UT8: LARAVEL

¿Qué es Laravel?

Laravel es un framework de PHP diseñado para crear aplicaciones web de manera rápida, sencilla y estructurada. Utiliza una arquitectura **MVC (Modelo-Vista-Controlador)** que separa la lógica de negocio (Modelos), las vistas (HTML) y el control de las peticiones (Controladores).

- **Modelo**: Contiene la lógica de negocio y maneja los datos.
- **Controlador**: Actúa como intermediario entre la Vista y el Modelo.
- **Vista**: Muestra los datos al usuario.

Ventajas principales de Laravel:

- **Fácil de usar:** Ofrece herramientas para tareas comunes como autenticación, enrutamiento y manejo de bases de datos.
- **ORM Eloquent:** Simplifica el acceso a bases de datos.
- **Motor de plantillas Blade:** Permite diseñar vistas de forma rápida y limpia.
- **Comunidad amplia:** Excelente documentación y muchos recursos disponibles.
- **PHP moderno:** Compatible con las últimas versiones de PHP y buenas prácticas.

Requisitos previos

- **PHP 8+**: Laravel requiere una versión moderna de PHP.
- **Composer**: Administrador de dependencias de PHP.
- **Servidor local:** Puedes usar XAMPP, WAMP o Laravel Valet (si estás en Mac).

Instalación de Laravel

Se supone que ya tienes una copia de PHP instalada en tu sistema local.

Composer es un gestor de paquetes y dependencias. Para ejecuta este comando:

```
curl -Ss getcomposer.org/installer | php
```

Verás que se descarga y compila el script composer.phar, que es el que usamos para instalar Laravel. Aunque hay muchas formas de configurar una nueva aplicación Laravel, lo haremos a través del script composer.phar de Laravel. Para instalar este script, ejecuta:

```
composer global require laravel/installer
```

Esto descargará e instalará todos los archivos del framework, así como todas las dependencias que requiere. Los paquetes se guardarán dentro del directorio del proveedor.

Para crear una nueva aplicación ejecutar el siguiente comando:

```
laravel new nombre_Aplicacion
```

VSCode extensiones

- Recomendable instalar los siguientes plugins para Visual Studio Code

```
Laravel Blade Snippets
```

Carpetas en Laravel

Al crear un nuevo proyecto con este framework, Laravel crea una serie de carpetas por defecto. Esta estructura de carpetas es la recomendada para utilizar Laravel.

Public

Esta es la carpeta más importante ya que es donde se ponen todos los archivos que el cliente va a mostrar al usuario cuando introduzcamos la URL de nuestro sitio web. Normalmente se carga el archivo index.php por defecto.

Routes

Otra de las carpetas que más vamos a usar. En ella se albergan todas las rutas (redirecciones web) de nuestro proyecto, pero más concretamente en el archivo web.php

```
Dada una ruta → se cargará una vista
```

Resources

Esta es nuestra carpeta de recursos donde guardaremos los siguientes archivos, que también, están separados por sus carpetas... como cada nombre indica:

- *css* Archivos CSS
- *js* Archivos JS (JavaScript)
- *lang* Archivos relacionados con el idioma del sitio (variables & strings)
- *views* Archivos de nuestras vistas (Vistas Blade (HTML dinámico)), lo que las rutas cargan

Rutas

Las rutas en Laravel (y en casi cualquier Framework) sirven para redireccionar al cliente (o navegador) a las vistas que nosotros queramos.

Estas rutas se configuran en el archivo *routes/web.php* donde se define la ruta que el usuario pone en la URL después del dominio y se retorna la vista que se quiere cargar al introducir dicha dirección en el navegador.

Por ejemplo, la ruta:

```php
<?php

// Ruta por defecto para cargar la vista welcome cuando el usuario introduce simplemente el dominio

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

Indica que, al acceder a la URL raíz (/), Laravel mostrará la vista llamada welcome, ubicada en resources/views/welcome.blade.php.

Los archivos Blade (que veremos un poco más adelante) tienen la extensión *.blade.php* y se almacenan en el directorio:

resources/views

Un archivo llamado *welcome.blade.php* estará ubicado en *resources/views/welcome.blade.php*.

Alias

Es interesante darle un alias o un nombre a nuestras rutas para poder utilizar dichos alias en nuestras plantillas de Laravel que veremos más adelante.

Para ello, basta con utilizar la palabra *name* al final de la estructura de la ruta y darle un nombre que queramos; normalmente descriptivo y asociado a la vista que tiene que cargar el enroutador de Laravel.

```php
<?php

Route::get('/users', function () {
    return view('users');
}) -> name('usuarios');
```

Después veremos que es muy útil ya que a la hora de refactorizar o hacer un cambio, si tenemos enlaces o menús de navegación que apuntan a esta ruta, sólo tendríamos que cambiar el parámetro dentro del get() y no tener que ir archivo por archivo.

Laravel nos proporciona una manera más cómoda a la hora de cargar una vista si no queremos parámetros ni condiciones. Tan sólo definiremos la siguiente línea que hace referencia la ruta datos en la URL y va a cargar el archivo usuarios.php de nuestra carpeta views como le hemos indicado en el segundo parámetro.

```php
<?php

/* http://localhost/datos/ */

Route::view('datos', 'usuarios');
```

- *Route::view:* Es un método que conecta una URL directamente con una vista sin necesidad de definir una función.
- Primer parámetro: *'datos':* Define la URL a la que el usuario accede, en este caso: http://localhost/datos.
- Segundo parámetro: *'usuarios':* Es el nombre de la vista que se cargará, ubicada en el directorio resources/views/usuarios.blade.php.

Pero no sólo podemos retornar una vista sino, desde un simple string a módulos propios de Laravel.

Parámetros

Ya hemos visto que con PHP podemos pasar parámetros a través de la URL, como si fueran variables, que las recuperábamos a través del método GET o POST.

Con Laravel también podemos introducir parámetros pero de una forma más vistosa y ordenada, de tal manera que sea visualmente más cómodo de recordar y de indexar por los motores de búsqueda como Google.

```
http://localhost/cliente/324
```

Para configurar este tipo de rutas en nuestro archivo de rutas `public/routes/web.php` haremos lo siguiente.

