

# Unidad 7

---

Framework Laravel 9  
Páginas web dinámicas

# ÍNDICE

1. Framework Laravel
2. Rutas Laravel
3. Controladores
4. Vistas
5. Base de datos en Laravel
6. CRUD
7. I

# 1. Framework Laravel

**Laravel** es un framework de código abierto para desarrollar aplicaciones a partir de PHP 5, evitando el código spaghetti.



<https://laravel.com/docs/9.x>

Un framework es un entorno de trabajo con código preescrito por una comunidad de programadores.

## Ventajas:

- Evita escribir código desde 0
- Buenas prácticas a la hora de programar
- Basado en el modelo MVC -> Código más ordenado

## 1.1 Instalación

1. Tener instalado Xammp
2. Instalar Composer <https://getcomposer.org>
3. Tenemos que usar una consola para ejecutar un comando, podríamos usar la consola de Windows pero vamos a usar Git Bash para ello, primero vamos a instalar Git <https://git-scm.com/download/win>
4. Abrimos la consola Git Bash. Escribimos: cd /c/xampp/htdocs (la url de donde irá nuestro proyecto) y le damos a enter.
5. Luego escribimos: composer global require laravel/installer y comenzará a instalarse.
6. Cuando se termine de instalar, limpiamos la terminal con clear
7. Para crear nuestro primer proyecto en Laravel ponemos: laravel new unidad7

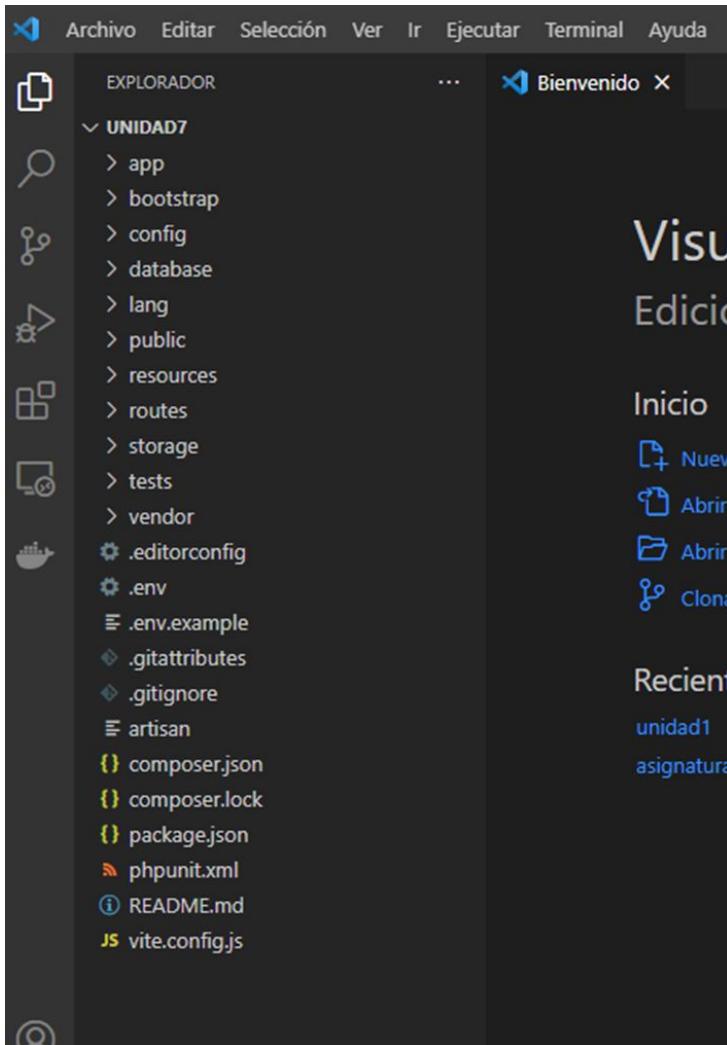
## 1.2 Primer proyecto

Esto lo que hace es crear en htdocs una carpeta que pondrá unidad7. Dentro de esta carpeta está toda la estructura de archivos que Laravel utiliza para poder trabajar.

Si queremos ver el contenido: localhost/unidad7 si queremos ver el contenido: public. Y nos muestra el contenido principal que tiene nuestro sitio web.

**Abrimos el Visual:** archivo-> abrir carpeta->htdocs-> unidad7

Si quiero modificar algo de lo que sale en public, nos dirigimos a la carpeta resources>views>welcomeblade.php y por ejemplo, si cambiamos Laravel por Unidad 7, en la pestaña en lugar de salir Laravel pondrá Unidad 7.



## 2. Rutas en Laravel

Laravel se basa en el patrón **Front Controller**. Un único punto de entrada a nuestra aplicación: que será: public -> index.php

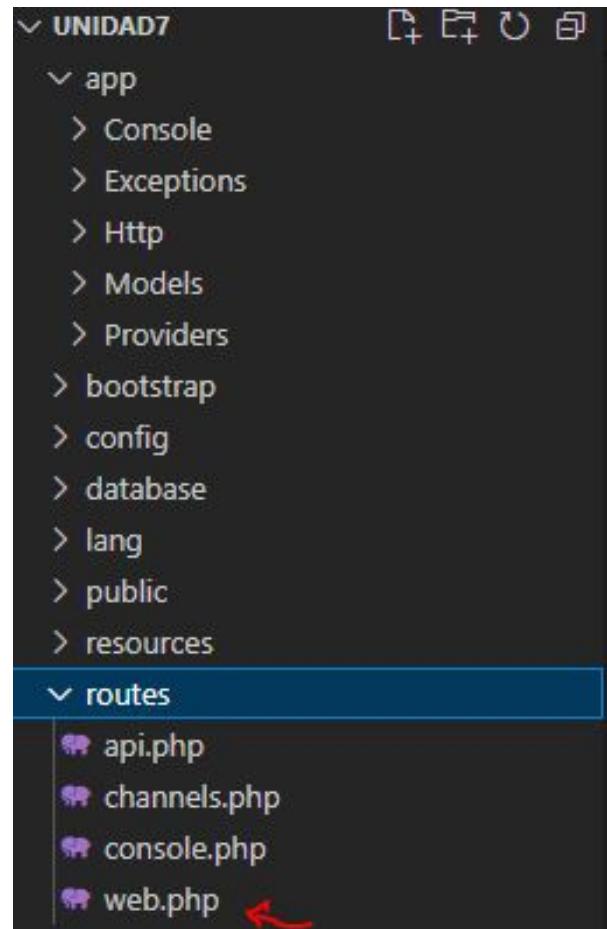
Los únicos archivos a los que el usuario va a poder acceder serán los de esta carpeta.

Se hace de esta manera por un motivo de seguridad. No vamos a querer que nadie acceda a la carpeta app porque va a ser el corazón de nuestra aplicación. Ni tampoco config.

