="text" name="login" id="login">  
    </div>  
    <div>
```



```

<label for="password">Password:</label>
<input type="password" name="password" id="password">
</div>
<button type="submit">Enviar</button>
</form>
@endsection

```

## 2.2. Controlador de Login

1. Generar controlador:

```
php artisan make:controller LoginController
```

2. **Mostrar el formulario:**

```

public function loginForm()
{
    return view('auth.login');
}

```

3. **Procesar login** usando `Auth::attempt` (recuerda `use Illuminate\Support\Facades\Auth;`):

```

public function login(Request $request)
{
    $credenciales = $request->only('login', 'password');

    if (Auth::attempt($credenciales)) {
        // Autenticación exitosa
        return redirect()->intended(route('libros.index'));
    } else {
        $error = 'Usuario o password incorrectos';
        return view('auth.login', compact('error'));
    }
}

```

- `Auth::attempt()` verifica usuario y contraseña (usa bcrypt para comparar).

- `redirect()→intended(...)` manda al usuario a la ruta que originalmente pidió.

## 2.3. Rutas asociadas

```
use App\Http\Controllers\LoginController;

Route::get('login', [LoginController::class, 'loginForm'])→name('login');
Route::post('login', [LoginController::class, 'login']);
```

- Laravel redirige automáticamente a `route('login')` cuando un usuario no autenticado intenta acceder a una ruta protegida.

## 2.4. Proteger rutas (Middleware `auth` )

- Para **controladores resource** (ej. `LibroController`):

```
class LibroController extends Controller
{
    public function __construct()
    {
        $this→middleware('auth', [
            'only' => ['create','store','edit','update','destroy']
        ]);
    }
    // ...
}
```

- Así, sólo usuarios **autenticados** pueden crear, editar, o borrar libros.
- O bien, en **definición de rutas**:

```
Route::get('prueba', [PruebaController::class, 'create'])→middleware
('auth');
```

## 2.5. Detectar al usuario autenticado en las vistas

- `auth()→check()` : Devuelve `true` si el usuario está logueado.
- `auth()→guest()` : Devuelve `true` si NO está autenticado.
- `auth()→user()` : Devuelve el objeto del usuario autenticado.

```
@if(auth()→check())  
    <p>Bienvenido, {{ auth()→user()→login }}</p>  
@endif
```

- Podemos usarlo para mostrar/ocultar enlaces (por ejemplo, enlace a `libros.create` sólo si está autenticado).

### 3. Logout (cerrar sesión)

#### 1. Controlador:

```
public function logout()  
{  
    Auth::logout();  
    return redirect()→route('login'); // o a donde prefieras  
}
```

#### 2. Ruta:

```
Route::get('logout', [LoginController::class, 'logout'])→name('logout');
```

#### 3. Enlace en la vista (p.ej. menú):

```
@if(auth()→check())  
    <li><a href="{{ route('logout') }}">Logout</a></li>  
@endif
```

### 4. Definir roles y middleware personalizado

1. **Añadir un campo** `rol` a la tabla de usuarios (migración).
2. **Mostrar/ocultar contenido** en vistas, según el rol:

```
@if(auth()→user()→rol === 'admin')  
    <p>Eres admin</p>  
@endif
```

### 3. Middleware propio para roles:

- Crear:

```
php artisan make:middleware RolCheck
```

- Editar `handle` en `App\Http\Middleware\RolCheck.php` :

```
public function handle($request, Closure $next, $rol)
{
    if (auth()→user()→rol === $rol) {
        return $next($request);
    } else {
        return redirect('/');
    }
}
```

- Registrar en `app/Http/Kernel.php` :

```
protected $routeMiddleware = [
    // ...
    'roles' => \App\Http\Middleware\RolCheck::class,
];
```

- Usar en el constructor del controlador:

```
public function __construct()
{
    $this→middleware(['auth','roles:admin']);
}
```

- Aquí `roles:admin` indica que sólo usuarios con rol `'admin'` pueden acceder.