```php
<?php

Route::get('cliente/{id}', function($id) {
    return('Cliente con el id: ' . $id);
});
```

- *Route::get('cliente/{id}', ...):*
    - Define una ruta que acepta un parámetro dinámico {id} después de /cliente/.
    - Cuando alguien visita una URL como http://localhost/cliente/324, Laravel reconoce 324 como el valor del parámetro {id}.
- *function($id):*
    - Laravel pasa automáticamente el valor del parámetro {id} a esta función como la variable $id.
- *return 'Cliente con el id: ' . $id;:*
    - Devuelve una respuesta de texto que incluye el valor del parámetro.
    - Por ejemplo, si accedes a http://localhost/cliente/324, la respuesta será:

      Cliente con el id: 324

¿Qué ocurre si no pasas el parámetro?

- Si accedes a http://localhost/cliente/ sin un parámetro {id}, Laravel mostrará un error 404 (Not Found), porque la estructura de la URL no coincide con la ruta definida.

Para resolver ésto, podemos definir una ruta por defecto en caso de que el id (o parámetro) no sea pasado. Para ello usaremos el símbolo ? en nuestro nombre de ruta e inicializaremos la variable con el valor que queramos.

```php
<?php

Route::get('cliente/{id?}', function($id = 1) {
    return ('Cliente con el id: ' . $id);
});
```

- *Route::get('cliente/{id?}', ...):*
    - Define una ruta que acepta un parámetro dinámico {id}.
    - El signo de interrogación ? después de {id} indica que este parámetro es opcional. Es decir, la URL puede incluirlo o no.
- *function($id = 1):*
    - Si el parámetro id no es proporcionado en la URL, Laravel asignará automáticamente el valor predeterminado 1 a la variable $id.
- *return 'Cliente con el id: ' . $id;:*
    - Devuelve una respuesta que incluye el valor de $id. Dependiendo de si el parámetro fue pasado o no, el texto cambiará.

Ahora tenemos otro problema, porque estamos filtrando por id del cliente que, normalmente es un número, pero si metemos un parámetro que no sea un número, vamos a obtener un resultado no deseado.

Para resolver este caso haremos uso de la cláusula where junto con una expresión regular númerica.

```php
<?php

Route::get('cliente/{id?}', function($id = 1) {
    return ('Cliente con el id: ' . $id);
}) -> where('id', '[0-9]+');
```

- *Route::get('cliente/{id?}', ...):*
    - Define una ruta que acepta un parámetro opcional {id}.
    - Si el parámetro no se proporciona, tomará el valor por defecto 1.
- *->where('id', '[0-9]+'):*
    - Usa la cláusula **where** para validar el parámetro id con una expresión regular.

- *[0-9]+:*
    - **[0-9]**: Acepta solo números del 0 al 9.
    - **+**: Indica que debe haber al menos un dígito (uno o más números).

Comportamiento esperado

1. Cuando el parámetro es un número:
    - URL: http://localhost/cliente/324
      Respuesta: Cliente con el id: 324
2. Cuando no se proporciona el parámetro:
    - URL: http://localhost/cliente/
      Respuesta: Cliente con el id: 1
3. Cuando el parámetro no es un número:
    - URL: http://localhost/cliente/abc
      Resultado: Laravel devuelve un error 404 (Not Found).

Además, podemos pasarle variables a nuestra URL para luego utilzarlas en nuestros archivos de plantillas o en archivos .php haciendo uso de un array asociativo. Veamos un ejemplo con la forma reducida para ahorrarnos código

```php
<?php

Route::view('datos', 'usuarios', ['id' => 5446]);
```

- *Route::view('datos', 'usuarios'):*
    - datos: Define la URL que el usuario visitará, en este caso: http://localhost/datos.
    - usuarios: Indica la vista que se cargará, ubicada en el archivo: resources/views/usuarios.blade.php.
- *['id' => 5446]:*
    - Es un array asociativo que contiene las variables que se pasan a la vista.
    - En este caso, estamos pasando la variable id con el valor 5446 para que esté disponible en la plantilla Blade.

... y el archivo `resources/views/usuarios.blade.php` debe tener algo parecido a esto

```php
<!-- Estructura típica de un archivo HTML5 -->

<p>Usuario con id: <?= $id ?></p>

<!-- ... -->
```

Puedes pasar múltiples variables a la vista utilizando un array asociativo.

```
Route::view('datos', 'usuarios', ['id' => 5446, 'nombre' => 'Manolo']);
```

Con las plantillas de Laravel blade.php veremos cómo simplificar aún más nuestro código.

Para más información acerca de las rutas, parámetros y expresiones regulares en las rutas puedes echar un vistazo a la documentación oficial de rutas que contiene numerosos ejemplos.

Actividad 1:

Haz un ejemplo de cada caso que hemos visto y alguno nuevo que encuentres en la documentación oficial y pruebalos.

Plantillas o Templates

A través de las plantillas de Laravel vamos a escribir menos código PHP y vamos a tener nuestros archivos mejor organizados.

Blade es el sistema de plantillas que trae Laravel, por eso los archivos de plantillas que guardamos en el directorio de views llevan la extensión blade.php.

De esta manera sabemos inmediatamente que se trata de una plantilla de Laravel y que forma parte de una vista que se mostrará en el navegador.

Directivas

- Laravel tiene un gran número de directivas que podemos utilizar para ahorrarnos mucho código repetitivo entre otras funciones.
- Digamos que las directivas son pequeñas funciones ya escritas que aceptan parámetros y que cada una de ellas hace una función diferente dentro de Laravel.
- *@yield* Marca una sección en la plantilla principal donde se cargará contenido dinámico. Se usa conjuntamente con *@section*.
- *@section* y *@endsection*. Usados para definir bloques de contenido dinámico en las vistas que extienden la plantilla.
- *@extends* importa el contenido de una plantilla ya creada.

Separando código

Veamos un ejemplo de cómo hacer uso del poder de Laravel para crear plantillas y no repetir código.

Supongamos que tenemos ciertas estructuras HTML repetidas como puede ser una cabecera header, un menú de navegación nav y un par de secciones que hacen uso de este mismo código.

Supongamos que tenemos 2 apartados en la web:

- Blog
- Fotos

Primero de todo tendremos que generar un archivo que haga de plantilla de nuestro sitio web.

Para ello creamos el archivo *plantilla.blade.php* dentro de nuestro directorio de plantillas *resources/views*.

Dicho archivo va a contener el típico código de una página simple de HTML y en el body añadiremos nuestros contenido estático y dinámico.