Si solo tenemos un punto de entrada a nuestra aplicación que es nuestro archivo index.php, Laravel cómo sabría cómo actuar cuando un usuario escribe una url. Para eso vamos a ver las **rutas**.

El archivo que se encarga de ver las rutas es:

Routes -> web.php



## 2. Rutas en Laravel

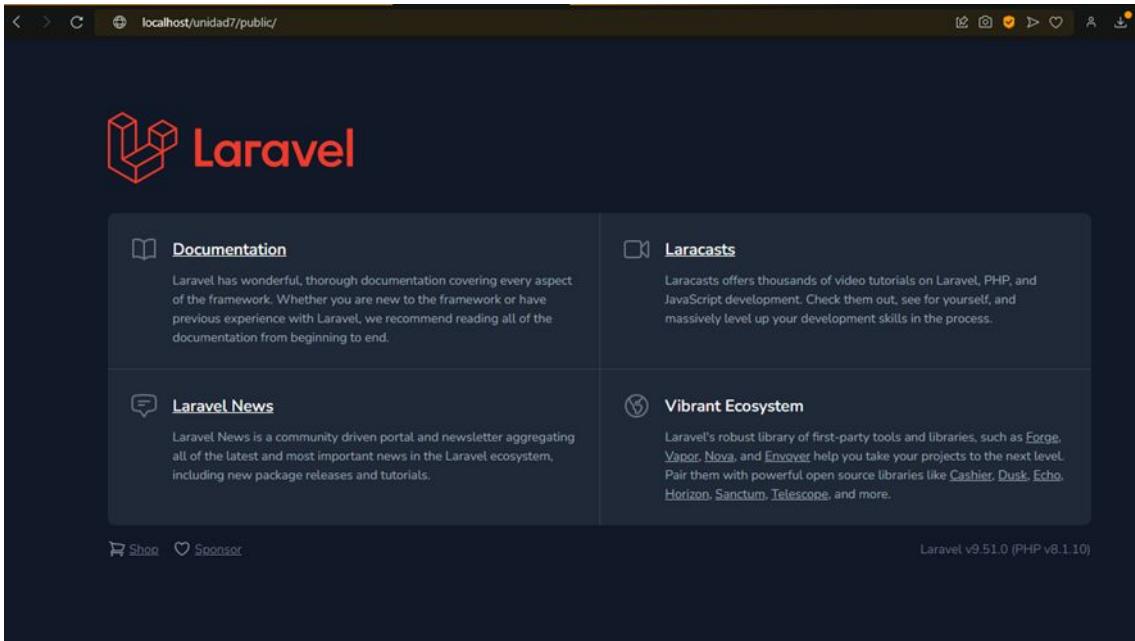
```
1 <?php
2
3 use Illuminate\Support\Facades\Route;
4
5 /*
6 |-----|
7 | Web Routes
8 |-----|
9 |
10| Here is where you can register web routes for your application. These
11| routes are loaded by the RouteServiceProvider within a group which
12| contains the "web" middleware group. Now create something great!
13|
14*/
15
16 Route::get('/', function () {
17     return view('welcome');
18 });
19 |
```

Cuando un usuario escriba una url lo que va a hacer Laravel es verificar si esa URL la hemos definido en este archivo. / Que será la página principal de mi sitio web.

Para comprobarlo:  
localhost/unidad7/public

El código significa: cuando un usuario entre al sitio principal, devuelves una vista. La vista Welcome.

## 2. Rutas en Laravel



Cuando un usuario escriba una url  
lo que va a hacer Laravel es  
verificar si esa URL la hemos  
definido en este archivo. /  
Que será la página principal de mi  
sitio web.

Para comprobarlo:  
`localhost/unidad7/public`

El código significa: cuando un  
usuario entre al sitio principal,  
devuelves una vista. La vista  
Welcome.

## 2. Rutas en Laravel

Como aún no sabemos lo que es un view, vamos a comentar la línea del return y ponemos:  
return "Bienvenidos a la página principal"

< > C : localhost/unidad7/public/

Bienvenidos a la página principal

```
1  <?php
2
3  use Illuminate\Support\Facades\Route;
4
5  /*
6  |-----
7  | Web Routes
8  |-----
9  |
10 | Here is where you can register web routes for your application. These
11 | routes are loaded by the RouteServiceProvider within a group which
12 | contains the "web" middleware group. Now create something great!
13 |
14 */
15
16 ~ Route::get('/', function () {
17 |     //return view('welcome');
18 |     return "Bienvenidos a la página principal";
19 |});
```

## 2. Rutas en Laravel

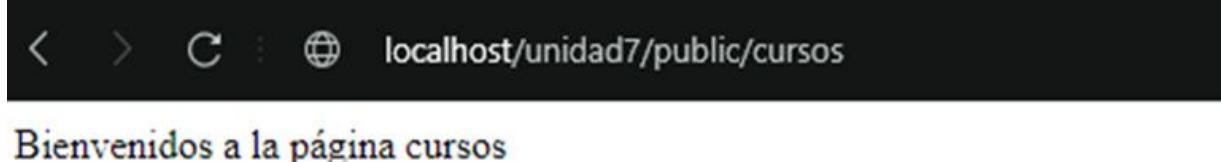
Imaginemos ahora que estamos desarrollando una página web de cursos. Necesitamos una URL para que nos muestre todo el listado de los cursos.

Para definir otra ruta:

```
15
16  ↘ Route::get('/', function () {
17      //return view('welcome');
18      return "Bienvenidos a la página principal";
19  });
20
21  ↘ Route::get('cursos', function(){
22      return "Bienvenidos a la página cursos";
23  });|
```

Si vamos a probarla:

localhost/unidad7/public/cursos



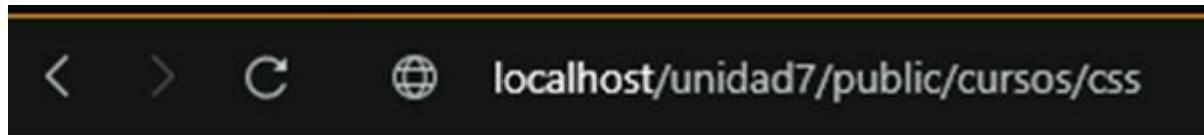
## 2. Rutas en Laravel

Muchas veces, en cursos, vamos a necesitar que haya más páginas que den acceso a los cursos.  
Cómo hacemos esto:

```
15
16 Route::get('/', function () {
17     //return view('welcome');
18     return "Bienvenidos a la página principal";
19 });
20
21 Route::get('cursos', function(){
22     return "Bienvenidos a la página cursos";
23 });
24
25 Route::get('cursos/{curso}', function($curso){
26     return "Bienvenido al curso: $curso";
27 });
```

## 2. Rutas en Laravel

Y si lo probamos: localhost/unidad7/public/cursos/css



Bienvenido al curso: css

Esta información la saca de la URL.

Si pusiéramos HTML en lugar de CSS pues saldría HTML.

