

# 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 ejecutar 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:

```
1 <?php
2
3 // Ruta por defecto para cargar la vista welcome cuando el usuario introduce simplemente el dominio
4
5 Route::get('/', function () {
6     return view('welcome');
7 });
```

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 enrutador de Laravel.

```
1 <?php
2
3 Route::get('/users', function () {
4     return view('users');
5 }) -> 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.

```

1 <?php
2
3 /* http://localhost/datos/ */
4
5 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.

```

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

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

```

1 <?php
2
3 Route::get('cliente/{id}', function($id) {
4     return('Cliente con el id: ' . $id);
5 });
```

- 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 esto, 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.

```
1 <?php
2
3 Route::get('cliente/{id?}', function($id = 1) {
4     return 'Cliente con el id: ' . $id;
5 });
```

- `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 numérica.

```
1 <?php
2
3 Route::get('cliente/{id?}', function($id = 1) {
4     return 'Cliente con el id: ' . $id;
5 }) -> 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 utilizarlas 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

```
1 <?php
2
3 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

```
1 <!-- Estructura típica de un archivo HTML5 -->
2
3 <p>Usuario con id: <?= $id ?></p>
4
5 <!-- ... -->
```

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 nuestro contenido estático y dinámico.

```
1 <body>
2 <!-- CONTENIDO ESTÁTICO PARA TODAS LAS SECCIONES -->
3 <h1>Bienvenid@s a Laravel</h1>
4 <hr>
5
6 <!-- MENÚ -->
7 <nav>
8 <a href={{ route('noticias') }}>Blog</a> |
9 <a href={{ route('galeria') }}>Fotos</a>
10 </nav>
11
12 <!-- CONTENIDO DINÁMICO EN FUNCIÓN DE LA SECCIÓN QUE SE VISITA -->
13 <header>
14 @yield('apartado')
15 </header>
16 </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*

```
1 <?php
2
3 // blog.blade.php
4
5 @extends('plantilla')
6
7 @section('apartado')
8 <h2>Estás en BLOG</h2>
9 @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

```
1 <?php
2
3 // fotos.blade.html
4
5 @extends('plantilla')
6
7 @section('apartado')
8 <h2>Estás en FOTOS</h2>
9 @endsection
```

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

```
1 <?php
2
3 // web.php
4
5 Route::view('blog', 'blog') -> name('noticias');
6 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

```

1  <?php
2
3  $equipo = ['María', 'Alfredo', 'William', 'Verónica'];
4
5  @foreach ($equipo as $nombre)
6      <p> {{ $nombre }} </p>
7  @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.php` para dejarlo limpio de lógica y trasladar ésta a nuestro nuevo controlador.

La idea de esto 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

```
1  <?php
2
3  // web.php (v2.0) Refactorizado
4
5  use App\Http\Controllers\PagesController;
6  use Illuminate\Support\Facades\Route;
7
8  Route::get('/', [ PagesController::class, 'inicio' ]);
9  Route::get('datos', [ PagesController::class, 'datos' ]);
10 Route::get('cliente/{id?}', [ PagesController::class, 'cliente' ] -> where('id', '[0-9]+'));
11 Route::get('nosotros/{nosotros?}', [ PagesController::class, 'nosotros' ] -> name('nosotros'));
```

y en nuestro archivo controlador lo dejaríamos de la siguiente manera

```
1  <?php
2
3  // PagesController.php
4
5  namespace App\Http\Controllers;
6
7  class PagesController extends Controller
8  {
9      public function inicio() { return view('welcome'); }
10
11     public function datos() {
12         return view('usuarios', ['id' => 56]);
13     }
14
15     public function cliente($id = 1) {
16         return ('Cliente con el id: ' . $id);
17     }
18
19     public function nosotros($nombre = null) {
20         $equipo = [
21             'Paco',
22             'Enrique',
23             'Maria',
24             'Veronica'
25         ];
26
27         return view('nosotros', compact('equipo', 'nombre'));
28     }
29 }
```

### 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, en un proyecto 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:

```
1. <?php
2.
3. use Illuminate\Database\Migrations\Migration;
4. use Illuminate\Database\Schema\Blueprint;
5. use Illuminate\Support\Facades\Schema;
6.
7. class CreateUsersTable extends Migration
8. {
9.     public function up()
10.    {
11.        Schema::create('users', function (Blueprint $table) {
12.            $table->id(); // Clave primaria
13.            $table->string('name'); // Columna 'name'
14.            $table->string('email')->unique(); // Columna 'email', única
15.            $table->timestamps(); // Columnas 'created_at' y 'updated_at'
16.        });
17.    }
18.
19.    public function down()
20.    {
21.        Schema::dropIfExists('users'); // Eliminar tabla si existe
22.    }
23. }
```

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.

```
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table (51.46ms)
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table (37.06ms)
Migrating: 2019_08_19_000000_create_failed_jobs_table
Migrated: 2019_08_19_000000_create_failed_jobs_table (29.95ms)
Migrating: 2019_12_14_000001_create_personal_access_tokens_table
Migrated: 2019_12_14_000001_create_personal_access_tokens_table (53.04ms)
```

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).

created\_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 Mayúscula 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:

```
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 PHPMysqlAdmin.

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:

```
1 <?php
2
3 // estamos en web.php
4
5 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:

```
1 <?php
2
3 // estamos en PagesController.php
4
5 public function notas() {
6     $notas = Nota::all();
7
8     return view('notas', compact('notas'));
9 }
```

- **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.

```
1  <?php
2
3  // estamos en notas.blade.php
4
5  <h1>Notas desde base de datos</h1>
6
7  <table border="1">
8      <thead>
9          <tr>
10             <th>Nombre</th>
11             <th>Descripción</th>
12          </tr>
13      </thead>
14
15      @foreach ($notas as $nota)
16          <tr>
17              <td>{{ $nota -> nombre }}</td>
18              <td>{{ $nota -> descripcion }}</td>
19          </tr>
20      @endforeach
21  </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.

```
1  <?php
2
3  // estamos en PagesController.php
4
5  public function detalle($id) {
6      $nota = Nota::findOrFail($id);
7
8      return view('notas.detalle', compact('nota'));
9  }
```

- **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.

```
1 <?php
2
3 // estamos en web.php
4
5 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

```
1 <?php
2
3 // estamos en detalle.blade.php
4
5 @extends('plantilla')
6
7 @section('apartado')
8     <h1>Detalle de la nota</h1>
9
10     <h3>ID: {{ $nota -> id }}</h3>
11     <h3>Nombre: {{ $nota -> nombre }}</h3>
12     <h3>Descripción: {{ $nota -> descripcion }}</h3>
13 @endsection
```