```
<body>
    <!-- CONTENIDO ESTÁTICO PARA TODAS LAS SECCIONES -->
    <h1>Bienvenid@s a Laravel</h1>
    <hr>

    <!-- MENÚ -->
    <nav>
        <a href={{ route('noticias') }}>Blog</a> |
        <a href={{ route('galeria') }}>Fotos</a>
    </nav>

    <!-- CONTENIDO DINÁMICO EN FUNCIÓN DE LA SECCIÓN QUE SE VISITA -->
    <header>
        @yield('apartado')
    </header>
</body>
```

Cada sección que haga uso de esta plantilla contendrá el texto estático *Bienvenid@s a Laravel* seguido de un menú de navegación con enlaces a cada una de las secciones y el contenido dinámico de cada sección.

Ahora crearemos los archivos dinámicos de cada una de las secciones, en nuestro caso *blog.blade.php* y *fotos.blade.php*

```
<?php

// blog.blade.php

@extends('plantilla')

@section('apartado')
  <h2>Estás en BLOG</h2>
@endsection
```

Importamos el contenido de plantilla bajo la directiva @extends para que cargue los elementos estáticos que hemos declarado y con la directiva @section y @endsection definimos el bloque de código dinámico, en función de la sección que estemos visitando.

Ahora casi lo mismo para la sección de fotos

```php
<?php

// fotos.blade.html

@extends('plantilla')

@section('apartado')
  <h2>Estás en FOTOS</h2>
@endsection
```

El último paso que nos queda es configurar el archivo de rutas *routes/web.php*

```php
<?php

// web.php

Route::view('blog', 'blog') -> name('noticias');
Route::view('fotos', 'fotos') -> name('galeria');
```

Esto enlaza las URLs /blog y /fotos con las vistas blog.blade.php y fotos.blade.php, respectivamente. Los nombres de las rutas (noticias y galeria) se usan en el menú de navegación para hacerlas más claras.

De esta manera podremos hacer uso del menú de navegación que hemos puesto en nuestra plantilla (*plantilla.blade.php*) y gracias a los alias noticias y galeria, la URL será más amigable.

Nota:

- *route('noticias')* genera automáticamente la URL asociada al nombre de ruta noticias, que es */blog*.
- *route('galeria')* genera automáticamente la URL asociada al nombre de ruta galeria, que es */fotos*.

Estructuras **de** control

Como en todo buen lenguaje de programación, en Laravel también tenemos estructuras de control.

En Blade (plantillas de Laravel) siempre que iniciemos un bloque de estructura de control DEBEMOS cerrarla

- *@foreach* ~ *@endforeach* lo usamos para recorrer arrays
- *@if* ~ *@endif* para comprobar condiciones lógicas
- *@switch* ~ *@endswitch* en función del valor de una variable ejecutar un código
- *@case* define la casuística del switch
- *@break* rompe la ejecución del código en curso
- *@default* si ninguna casuística se cumple

```php
<?php

$equipo = ['María', 'Alfredo', 'William', 'Verónica'];

@foreach ($equipo as $nombre)
  <p> {{ $nombre }} </p>
@endforeach
```

Actividad 2:

Crea un sitio web con Laravel que contenga el título "Bienvenidos a Laravel", un texto de bienvenida (puede ser un poco de Lorem Ipsum) y a continuación un menú de navegación con sus correspondientes alias y los siguientes enlaces:

- *Inicio* enlace a la página principal donde se visualizará el texto de Lorem Ipsum además de los elementos estáticos (Título y menú de navegación).

- *Nosotros* enlace que vaya a la página "nosotros" y muestre, además de los elementos estáticos de todo el sitio, un h2 que diga "Estás en la sección Nosotros"

- *Proyecto* enlace que cargue una vista con el siguiente texto "Estás en el proyecto numero: X" donde X es un número entero que podamos introducirlo en la propia ruta. Si no se mete ningún número en la ruta, por defecto tiene que ser 1; por ejemplo

http://localhost:8000s/proyecto/210937

- Recuerda que el título y el menú de navegación han de aparecer en todas las vistas que cargues.

Nota: generador de Lorem Ipsum https://www.lipsum.com/

Controladores

Los controladores son el lugar perfecto para definir la lógica de negocio de nuestra aplicación o sitio web.

Hace de intermediario entre la vista (lo que vemos con nuestro navegador o cliente) y el servidor donde la app está alojada.

Por defecto, los controladores se guardan en una carpeta específica situada en `app/Http/Controllers` y tienen extensión .php.

Para crear un controlador nuevo debemos hacer uso de nuestro querido autómata artisan donde le diremos que cree un controlador con el nombre que nosotros queramos.

Abrimos la consola y nos situamos en la raíz de nuestro proyecto y ejecutamos el siguiente comando:

```
php artisan make:controller PagesController
```

Si todo ha salido bien, recibiremos un mensaje por consola con que todo ha ido bien y podremos comprobar que, efectivamente se ha creado el archivo PagesController.php con una estructura básica de controlador, dentro de la carpeta Controllers que hemos descrito anteriormente.

Ahora podemos modificar nuestro archivo de rutas web.pbp para dejarlo limpio de lógica y trasladar ésta a nuestro nuevo controlador.

La idea de ésto es dejar el archivo web.php tan limpio como podamos para que, de un vistazo, se entienda todo perfectamente.

RECUERDA que sólo movemos la lógica, mientras que las cláusulas como where y name las seguimos dejando en el archivo de rutas web.php

Veamos cómo quedaría un refactor del archivo de rutas utilizando un Controller como el que acabamos de crear (refactorizar implica mover la **lógica de negocio** (código complejo, como consultas a la base de datos) desde el archivo de rutas (routes/web.php) a un controlador).

Ahora nos quedaría de la siguiente manera

```php
<?php

// web.php (v2.0) ▒▒ Refactorizado

use App\Http\Controllers\PagesController;
use Illuminate\Support\Facades\Route;

Route::get('/', [ PagesController::class, 'inicio' ]);
Route::get('datos', [ PagesController::class, 'datos' ]);
Route::get('cliente/{id?}', [ PagesController::class, 'cliente' ]) -> where('id', '[0-9]+');
Route::get('nosotros/{nosotros?}', [ PagesController::class, 'nosotros']) -> name('nosotros');
```

y en nuestro archivo controlador lo dejaríamos de la siguiente manera

```php
<?php

// PagesController.php

namespace App\Http\Controllers;

class PagesController extends Controller
{
    public function inicio() { return view('welcome'); }

    public function datos() {
        return view('usuarios', ['id' => 56]);
    }

    public function cliente($id = 1) {
        return ('Cliente con el id: ' . $id);
    }

    public function nosotros($nombre = null) {
        $equipo = [
            'Paco',
            'Enrique',
            'Maria',
            'Veronica'
        ];