Esta tarea de crear URL es bastante repetitiva, por eso existen extensiones que hacen la tarea más amena: **Laravel snippets**

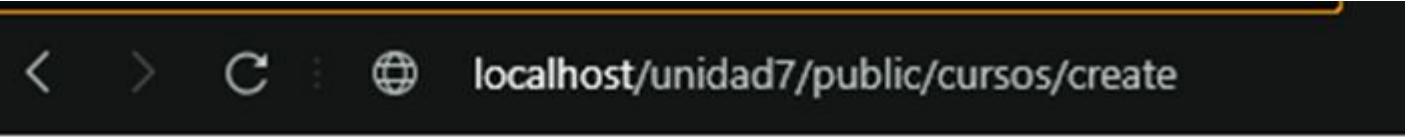
Lo que hace es darte opciones para autocompletar.

## 2. Rutas en Laravel

Imaginemos ahora que queremos crear una ruta con un formulario para poder crear un curso. En este caso, lo que hacemos es:

```
Route::get('cursos/create', function(){
    return "En esta página podrás crear un curso";
});
```

Y si lo probamos:



localhost/unidad7/public/cursos/create

Bienvenido al curso: create

## 2. Rutas en Laravel

Pero esto no es lo que queríamos. Nosotros esperábamos un

“En esta página podrás crear un curso”.

Entonces, ¿por qué ocurre esto? Porque Laravel **lee de arriba abajo**.

Y en la anterior Route habíamos puesto una variable {}, entonces Laravel lo que entiende es que creáte es una variable. Nunca llegando a leer la última ruta.

Cómo solucionamos esto:  
**cambiando el orden de la ruta.**

```
Route::get('/', function () {
    //return view('welcome');
    return "Bienvenidos a la página principal";
});

Route::get('cursos', function(){
    return "Bienvenidos a la página cursos";
});

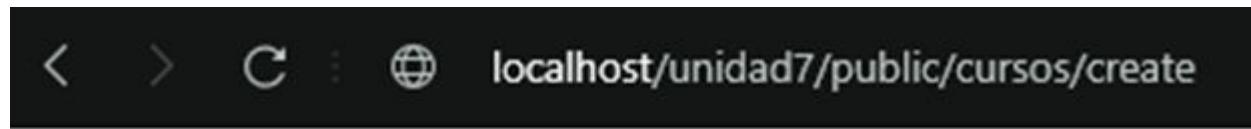
//Si un usuario escribiera /cursos/css

Route::get('cursos/create', function(){
    return "En esta página podrás crear un curso";
});

Route::get('cursos/{curso}', function($curso){
    return "Bienvenido al curso: $curso";
});
```

## 2. Rutas en Laravel

Y si lo probamos: localhost/unidad7/public/cursos/create



En esta página podrás crear un curso

## 2. Rutas en Laravel

También podremos pasar más de una variable en una url. Por ejemplo:

```
35 Route::get('cursos/{curso}/{categoría}', function($curso, $categoría){  
36     |     return "Bienvenido al curso: $curso, de la categoría $categoría";  
37 });|
```

Y si lo probamos:



## 2. Rutas en Laravel

Pero qué pasa, que al final en el archivo rutas tenemos un montón de código y no queda nada limpio. Entonces, ¿cómo hacemos para no tener tantas rutas?

```
31  /*Route::get('cursos/{curso}', function($curso){
32  |     return "Bienvenido al curso: $curso";
33  });*/
34
35  Route::get('cursos/{curso}/{categoría?}', function($curso, $categoría = null){
36  |     return "Bienvenido al curso: $curso, de la categoría $categoría";
37  });|
```

Ponemos en categoría ? y en function el valor null.

Qué va a ocurrir, pues si yo no le paso el valor de la categoría por la url, automáticamente va a ser null. Mientras que si lo se lo paso, tomará el valor de la URL.

## 2. Rutas en Laravel

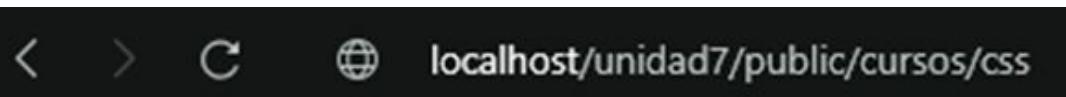
Entonces podríamos poner:

```
30
31     /*Route::get('cursos/{curso}', function($curso){
32         return "Bienvenido al curso: $curso";
33    });*/
34
35     Route::get('cursos/{curso}/{categoría?}', function($curso, $categoría = null){
36         if($categoría){
37             return "Bienvenido al curso: $curso, de la categoría $categoría";
38         }else{
39             return "Bienvenido al curso: $curso";
40         }
41     });

```

## 2. Rutas en Laravel

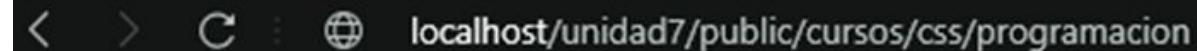
Y si lo probamos:



Poniendo solo el curso:

Bienvenido al curso: css

Y poniéndolo todo:



Bienvenido al curso: css, de la categoría programacion

Pero seguimos “ensuciando mucho” nuestro archivo web porque la idea es que en web.php **SOLO** estén las URL y que la lógica esté derivada en un **controlador**.

Es lo que vamos a ver ahora.

### 3. Controladores

Como ya hemos dicho antes, Laravel se basa en un patrón de diseño **modelo vista controlador**. (**MVC**). Consiste en que debemos separar nuestros archivos HTML (Vista) del código php (Modelo). La idea es separar lo más posible los diferentes lenguajes.



Entonces teniendo separado estos dos lenguajes, la pregunta es:

¿Cómo vamos a enlazarlo?

Ahí es donde entran en juego los **controladores**.

### 3. Controladores

Los controladores se crean en app>Http>Controllers .

Se puede:

- Crearlo manualmente y escribir la estructura del código manualmente
- O crearlo desde la terminal con un código

Para **crear cualquier elemento** en Laravel será con el comando: `php artisan make`

En particular, para crear un controlador, por ejemplo de la ruta principal:

```
Route::get('/', function () {
    //return view('welcome');
    return "Bienvenidos a la página principal";
});
```

Creamos el controlador Home y siempre se pone al final Controller. Escribimos entonces:



IMPORTANTE  
MEMORIZAR

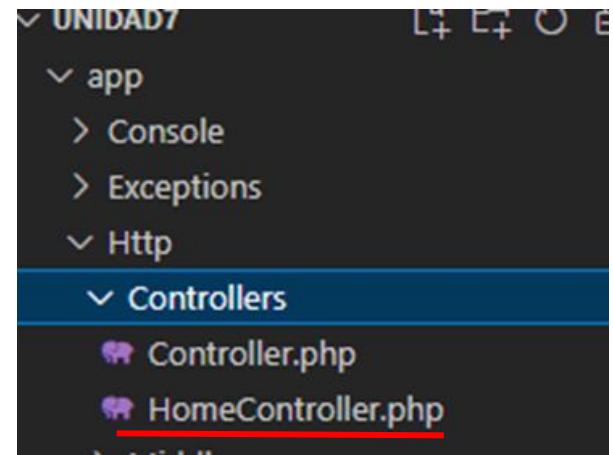
### 3. Controladores

Crear controlador: `php artisan make:controller HomeController`

Si nos vamos a nuestro proyecto en el Visual:  
app>Http>Controllers

Aparecerá el controlador que hemos creado.