        return view('nosotros', compact('equipo', 'nombre'));
    }
}
```

Cómo funciona compact:

1. Recoge los nombres de las variables: En este caso, compact('equipo', 'nombre'), toma los nombres de las variables $equipo y $nombre como cadenas.
2. Crea una matriz asociativa: Devuelve una matriz asociativa donde las claves son los nombres de las variables y los valores son los contenidos de dichas variables.
    - Esto significa que compact('equipo', 'nombre') crea una matriz como: ['equipo' => ['Paco', 'Enrique', 'Maria', 'Veronica'], 'nombre' => $nombre].
3. Pasa la matriz a la vista: La matriz asociativa se pasa a la vista nosotros.

Actividad 3:

Utilizando las estructuras de control y los controladores crea un sitio web que contenga lo siguiente:

- *Inicio:* página principal con un título que diga "Bienvenid@s a FOTO BLOG" y un texto de bienvenida (con un par de frases sobra)

- *Nosotros* un título de la sección en la que te encuentras y un listado de 3 personas diferentes que, cada uno de los nombres mostrará un texto descriptivo de cada persona cuando pinchemos sobre su nombre.

- *Fotos* Una sección que aparezca el texto "Estás visualizando la foto con el ID: X" donde X es un parámetro que dependerá de la ruta que se haya introducido. Por defecto, si no se introduce dicho parámetro éste debe valer 1.

Como elementos estáticos debe aparecer:

- Imagen como logtipo

- El título FOTO BLOG a la derecha del logotipo

- Menú de navegación para moverse por las distintas secciones

Migraciones & Eloquent

Con las migraciones vamos a gestionar la base de datos de nuestro sitio web; tanto crear nuevas BBDD como editarlas desde Laravel.

Las migraciones de un sitio hecho con Laravel se alojan en la ruta database/migrations y tienen extensión .php.

Archivos .env

Es de uso común trabajar con archivos de entorno llamados también archivos *.env*. Normalmente, eun unproyecto real puedes encontrarte con varios archivos de este tipo en función del despliegue que se quiera hacer; como por ejemplo:

- *test.env* Usado para pruebas seguras.
- *release.env* Usado para cambios que serán enviados a un grupo de prueba (beta testers).
- *production.env* Usado cuando la aplicación está lista para ser utilizada por el público.

En nuestro caso, como no vamos a desplegar nada, sólo vamos a usar un único archivo: .env

Un ejemplo básico para configurar una conexión a una base de datos MySQL sería:

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

- DB_CONNECTION: Define el tipo de base de datos (en este caso, MySQL).
- DB_HOST: Especifica la dirección del servidor donde está alojada la base de datos (aquí, es el mismo equipo local: 127.0.0.1).
- DB_PORT: El puerto que usa MySQL (por defecto, es el 3306).
- DB_DATABASE: El nombre de la base de datos (en este caso, "blog").
- DB_USERNAME: El usuario que accede a la base de datos (aquí, el usuario "root").
- DB_PASSWORD: La contraseña para ese usuario (en este ejemplo, está en blanco).

En este archivo debemos configurar los datos de nuestro servidor MySQL y rellenarlo con la información correspondiente a nuestra base de datos ***ya creada***

Una vez tengamos ésto, lo que nos queda es ejecutar el comando de las migraciones a través del CLI artisan

Las **migraciones** son archivos de Laravel que contienen instrucciones para crear o modificar tablas en la base de datos. Son una forma organizada de manejar los cambios en las bases

de datos a lo largo del desarrollo. Veremos sus características básicas aunque los generaremos a partir de modelos en el siguiente apartado.

- Los archivos de migración están en **database/migrations/**.
- Sus nombres tienen un timestamp seguido de una descripción de la operación que realizan.
- Para crear un nuevo archivo de migración, usa:

```
php artisan make:migration <nombre_de_la_migracion>
```

- Los métodos up() y down() definen qué cambios realizar y cómo revertirlos.

Por ejemplo, al ejecutar:

```
php artisan make:migration create_users_table
```

Esto generará un archivo en la carpeta database/migrations con un nombre como *2025_01_23_123456_create_users_table.php*

Ejemplo básico de migración:

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateUsersTable extends Migration
{
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->id(); // Clave primaria
            $table->string('name'); // Columna 'name'
            $table->string('email')->unique(); // Columna 'email', única
            $table->timestamps(); // Columnas 'created_at' y 'updated_at'
        });
    }

    public function down()
    {
        Schema::dropIfExists('users'); // Eliminar tabla si existe
    }
}
```

A continuación, comentamos este código:

Importaciones de clases:

- **Migration**: Base para crear migraciones.
- **Blueprint**: Define la estructura de las tablas.
- **Schema**: Proporciona métodos para manipular tablas en la base de datos.

El método up() define lo que debe ocurrir al ejecutar la migración:

1. **Schema::create('users', function (Blueprint $table)**:
    - Crea una tabla llamada users.
2. **$table->id()**:
    - Crea una columna llamada id como clave primaria (tipo autoincremental).
3. **$table->string('name')**:
    - Crea una columna name de tipo VARCHAR para guardar el nombre del usuario.
4. **$table->string('email')->unique()**:
    - Crea una columna email de tipo VARCHAR y la marca como única para evitar duplicados.
5. **$table->timestamps()**:
    - Agrega dos columnas automáticas:
        - created_at: Fecha y hora de creación del registro.
        - updated_at: Fecha y hora de la última actualización del registro.

El método down():

Este método define lo que ocurre al revertir la migración:

1. **Schema::dropIfExists('users')**:
    - Elimina la tabla users si existe.

Ejecutar migraciones

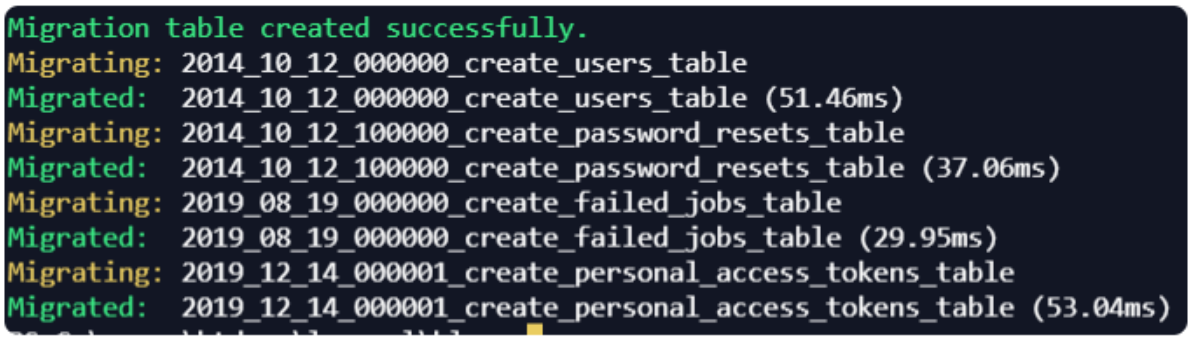
Comando para **ejecutar migraciones**:

```
php artisan migrate
```

Qué ocurre al ejecutar este comando?

- Laravel lee las instrucciones de los archivos de migración en la carpeta database/migrations.
- Crea las tablas correspondientes en la base de datos que configuraste en el archivo .env.
- Crea una tabla especial llamada migrations, que registra qué migraciones ya se ejecutaron.

Si todo ha salido bien obtendremos el siguiente resultado donde podremos observar que todas las migraciones se han insertado correctamente en la base de datos.



Puedes abrir phpMyAdmin y comprobar que las tablas se han creado.

Laravel permite deshacer cambios en la base de datos si algo sale mal o quieres realizar pruebas:

1. migrate:rollback: Revertir la última migración ejecutada.

   ```
   php artisan migrate:rollback
   ```

2. migrate:reset: Revertir todas las migraciones (deja la base de datos limpia).

   ```
   php artisan migrate:reset
   ```

Actividad 4:

Crear un sistema básico de gestión de cursos utilizando migraciones en Laravel. Tu tarea es generar y configurar las siguientes tablas en la base de datos:

Cursos (cursos)

id: Clave primaria.

nombre: Nombre del curso (máximo 100 caracteres).

descripcion: Descripción detallada del curso.

precio: Precio del curso (número decimal con dos decimales).

reated_at y updated_at: Tiempos automáticos generados por Laravel.

Estudiantes (estudiantes)

id: Clave primaria.

nombre: Nombre del estudiante (máximo 50 caracteres).

email: Correo electrónico único.

created_at y updated_at: Tiempos automáticos generados por Laravel.

Inscripciones (inscripciones)

id: Clave primaria.

curso_id: Relación con la tabla cursos.

estudiante_id: Relación con la tabla estudiantes.

fecha_inscripcion: Fecha en la que el estudiante se inscribió.

created_at y updated_at: Tiempos automáticos generados por Laravel.

Instrucciones

Genera las migraciones necesarias para crear las tablas descritas.

Asegúrate de configurar las claves foráneas en la tabla inscripciones para que se relacionen correctamente con las tablas cursos y estudiantes.

Ejecuta las migraciones y verifica que las tablas se hayan creado correctamente en la base de datos.

Modelos

Gracias a Eloquent y su integración con Laravel, podremos crear modelos de datos de una manera automatizada a través de artisan

Ahora que ya sabemos manejar las migraciones es hora de crear nuestras propias migraciones pero a través de Eloquent.

A través de la instrucción make:model creamos un nuevo modelo de datos, a continuación ponemos el nombre ***siempre empezando en Mayúsucla y en SINGULAR*** y pasamos el parámetro relacionado con las migraciones -m.

```
php artisan make:model Nota -m
```

make:model: Crea un modelo llamado Nota (nombre en singular y con mayúscula inicial)

-m: Crea automáticamente una migración asociada al modelo (create_notas_table).

Resultado:

- Se genera el archivo app/Models/Nota.php (el modelo).
- Se genera un archivo en database/migrations/ con un nombre del tipo 2022_01_07_81237_create_notas_table.php

Editar la migración: Abrimos el archivo generado en database/migrations y modificamos el esquema para añadir las columnas que queremos:

```php
Schema::create('notas', function (Blueprint $table) {
    $table->id(); // Clave primaria
    $table->timestamps(); // created_at y updated_at
    $table->string('nombre'); // Columna 'nombre' (tipo string)
    $table->text('descripcion'); // Columna 'descripcion' (tipo texto)
});
```

Hemos añadido las columnas nombre y descripción al esquema básico.

Ejecutar la migración: Una vez definida la estructura de la tabla, ejecutamos el comando:

```
php artisan migrate
```

Recuperando datos

Antes de mostrar datos en la web, primero necesitas que las tablas de tu base de datos contengan información. Para eso, puedes usar PHPMyAdmin.

Ahora tendremos que irnos a una vista ya creada o creamos una nueva y solicitamos los datos desde el HTML.

En el ejemplo anterior de PagesController.php tendríamos:

```php
<?php

// estamos en ▓▓ web.php

Route::get('notas', [ PagesController::class, 'notas' ]);
```

- **Route::get**: Define una ruta para solicitudes GET (es decir, URLs normales que un usuario visita en el navegador).
- **'notas'**: Es el segmento de la URL, por ejemplo, http://tu-app/notas.
- **[PagesController::class, 'notas']**: Indica que Laravel llamará al método notas() en el controlador PagesController.

Antes de intentar entrar, debemos configurar nuestro controlador de la siguiente manera:

```php
<?php

// estamos en ▓▓ PagesController.php

public function notas() {
  $notas = Nota::all();

  return view('notas', compact('notas'));
}
```

- **Nota::all()**: Usa Eloquent, el ORM de Laravel, para obtener todos los registros de la tabla notas.
- **return view()**: Devuelve una vista llamada notas.blade.php y le pasa la variable $notas, que contiene las notas obtenidas.

Esta vista se encarga de mostrar los datos en HTML.

```php
<?php

// estamos en ▓▓ notas.blade.php

<h1>Notas desde base de datos</h1>

<table border="1">
    <thead>
        <tr>
            <th>Nombre</th>
            <th>Descripción</th>
        </tr>
    </thead>

    @foreach ($notas as $nota)
        <tr>
            <td>{{$nota -> nombre}}</td>
            <td>{{$nota -> descripcion}}</td>
        </tr>
    @endforeach
</table>
```

- **@foreach ($notas as $nota)**: Recorre todas las notas obtenidas desde la base de datos.
- **{{ $nota->campo }}**: Imprime en pantalla el valor del campo correspondiente, como nombre o descripcion.

Hay que fijarse bien en los nombres de las columnas que tienen nuestras bases de datos, es justo lo que va después de -> y siempre rodeado por los símbolos {{ }} ya que estamos en un archivo de plantilla.

¿Qué pasaría si sólo queremos acceder a un único elemento? como si hiciésemos un SELECT * from usuarios where id = 1

Para éso, tenemos una instrucción específica en **Eloquent** que nos soluciona el problema. En este caso usaremos la instrucción **findOrFail** y como buenos usuarios de Laravel, lo utilizaremos dentro del controlador.

```php
<?php

// estamos en ▓▓ PagesController.php

public function detalle($id) {
  $nota = Nota::findOrFail($id);

  return view('notas.detalle', compact('nota'));
}
```

- **findOrFail($id)**: Busca un registro por su ID. Si no lo encuentra, lanza un error 404 automáticamente.
- **return view('notas.detalle', compact('nota'))**: Envía los datos de la nota a una nueva vista llamada detalle.blade.php.

Hay que acordarse que debemos configurar la ruta en nuestro archivo de rutas.

```php
<?php

// estamos en ▓▓ web.php

Route::get('notas/{id?}', [ PagesController::class, 'detalle' ]) -> name('notas.detalle');
```

Y por último, debemos crear la plantilla, pero como es un archivo de detalle o que está relacionado con otra plantilla ya creada, podemos crear una carpeta con el nombre de la plantilla y dentro, el archivo de plantilla en cuestión.