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use Illuminate\Http\Request; //Lo veremos más adelante
6
7 class HomeController extends Controller
8 {
9
10 }
11 |
```



HomeController.php

### 3. Controladores

#### Asignar ruta al controlador:

Vamos a ver ahora cómo una vez creado nuestro controlador, le asignamos la ruta que queremos.

Tenemos que irnos a web.php y poner debajo del use:

Una vez hagamos esto, tenemos que irnos a donde tenemos los Route y modificarlo de la siguiente manera:

```
Route::get('/', function () {
    //return view('welcome');
    return "Bienvenidos a la página principal";
});
```

```
routes > web.php > ...
1  <?php
2
3  use Illuminate\Support\Facades\Route;
4  use App\Http\Controllers\HomeController;
5
6  /*
16
17  Route::get('/', HomeController::class);
```

Y lo que teníamos puesto, lo pegamos dentro de un método de la clase HomeController de antes:

### 3. Controladores

#### Asignar ruta al controlador:

```
Route::get('/', function () {
    //return view('welcome');
    return "Bienvenidos a la página principal";
});
```

Ya tendríamos nuestro primer controlador que administra la página principal

```
app > Http > Controllers > HomeController.php > ...
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request; //Lo veremos más adelante
6
7  class HomeController extends Controller
8  {
9      public function __invoke(){
10         //return view('welcome');
11         return "Bienvenidos a la página principal";
12     }
13 }
14 |
```

### 3. Controladores

#### Controladores que administran un mismo concepto

Volvemos a web.php y vamos a comentar y descomentar:

```
17 Route::get('/', HomeController::class);
18
19
20 Route::get('cursos', function(){
|    return "Bienvenidos a la página cursos";
21 });
22
23
24
25 Route::get('cursos/create', function(){
26     return "En esta página podrás crear un curso";
27 });
28
29 Route::get('cursos/{curso}', function($curso){
30     return "Bienvenido al curso: $curso";
31 });
32
33 /*Route::get('cursos/{curso}/{categoría?}', function($curso, $categoría = null){
34     if($categoría){
35         return "Bienvenido al curso: $curso, de la categoría $categoría";
36     }else{
37         return "Bienvenido al curso: $curso";
38     }
39}); */
```

Nos fijamos en lo siguiente:  
tenemos 3 rutas (Route)

Una url que administra la página principal de cursos, otra de cursos/create y otra ruta que lo que debería hacer es mostrar un curso.

Estas 3 rutas **tienen el mismo concepto**.

En lugar de crear un controlador para cada ruta, creamos uno que se encargue de administrar estas 3 rutas.

### 3. Controladores

#### Controladores que administran un mismo concepto

Escribimos php artisan make:controller CursoController

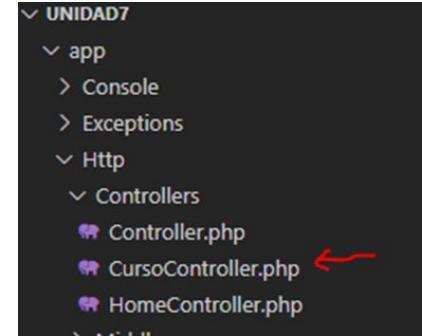
Como este controlador va a administrar 3 rutas diferentes, no vamos a usar el método invoke de antes.

Si no que vamos a crear 3 métodos diferentes, que le vamos a poner el nombre que queremos, pero por convención:

index

create

show



```
app > Http > Controllers > CursoController.php > CursoController > index
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6
7  class CursoController extends Controller
8  {
9      public function index(){
10
11
12
13      public function create(){
14
15
16
17      public function show(){
18
19
20
21 }
```

### 3. Controladores

#### Controladores que administran un mismo concepto

Y vamos a copiar dentro lo que teníamos en cada una de web.php



Quedando:

```
app > Http > Controllers > CursoController.php > CursoController
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6
7  class CursoController extends Controller
8  {
9      public function index(){ //Hace referencia a cursos
10         return "Bienvenidos a la página cursos";
11     }
12
13
14     public function create(){ //Hace referencia a cursos/create
15         return "En esta página podrás crear un curso";
16     }
17
18
19     public function show($curso){ //Hace referencia a cursos/muestra
20         return "Bienvenido al curso: $curso";
21     }
22 }
```

```
Route::get('cursos', function(){
    return "Bienvenidos a la página cursos";
});

Route::get('cursos/create', function(){
    return "En esta página podrás crear un curso";
});

Route::get('cursos/{curso}', function($curso){
    return "Bienvenido al curso: $curso";
});
```

### 3. Controladores

#### Controladores que administran un mismo concepto

Nos vamos de nuevo a web.php y tenemos que escribir en las primeras líneas:

```
1 <?php
2
3 use Illuminate\Support\Facades\Route;
4 use App\Http\Controllers\HomeController;
5 use App\Http\Controllers\CursoController; |
6
```

E igual que antes modificamos cada Route de la siguiente manera, creando un array porque si no lo creamos y ponemos CursoController::class estaría buscando el invoke de antes pero **no hay** invoke.

```
17
18 Route::get('/', HomeController::class);
19 |
20 Route::get('cursos', [CursoController::class, 'index']);
21
22 Route::get('cursos/create', [CursoController::class, 'create']);
23
24 Route::get('cursos/{curso}', [CursoController::class, 'show']);
25
```

Quedando una estructura de rutas mucho más limpia

### 3. Controladores

#### Grupo de rutas por controlador

**Novedad Laravel 9** – Para que aún quede más limpio el código, podemos crear un **grupo de rutas** para que las rutas de las líneas 20, 22 y 24 sean llamadas desde una única ruta. Cómo lo hacemos:

```
17
18     Route::get('/', HomeController::class);
19
20     Route::controller(CursoController::class)->group(function(){
21         Route::get('cursos', 'index');
22         Route::get('cursos/create', 'create');
23         Route::get('cursos/{curso}', 'show');
24
25     });
26
```

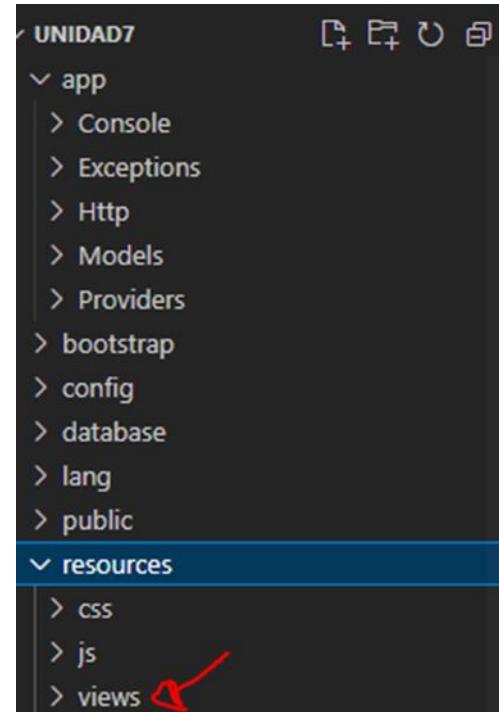
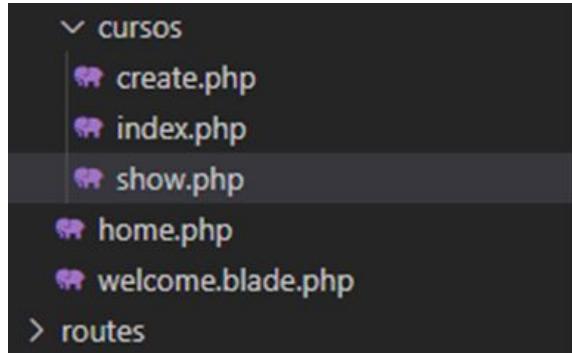
## 4. Vistas

Vamos a ver cómo hacer para que no solo se nos muestre una cadena de texto.  
Si no un documento HTML, a estos documentos se les llama **Vistas**.

Estas vistas están en: resources>views

Ahí creo un documento que se llame home.php

Y como en nuestro fichero de rutas, hemos creado una dirección que sea cursos, dentro de view creo una carpeta que se llame cursos y dentro, una vista con el mismo nombre de cada método (index, create, show):



## 4. Vistas

Dentro de cada uno, creamos un documento HTML (Si pulsamos ! se crea solo)

Abrimos nuestro HomeController y el texto del return, lo pegamos en un <h1></h1> en nuestro home.php

```
resources > views > 🗂 home.php > ⚙ html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Document</title>
8  </head>
9  <body>
10     <h1>Bienvenidos a la página principal</h1>
11 </body>
12 </html>
```

```
app > Http > Controllers > 🗂 HomeController.php > 🛡 HomeController > ⚙ __invoke
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request; //Lo veremos más adelante
6
7  class HomeController extends Controller
8  {
9      public function __invoke()
10         return view('home');
11
12     }
13 }
```

Y en HomeController ponemos  
return view('home');

## 4. Vistas

Y hacemos el mismo procedimiento con `CursoController`.

Observación, en `show.php` tendríamos que poner:

```
sources > views > cursos > show.php > html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Document</title>
8  </head>
9  <body>
10     <h1>Bienvenido al curso: <?php echo $curso ?> </h1>
11 </body>
12 </html>
```

## 4. Vistas

Quedaría:

### Observación

Ponemos  
cursos.index,  
cursos.create,  
cursos.show  
porque forman parte  
de la carpeta cursos  
dentro de las views.

```
6
7 class CursoController extends Controller
8 {
9     public function index(){ //Hace referencia a cursos
10        return view('cursos.index');
11    }
12
13
14    public function create(){ //Hace referencia a cursos/create
15        return view('cursos.create');
16    }
17
18
19    public function show($curso){ //Hace referencia a cursos/muestra
20        return view('cursos.show');
21    }
22
23
24
```

## 4. Vistas

Si lo probamos:

< > C : localhost/unidad7/public/

# Bienvenidos a la página principal

< > C : localhost/unidad7/public/cursos

# Bienvenidos a la página cursos

< > C : localhost/unidad7/public/cursos/create

# En esta página podrás crear un curso

## 4. Vistas

Pero veamos qué pasa si escribimos: localhost/unidad7/public/cursos/css

The screenshot shows a browser window with the URL `localhost/unidad7/public/cursos/css`. The page displays an `ErrorException` with the message `Undefined variable $curso`. Below the error message, the stack trace and the source code are shown.

`Stack` `Context` `Share`

`ErrorException`  
PHP 8.0.12

`Undefined variable $curso`

`Collapse vendor frames`

`Illuminate\Foundation\Bootstrap\HandleExceptions::handleError`

`Illuminate\Foundation\Bootstrap\HandleExceptions::266`

`Illuminate\Foundation\Bootstrap\{closure}`

`C:\xampp\htdocs\unidad7\resources\views\cursos\show.php:10`

`require`

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Document</title>
8  </head>
9  <body>
10     <h1>Bienvenido al curso: <?php echo $curso ?> </h1>
11  </body>
```

## 4. Vistas

Nos sale un error. Esto ocurre porque no lo hemos pasado en la vista, cómo se hace esto:

```
app > Http > Controllers > CursoController.php > CursoController
1
2
3
4
5     use Illuminate\Http\Request;
6
7     class CursoController extends Controller
8     {
9         public function index(){ //Hace referencia a cursos
10            return view('cursos.index');
11        }
12    }
13
14    public function create(){ //Hace referencia a cursos/create
15        return view('cursos.create');
16    }
17
18    public function show($curso){ //Hace referencia a cursos/muestra
19        return view('cursos.show', ['curso'=> $curso]);
20
21    }
22}
23
```

## 4. Vistas

Si lo comprobamos:



¡Funciona!

Esto se puede hacer de **otra manera**: usando el método `compact`

```
public function show($curso){ //Hace referencia a cursos/muestra
    return view('cursos.show', compact('curso'));
}
```

## 4. Vistas

Si nos fijamos: tanto la vista home, index, create, show comparten la **misma estructura de HTML**

Imaginemos que tenemos:

```
resources > views > 📄 home.php > ⚙️ html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1.0">
7      <title>Document</title>
8      <!-- favicon -->
9      <!-- estilos -->
10     </head>
11     <body>
12         <!-- header -->
13         <!-- nav -->
14         <h1>Bienvenidos a la página principal</h1>
15         <!-- footer -->
16         <!-- script -->
17     </body>
18 </html>
```

Imaginemos que queremos cambiar el favicon y el nav de tus páginas, tal y como lo tenemos, tendríamos que ir página por página cambiándolo manualmente.

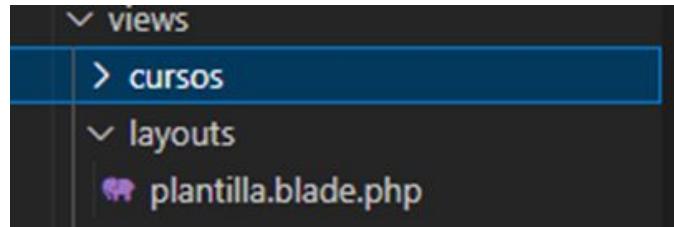
Si el sitio web es pequeño, no hay problema pero si el sitio web es más grande se complica la cosa.

## 4. Vistas

Laravel nos proporciona un sistema de **plantillas de Blade**.

Cómo se utiliza esto: dentro de la carpeta view, creamos una nueva carpeta: layouts.

Dentro de esta carpeta, plantilla.blade.php es importante que tenga .blade.



Copiamos lo que tenemos en home.php y borramos lo que queremos que sea distinto de la siguiente manera:

```
resources > views > layouts > 🐾 plantilla.blade.php > ⚙ html
 1  <!DOCTYPE html>
 2  <html lang="en">
 3  <head>
 4    <meta charset="UTF-8">
 5    <meta http-equiv="X-UA-Compatible" content="IE=edge">
 6    <meta name="viewport" content="width=device-width, initial-scale=1.0">
 7    <title>@yield('title')</title> ←
 8    <!-- favicon -->
 9    <!-- estilos -->
10  </head>
11  <body>
12    <!-- header -->
13    <!-- nav -->
14    <!-- content --> ←
15    <!-- footer -->
16    <!-- script -->
17  </body>
18  </html>
```

## 4. Vistas

Y ahora, a las vistas que quiero que hereden de esto:

Por ejemplo, en `home.php`, pondré:

### Observación

`@extends`,  
`@section`,  
`@endsection`  
son directivas de Blade pero  
nosotros tenemos un documento  
.php y no lo va a entender.

Para que lo entienda tenemos que  
cambiarle el nombre de `home.php`  
a [home.blade.php](#)

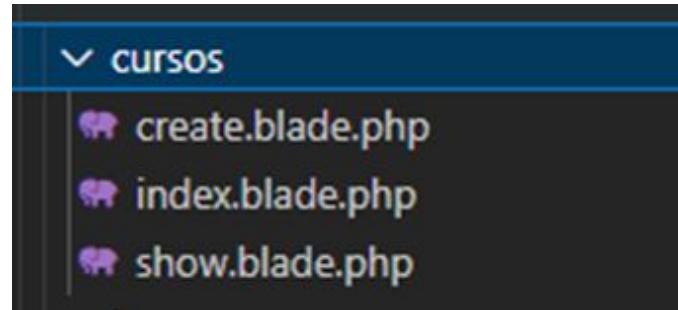
```
resources > views > home.php > h1
1   @extends('layouts.plantilla')
2
3   @section('title', 'Home')
4
5   @section('content') <!-- content porque fue el nombre que puse en la plantilla -->
6
7       <h1>Bienvenidos a la página principal</h1>
8
9   @endsection
10
11  <!DOCTYPE html>
12  <html lang="en">
13      <head>
14          <meta charset="UTF-8">
15          <meta http-equiv="X-UA-Compatible" content="IE=edge">
16          <meta name="viewport" content="width=device-width, initial-scale=1.0">
17          <title>Document</title>
18          <!-- favicon -->
19          <!-- estilos -->
20      </head>
21      <body>
22          <!-- header -->
23          <!-- nav -->
24
25          <!-- footer -->
26          <!-- script -->
27      </body>
28  </html>
```

## 4. Vistas

Y ya podríamos eliminar todo el HTML del home, quedando:

```
resources > views > 🐾 home.blade.php > ...
1   @extends('layouts.plantilla')
2
3   @section('title', 'Home')
4
5   @section('content') <!-- content porque fue el nombre que puse en la plantilla -->
6
7       <h1>Bienvenidos a la página principal</h1>
8
9   @endsection
10  |
```

Vamos a hacer lo mismo con todo lo demás:



## 4. Vistas

Antes de continuar vamos a añadir una extensión más a Visual: [Laravel Blade Snippets](#) que nos ayudará con las directrices de Blade.

Ahora nos saldrán:

```
resources > views > home.blade.php > ...
1 @extends('layouts.plantilla')
2
3 @section('title', 'Home')
4
5 @section('content') <!-- content porque fue el nombre que puse en la plantilla -->
6
7     <h1>Bienvenidos a la página principal</h1>
8
9 @endsection
10 |
```

Y no todo en blanco como antes. Y si vamos a cada uno de los documentos, se podrá autocompletar con cada una de las directivas

## 4. Vistas

Entonces, haciendo lo de antes: create.blade, index.blade, show.blade nos quedaría:

create.blade

```
resources > views > cursos > 📄 create.blade.php > ...
1   @extends('layouts.plantilla')
2
3   @section('title','Cursos create')
4
5   @section('content') <!-- content porque fue el nombre que puse en la plantilla -->
6
7       <h1>En esta página podrás crear un curso</h1>
8
9   @endsection
10  |
```

## 4. Vistas

Entonces, haciendo lo de antes: create.blade, index.blade, show.blade nos quedaría:

index.blade

```
resources > views > cursos > index.blade.php > ...
1  @extends('layouts.plantilla')
2
3  @section('title','Cursos')
4
5  @section('content') <!-- content porque fue el nombre que puse en la plantilla -->
6
7      <h1>Bienvenidos a la página cursos</h1>
8
9  @endsection
10
```

## 4. Vistas

Entonces, haciendo lo de antes: create.blade, index.blade, show.blade nos quedaría:

show.blade

```
resources > views > cursos > 📄 show.blade.php > ...
1  @extends('layouts.plantilla')
2
3  @section('title','Curso' . $curso)
4
5  @section('content') <!-- content porque fue el nombre que puse en la plantilla -->
6
7      <h1>Bienvenido al curso: <?php echo $curso ?> </h1>
8
9  @endsection
10 |
```

## 4. Vistas

**Observación:** Si nos fijamos en el último archivo, meter php ahí queda horrible.



```
resources > views > cursos > show.blade.php > ...
1  @extends('layouts.plantilla')
2
3  @section('title','Curso' . $curso)
4
5  @section('content') <!-- content porque fue el nombre que puse en la plantilla -->
6
7      <h1>Bienvenido al curso: <?php echo $curso ?> </h1>
8
9  @endsection
10 |
```

Para esto, Blade nos brinda otra directiva:



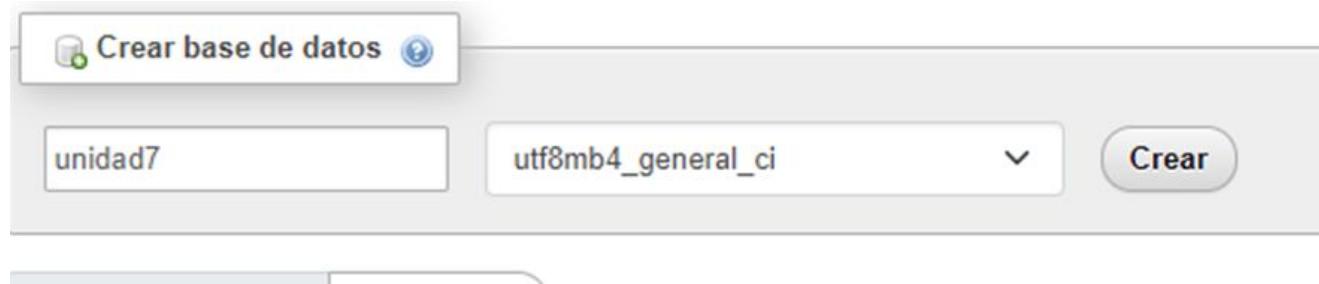
```
resources > views > cursos > show.blade.php > ...
1  @extends('layouts.plantilla')
2
3  @section('title','Curso' . $curso)
4
5  @section('content') <!-- content porque fue el nombre que puse en la plantilla -->
6
7      <h1>Bienvenido al curso: {{$curso}} </h1>
8
9  @endsection
10 |
```

## 5. Base de datos con Laravel

Laravel trabaja con: MySQL, Postgre SQL, SQLite, SQL Server

Primero, **vamos a crear una base de datos.**

Como siempre, nos dirigimos a localhost/phpmyadmin y creamos una base de datos que se llame unidad7



Vamos a hacer que Laravel se conecte con esta base de datos.

## 5. Base de datos con Laravel

Nos dirigimos a la carpeta config > database.php

Y ahí vamos a definir con qué tipo de base vamos a trabajar.

```
1/          // 'default' => env('DB_CONNECTION', 'mysql'),  
18  
19  
20      'default' =>'mysql', |
```

Si bajamos, como nosotros hemos elegido mysql tendremos que rellenar lo correspondiente a este apartado.

Pero si queremos luego compartir nuestro proyecto, por ejemplo, subiéndolo a GitHub, toda esa información va a poder ser vista por cualquier persona. Y son datos sensibles.

## 5. Base de datos con Laravel

Por esta razón, Laravel recomienda guardar la información en `.env`.

Si vemos en la línea 18, se está llamando al método `env` y dentro hay 2 valores. Esto significa que Laravel busca en el archivo `.env`, si encuentra una variable `DB_CONNECTION`

el valor que le hayamos asignado a esta variable, será asignado por defecto a la variable llamada `default`.

Si en el archivo `.env` no hay ningún valor en `DB_CONNECTION`, se le asignará `mysql`.

```
48     'mysql' => [
49         'driver' => 'mysql',
50         'url' => env('DATABASE_URL'),
51         'host' => env('DB_HOST', '127.0.0.1'),
52         'port' => env('DB_PORT', '3306'),
53         'database' => env('DB_DATABASE', 'forge'),
54         'username' => env('DB_USERNAME', 'forge'),
55         'password' => env('DB_PASSWORD', ''),
56         'unix_socket' => env('DB_SOCKET', ''),
57         'charset' => 'utf8mb4',
58         'collation' => 'utf8mb4_unicode_ci',
59         'prefix' => '',
60         'prefix_indexes' => true,
61         'strict' => true,
62         'engine' => null,
63         'options' => extension_loaded('pdo_mysql') ? array_filter([
64             PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'),
65         ]) : [],
66     ],
67 
```

Y así pasa con las credenciales también:

## 5. Base de datos con Laravel

Entonces, nos vamos a dirigir al archivo `.env` y ahí vamos a definir nuestras credenciales.

Las definimos aquí porque este archivo si luego lo quiero subir, no se va a subir en ningún lugar.

Si nos vamos al archivo `.env`, Laravel con la finalidad de ayudarnos, ya ha rellenado por nosotros mismos ciertas credenciales que él cree que vamos a utilizar.

El nombre de la base es `unidad7`, si nos queremos conectar a otra base de datos, cambiamos ese nombre. Y así con todo

```
⚙ .env
1 APP_NAME=Laravel
2 APP_ENV=local
3 APP_KEY=base64:ax83wk4IhWjN2m0EhYrE9dR1/ruVKmLbQBTZKmq7J8E=
4 APP_DEBUG=true
5 APP_URL=http://localhost
6
7 LOG_CHANNEL=stack
8 LOG_DEPRECATIONS_CHANNEL=null
9 LOG_LEVEL=debug
10
11 DB_CONNECTION=mysql
12 DB_HOST=127.0.0.1
13 DB_PORT=3306
14 DB_DATABASE=unidad7
15 DB_USERNAME=root
16 DB_PASSWORD=
17
18 BROADCAST_DRIVER=log
19 CACHE_DRIVER=file
20 FILESYSTEM_DISK=local
21 QUEUE_CONNECTION=sync
22 SESSION_DRIVER=file
23 SESSION_LIFETIME=120
24
25 MEMCACHED_HOST=127.0.0.1
26
27 REDIS_HOST=127.0.0.1
28 REDIS_PASSWORD=null
29 REDIS_PORT=6379
30
```

## 5.1 Migraciones

Ya hemos creado nuestra base de datos, el siguiente paso sería ir rellenando esa base de datos pero hacerlo manualmente es algo tedioso y si en un futuro se modifica, tendríamos que ir modificándolo en nuestro proyecto también.

Laravel nos proporciona una herramienta para esto, llamada **migraciones**.

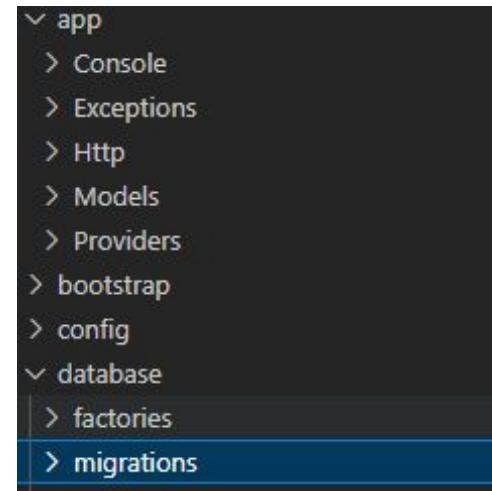
Las **migraciones** son como el control de versiones de nuestra base de datos.

Que básicamente lo que hace es permitir al equipo modificar y compartir el esquema de base de datos de la aplicación. Teniendo un registro de todas esas modificaciones.

Las migraciones se encuentran en:

Database > migrations >

Nos aparecen 4 migraciones



## 5.1 Migraciones

Si abrimos por ejemplo el `create_user_table`  
Observamos que tenemos 2 métodos: up y down

*El método up crea y el down elimina*

`Create` lo que hace es crearnos las tablas  
'users' sería el nombre de la tabla, se le pasa  
una función anónima con variable Blueprint  
que hace crear columnas de nuestras tablas.