De tal manera que quedaría así resources/views/notas/detalle.blade.php

```php
<?php

// estamos en ▓▓ detalle.blade.php

@extends('plantilla')

@section('apartado')
    <h1>Detalle de la nota</h1>

    <h3>ID: {{ $nota -> id }}</h3>
    <h3>Nombre: {{ $nota -> nombre }}</h3>
    <h3>Descripción: {{ $nota -> descripcion }}</h3>
@endsection
```

Modificar tablas sin perder datos

Uno de los dilemas que tenemos a la hora de manejar las bases de datos con Laravel y Eloquent, es que a veces cometemos errores si queremos introducir una nueva columna dentro de nuestra tabla o modificar una de esas columnas ***SIN PERDER LOS DATOS DE LA BASE DE DATOS***.

Imaginemos que en nuestra tabla `notas` queremos agregar una columna con el nombre `autor`.

Lo primero de todo es crear una nueva migración para realizar este cambio. Para ello, haremos uso de artisan y debemos crear el nombre de esta migración con la siguiente fórmula: add_fields_to_ seguidamente del nombre de la tabla que queremos modificar.

```
php artisan make:migration add_fields_to_nota
```

Esto hace dos cosas:

- Crea un archivo de migración en la carpeta database/migrations/.
- Indica que esta migración modificará la tabla notas.

Seguidamente, nos metemos en el archivo de la migración que acabamos de crear y en el apartado up() debemos poner el cambio que queremos realizar y en la sección down() debemos hacer lo mismo pero tenemos que decirle a Eloquent que la elimine ¿por qué? para en caso de hacer migrate rollback, se cargue este nuevo campo que hemos creado.

```php
<?php

public function up()
{
  Schema::table('notas', function (Blueprint $table) {
      $table -> string('autor');
  });
}

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

Método **up():** Aquí defines los cambios que quieres realizar en la tabla. En este caso, quieres agregar una columna autor de tipo string

- **Schema::table('notas')**: Indica que se modificará la tabla notas.
- **$table->string('autor')**: Agrega una columna llamada autor de tipo string (texto).
- **->nullable()**: Permite que esta columna tenga valores NULL inicialmente, para evitar errores al insertar datos en registros existentes.

Método **down()**: Este método es el inverso del up(). Define cómo revertir los cambios en caso de que necesites deshacer esta migración (por ejemplo, usando php artisan migrate:rollback):

- **$table->dropColumn('autor')**: Elimina la columna autor de la tabla notas.

Es conveniente mantener actualizado el modelo app/Models/Nota.php para que nos funciones la operaciones de inserción y actualización de datos.

```php
<?php

namespace App\Models;

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

class Nota extends Model
{
    use HasFactory;

    // Define los campos que pueden ser asignados masivamente.
    // Esto es necesario para usar métodos como create() y update(),
    // que permiten insertar o actualizar registros en la base de dato
    protected $fillable = [
        'nombre',
        'descripcion',
    ];
}
```

Una vez editada la migración, ejecuta el comando para aplicar los cambios en tu base de datos: `php artisan migrate`

Formularios

Ahora que ya sabemos cómo cargar de una base de datos, vamos a ver cómo insertarlos con Laravel y sin escribir ni una sola línea de SQL.

Gracias al método save() de Laravel podremos guardar datos que provengan de un formulario desde nuestras plantillas. Para ello, lo primero que necesitamos lo siguiente:

- *formulario HTML* que recoja los datos que el usuario introduce
- Una *ruta* que sea la encargada de recibir los datos del formulario
- Método *POST* para enviar los datos al servidor
- Un método en nuestro *controlador* que procese los datos y los guarde a través de save()
- La cláusula de seguridad *@csrf* para evitar ataques desde otros sitios

Así pues, empecemos por el **formulario** (en una plantilla Blade dentro de resources/views)

```
<form action="{{ route('notas.crear') }}" method="POST">
    @csrf {{-- Cláusula para obtener un token de formulario al enviarlo --}}

    <input type="text" name="nombre" placeholder="Nombre de la nota" class="form-control mb-2" autofocus>
    <input type="text" name="descripcion" placeholder="Descripción de la nota" class="form-control mb-2">

    <button class="btn btn-primary btn-block" type="submit">
      Crear nueva nota
    </button>
</form>
```

Como vemos, creamos 2 inputs relacionados con nuestras columnas dentro de la tabla, en este caso nombre y descripción

El action del formulario debe apuntar a una nueva ruta que vayamos a crear y donde enviemos los datos mediante POST.

Ahora crearemos la ruta en nuestro archivo de rutas web.php

```
<?php

// estamos en ▓▓ web.php

Route::post('notas', [ PagesController::class, 'crear' ]) -> name('notas.crear');
```

Si nos fijamos, ya no estamos haciendo uso del get sino del método post y como son métodos diferentes, podemos nombrar la ruta de la misma manera que en get ya que no habrá conflicto.

- **Route::post**: Define que la ruta aceptará solicitudes POST.

- **notas**: Es la URL que se llamará desde el formulario.
- **PagesController::class**: Es el controlador que procesará la solicitud.
- **crear**: Es el método del controlador que ejecutará la lógica.
- **name('notas.crear')**: Da un nombre a la ruta (alias), que luego se usa en el action del formulario.

NOTA : necesitamos tener **dos rutas** para manejar formularios en Laravel:

- **Ruta GET** → Muestra el formulario en la vista.
- **Ruta POST** → Recibe los datos y los guarda en la base de datos.

```php
use App\Http\Controllers\PagesController;

// Ruta para mostrar el formulario (GET)
Route::get('/notas/crear', [PagesController::class, 'formulario'])->name('notas.formulario')

// Ruta para procesar el formulario (POST)
Route::post('/notas', [PagesController::class, 'crear'])->name('notas.crear');
```

Por otro lado, necesitamos invocar nuestro PagesControllerf y decirle que vamos a utilizar el método crear que todavía no existe pero que vamos a crear a continuación. No olvidemos crear un alias para poder vincularlo al actiondel formulario ***SUPER IMPORTANTE***.

Para terminar, editaremos nuetro archivo PagesController.php para que el controlador que estamos usando tenga el método que hemos nombrado previamente.

```php
<?php

// estamos en ▓▓ PagesController.php

use App\Models\Nota;
use Illuminate\Http\Request;

public function crear(Request $request) {
    $notaNueva = new Nota;

    $notaNueva -> nombre = $request -> nombre;
    $notaNueva -> descripcion = $request -> descripcion;

    $notaNueva -> save();

    return back() -> with('mensaje', 'Nota agregada exitósamente');
}
```

- **Request $request**: recibe un objeto llamado $request (el cual podemos cambiarle el nombre perfectamente) de tipo Request por lo que ***DEBEMOS IMPORTAR Request*** para poder utilizar dicha clase de Laravel
- **$request->nombre**: Accede al valor del campo name="nombre".
- **Nota**: Es el modelo de la tabla donde se guardan los datos.
- **save()**: Guarda el objeto en la base de datos automáticamente.
- **return back()**: Redirige al formulario anterior.
- **with('mensaje', '...')**: Envía un mensaje de éxito a la vista.

Pero ¿dónde va a salir este mensaje? -- lo tenemos que declarar en nuestra plantilla (la misma del formulario).

```
@if (session('mensaje'))
  <div class="mensaje-nota-creada">
      {{ session('mensaje') }}
  </div>
@endif
```

- **session('mensaje')**: Recupera el mensaje almacenado temporalmente en la sesión.

Validaciones

Laravel nos proporciona herramientas para poder validar los datos que el usuario introduce en los campos del formulario.

Además de poder hacerlo con la etiqueta required de HTML5, debemos validar los datos a través del Framework.

Para ello, necesitamos modificar varios elementos:

- En primer lugar, nuestro archivo controller
- En segundo lugar, nuestra plantilla que carga el formulario

Empecemos con el controlador. A través del método **validate()** le decimos a Eloquent qué campos son requeridos para poder enviar el formulario. Utilizaremos para ello un array asociativo con el nombre del input y la palabra reservada required

```
<?php

// estamos en ▓▓ PagesController.php

$request -> validate([
  'nombre' => 'required',
  'descripcion' => 'required'
]);
```

Seguidamente nos moveremos a la plantilla donde esté el formulario y a través de la directiva **@error** crearemos un bloque html con nuestro mensaje de error por cada uno de los inputs requeridos.

```
<?php

// estamos en ▓▓ notas.blade.php

@error('nombre')
    <div class="alert alert-danger">
      No olvides rellenar el nombre
    </div>
@enderror
```

Pero ¿qué pasa cuando ha habido un error y nos muestra el mensaje que hemos escrito? Si te fijas, los campos que habías rellenado perderán la información, pero con Laravel podemos persistirlos sin hacer que el usuario vuelva a introducirlos.

Para poder persistir los datos una vez enviados pero con algún error de campo requerido, utilizaremos la directiva **old()** como value del input dentro de nuestro formulario y le pasaremos el nombre del input declarado en la etiqueta name.

```php
<?php

// estamos en ▓▓ notas.blade.php

<input
  type="text"
  name="nombre"
  value="{{ old('nombre') }}"
  class="form-control mb-2"
  placeholder="Nombre de la nota"
  autofocus
>
```

Editando registros

Después de tener campos en la base de datos, lo interesante sería poder editarlos a través de un formulario.

Laravel nos proporciona las herramientas necesarias para ello; veamos pues lo que necesitamos para poder realizar el cambio a través de la directiva **put()**.

Para poder hacer el cambio de resgistros necesitamos lo siguiente:

- Un enlace para redirigir a la página de editar, pasando el id del elemento en cuestión
- Una nueva *ruta* que apunte a nuestra plantilla de editar
- Una *plantilla* para poder editar con un formulario que reciba los datos a editar
- Una nueva función dentro de nuestro *controlador* para poder manejar los datos ya introducidos
- Una nueva *ruta* que utilice el método *put()* para poder actualizar los datos
- Un nuevo método dentro de nuestro *controlador* para actualizar los datos nuevos introducidos

```php
<?php

// estamos en ▓▓ notas.blade.php

<a href="{{ route('notas.editar', $nota) }}" class="btn btn-warning btn-sm">
  Editar
</a>
```

Ahora creamos la ruta

```php
<?php

// estamos en ▓▓ wep.php

Route::get('editar/{id}', [ PagesController::class, 'editar' ]) -> name('notas.editar');
Route::put('editar/{id}', [ PagesController::class, 'actualizar' ]) -> name('notas.actualizar');
```

Ahora necesitamos crear una nueva plantilla resources/views/notas/editar.blade.php con el formulario pre-rellenado con los datos del registro a editar.

```php
<?php

// estamos en ▓▓▓ editar.blade.php
@extends('plantilla')

@section('apartado')
<h2>Editando la nota {{ $nota -> id }}</h2>

@if (session('mensaje'))
  <div class="alert alert-success">{{ session('mensaje')}}</div>
@endif

<form action="{{ route('notas.actualizar', $nota -> id) }}" method="POST">
  @method('PUT') {{-- Necesitamos cambiar al método PUT para editar --}}
  @csrf {{-- Cláusula para obtener un token de formulario al enviarlo --}}

  @error('nombre')
      <div class="alert alert-danger">
          El nombre es obligatorio
      </div>
  @enderror
  @error('descripcion')
      <div class="alert alert-danger">
          La descripción es obligatoria
      </div>
  @enderror

  <input
      type="text"
      name="nombre"
      class="form-control mb-2"
      value="{{ $nota -> nombre }}"
      placeholder="Nombre de la nota"
      autofocus
  >
  <input
      type="text"
      name="descripcion"
      placeholder="Descripción de la nota"
      class="form-control mb-2"
      value="{{ $nota -> descripcion }}"
  >

  <button class="btn btn-primary btn-block" type="submit">Guardar cambios</button>
</form>
@endsection
```

- **@method('PUT')**: Laravel requiere esta directiva para indicar que se usará el método PUT (necesario para actualizaciones).
- **@csrf**: Añade un token de seguridad para evitar ataques CSRF.
- **value="{{ $nota->nombre }}"**: Pre-rellena los campos con los valores actuales del registro.
- **Validaciones:** Muestra mensajes de error usando @error.

Y por último, modificamos nuestro PagesController

```php
<?php

// estamos en ▓▓ PagesController.php

public function editar($id) {
  $nota = Nota::findOrFail($id);

  return view('notas.editar', compact('nota'));
}

public function actualizar(Request $request, $id) {
  $request -> validate([
      'nombre' => 'required',
      'descripcion' => 'required'
  ]);

  $notaActualizar = Nota::findOrFail($id);

  $notaActualizar -> nombre = $request -> nombre;
  $notaActualizar -> descripcion = $request -> descripcion;

  $notaActualizar -> save();

  return back() -> with('mensaje', 'Nota actualizada');
}
```

Método editar

- Muestra el formulario de edición con los datos del registro a editar:
- **Nota::findOrFail($id)**: Busca el registro en la base de datos. Si no lo encuentra, devuelve un error 404 automáticamente.
- **compact('nota')**: Pasa la variable $nota a la vista.

Método actualizar

- Procesa los datos enviados desde el formulario y actualiza el registro:
- **Validación**: Verifica que los campos requeridos estén llenos antes de actualizar.
- **Actualización**: Modifica los valores del registro con los datos del formulario.
- **save()**: Guarda los cambios en la base de datos.
- **return back()**: Redirige al formulario de edición con un mensaje.

En la plantilla de edición (editar.blade.php), se muestra un mensaje si la actualización es exitosa:

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

Resumen del flujo

1. **Botón "Editar":** Llama a la ruta notas.editar con el ID del registro.
2. **Ruta notas.editar:** Llama al método editar en el controlador para mostrar el formulario.
3. **Formulario de edición:** Pre-rellena los campos con los datos actuales.
4. **Ruta notas.actualizar:** Procesa los datos enviados con el método PUT.
5. **Método actualizar:** Valida y guarda los nuevos datos en la base de datos.
6. **Mensaje de éxito:** Muestra un mensaje si la actualización se realizó correctamente.

Eliminando registros

A la hora de eliminar un registro nuevo, no necesitamos crear una plantilla nueva ya que podemos mandar la instrucción directamente a través de otro formulario.

Por lo tanto, para eliminar un registro de la base de datos utilizaremos lo siguiente.

- Un formulario básico con un único botón de eliminar
- Usaremos el método `DELETE` para sobreescribir el método del formulario HTML
- Una `ruta` nueva para controlar el `action` de este nuevo formulario
- Un nuevo método dentro de nuestro `Controlador` que lleve la lógica para borrar el registro

Vamos a ver cómo meter ese formulario dentro de nuestro listado de notas:

```php
<?php

// estamos en ▓▓ notas.blade.php

<form action="{{ route('notas.eliminar', $nota) }}" method="POST" class="d-inline">
  @method('DELETE')
  @csrf

  <button class="btn btn-danger btn-sm" type="submit">Eliminar</button>
</form>
```

Ahora que ya tenemos montado el formulario en nuestra plantilla, pasemos a crear la ruta que hemos colocado en el actiondel formulario para borrar elementos.

```php
<?php

// estamos en ▓▓ web.php

Route::delete('eliminar/{id}', [ PagesController::class, 'eliminar' ]) -> name('notas.eliminar');
```

El último paso que nos queda es modificar el PagesController

```php
<?php

// estamos en ▓▓ PagesController.php

public function eliminar($id) {
  $notaEliminar = Nota::findOrFail($id);
  $notaEliminar -> delete();

  return back() -> with('mensaje', 'Nota Eliminada');
}
```

Si todo ha salido bien, habremos creado un sitio en Laravel y Eloquent que es capaz de hacer un ***CRUD*** validando campos en formularios e insertando datos reales en una base de datos.

Paginación

Para añadir paginación a nuestros resultados, Eloquent tiene un método que se llama paginate() donde le pasamos un número entero como parámetro para indicarle el número de resultados que queremos por página.

```php
<?php

// estamos en ▓▓ PagesController.php

public function notas() {
  // $notas = Nota::all();
  $notas = Nota::paginate(5);

  return view('notas', compact('notas'));
}
```

Esto devuelve 5 resultados por página y, si se incluye `{{ $notas->links() }}` en la vista, genera automáticamente los enlaces de paginación.

Ahora veremos ciertos elementos HTML que se han generado en nuestra vista, ésto es porque Laravel hace uso de una librería de paginación situada en la carpeta `vendor/laravel/framework/src/illuminate/Pagination`

Si os metéis en el directorio y abrís el archivo `tailwind.blade.php` veréis la estructura HTML que os sale en la vista. Podéis modificar este archivo a vuestro antojo, pero es recomendable guardarse una copia del mismo.

Existe otra dependencia en `resources/lang/en/pagination.php` donde encontrarás el idioma para la paginación.

Autenticación

Para la autenticación de usuarios en Laravel, necesitamos instalar algunas dependencias específicas.

No es necesario crear un nuevo proyecto, pero en este caso lo haremos para tener uno con autenticación y otro sin ella (el que ya creamos al principio).

Crear un nuevo proyecto Laravel con autenticación

1. Creamos un nuevo proyecto llamado **notas_auth** y accedemos a su carpeta una vez se haya generado:

   ```
   laravel new notas_auth

   cd notas_auth
   ```

2. Dentro de la carpeta del proyecto, instalamos el paquete **laravel/ui**:

   ```
   composer require laravel/ui
   ```

3. Generamos la autenticación con Vue.js (crea automáticamente las tablas necesarias para el sistema de autenticación):

   ```
   php artisan ui vue --auth
   ```

4. Finalmente, ejecutamos la migración de la base de datos:

   ```
   php artisan migrate
   ```

NOTA: debes crear una nueva base de datos y posteriormente modificiar el archivo .env poniendo el nombre de esa base de datos que acabas de crear.

Si todo ha salido correctamente, en la carpeta resources/views aparecerá una nueva carpeta llamada **auth**, y también se habrá generado un nuevo controlador llamado **HomeController**.

Restringir una ruta

En el controlador **HomeController**, Laravel genera automáticamente un middleware que restringe el acceso a las rutas de ese controlador solo a usuarios autenticados.

```php
<?php
public function __construct()
{
    $this->middleware('auth');
}
```

El middleware `auth` obliga a que todas las rutas que usen este controlador requieran autenticación.
Por lo tanto, en nuestros proyectos es recomendable separar los controladores de vistas públicas y privadas.

NOTA: Un **middleware** en Laravel es un filtro que intercepta las solicitudes HTTP antes de que lleguen al controlador. Sirve para aplicar **lógica intermedia**. Laravel tiene muchos middlewares predefinidos, y están en app/Http/Middleware/. Algunos ejemplos:

`auth` Restringe el acceso a usuarios autenticados.

`guest` Permite acceso solo a usuarios NO autenticados.

`Verified` Requiere que el usuario haya verificado su correo electrónico.

Datos del usuario

Para acceder a los datos del usuario que ha iniciado sesión, utilizamos el helper auth(). Esto nos permite recuperar información del usuario y utilizarla en la lógica de nuestro código.

Imaginemos que tenemos una ruta donde accedemos a dicha información

```php
<?php

public function notas() {
  return auth()->user();

  // return auth()->user() -> name;
  // return auth()->user() -> email;
  // ...
}
```

Si visitamos esta ruta con nuestro login y password, nos aparecerá por pantalla toda la información de nuestro user a excepción de la contraseña y, aunque así fuera porque se lo forzamos, ésta aparecerá encriptada.

NOTA: Un **helper** en Laravel es una función global que se puede llamar en cualquier parte de la aplicación sin necesidad de importar clases o instanciar objetos.