`Id`: crea una columna con las siguientes  
propiedades: Integer Unsigned Increment

`string`: agrega una columna varchar. Por  
defecto, son 250 caracteres pero si queremos  
modificarlo, podríamos poner ('name',100) y solo  
sería 100 caracteres. Si queremos agregar más,  
ya se utiliza el método `text()` en lugar de  
`string()`

```
9  /**
1   * Run the migrations.
2   *
3   * @return void
4   */
5  public function up()
6  {
7      Schema::create('users', function (Blueprint $table) {
8          $table->id();
9          $table->string('name');
10         $table->string('email')->unique();
11         $table->timestamp('email_verified_at')->nullable();
12         $table->string('password');
13         $table->rememberToken();
14         $table->timestamps();
15     });
16 }
17
18 /**
19  * Reverse the migrations.
20  *
21  * @return void
22  */
23 public function down()
24 {
25     Schema::dropIfExists('users');
26 }
27
```

## 5.1 Migraciones

**unique:** lo que se almacena en el campo, debe ser único

**timestamp:** guarda fechas. Y se utiliza por si queremos activar la verificación de correos electrónicos. La propiedad **nullable** es para que no pueda quedar vacío.

**rememberToken:** crea una columna de tipo varchar pero de tamaño 100. Se va a almacenar un token y es para mantener la sesión iniciada

```
9  /**
1  * Run the migrations.
2  *
3  * @return void
4  */
5  public function up()
6  {
7      Schema::create('users', function (Blueprint $table) {
8          $table->id();
9          $table->string('name');
10         $table->string('email')->unique();
11         $table->timestamp('email_verified_at')->nullable();
12         $table->string('password');
13         $table->rememberToken();
14         $table->timestamps();
15     });
16 }
17
18 /**
19 * Reverse the migrations.
20 *
21 * @return void
22 */
23 public function down()
24 {
25     Schema::dropIfExists('users');
26 }
27
```

## 5.1 Migraciones

Timestamps: (no confundir con timestamp)  
crea 2 columnas: created\_at y update\_at.

Se utiliza para cada vez que se cree un nuevo registro, en `created_at` se va a quedar registrado la fecha y la hora. Y si lo actualizamos, por ejemplo, si modificamos el nombre del usuario, quedará registrado en `update_at` la hora y la fecha en la que se modificó.

Y como las migraciones son un control de versiones de nuestra base de datos, este método es **súper importante**.

Hay muchos más métodos que los podemos encontrar en: <https://laravel.com/docs/9.x/migrations>

```
9  /**
1   * Run the migrations.
2   *
3   * @return void
4   */
5  public function up()
6  {
7      Schema::create('users', function (Blueprint $table) {
8          $table->id();
9          $table->string('name');
10         $table->string('email')->unique();
11         $table->timestamp('email_verified_at')->nullable();
12         $table->string('password');
13         $table->rememberToken();
14         $table->timestamps();
15     });
16 }
17
18 /**
19  * Reverse the migrations.
20  *
21  * @return void
22  */
23 public function down()
24 {
25     Schema::dropIfExists('users');
26 }
```

## 5.1 Migraciones

Para **crear una migración** escribimos en consola: `php artisan migrate`

Lo que hace es recorrer uno a uno las migraciones que hemos creado y va a ejecutar el método up. Así se creará todas esas tablas con sus respectivas columnas

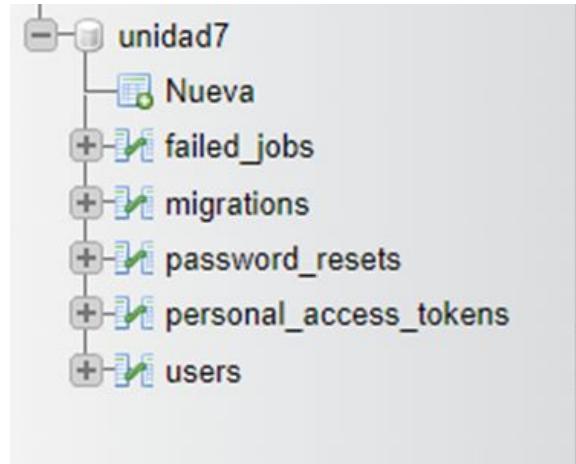
Si observamos, nos ha creado una migrations.

Batch: número de la migración en la que se ha realizado.

**Vamos a crear nuestras propias migraciones.**

Abrimos la consola y escribimos:

```
php artisan make:migration cursos
```



```
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL

PS C:\xampp\htdocs\unidad7> php artisan make:migration cursos

INFO Migration [C:\xampp\htdocs\unidad7\database\migrations\2023_02_13_200826_cursos.php] created successfully.
```

## 5.1 Migraciones

Para **crear una migración** escribimos en consola: `php artisan migrate`

Lo que hace es recorrer uno a uno las migraciones que hemos creado y va a ejecutar el método up. Así se creará todas esas tablas con sus respectivas columnas

Si observamos, nos ha creado una migrations.

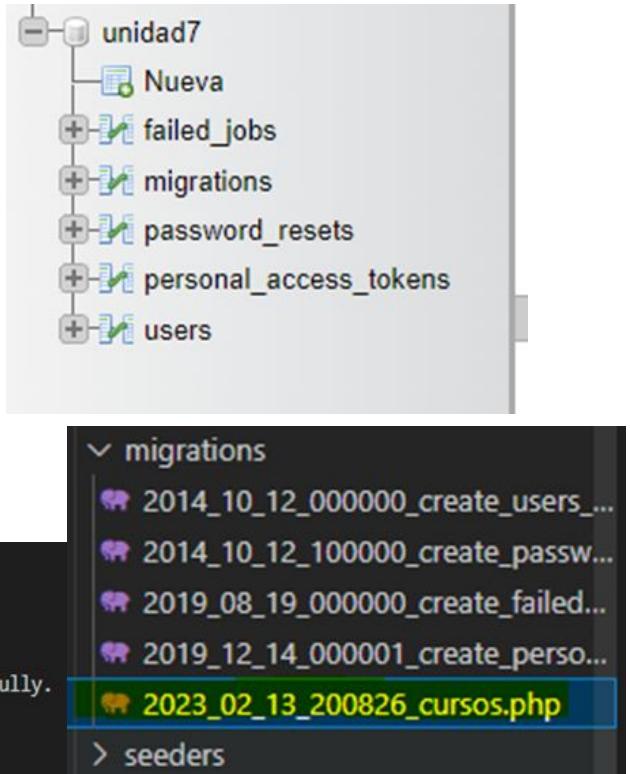
Batch: número de la migración en la que se ha realizado.

**Vamos a crear nuestras propias migraciones.**

Abrimos la consola y escribimos:

```
php artisan make:migration cursos
```

```
PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL  
PS C:\xampp\htdocs\unidad7> php artisan make:migration cursos  
INFO Migration [C:\xampp\htdocs\unidad7\database\migrations\2023_02_13_200826_cursos.php] created successfully.
```



## 5.1 Migraciones

El archivo que se nos genera, tendrá la siguiente estructura:

```
database > migrations > 2023_02_13_200826_cursos.php > class > up
1  <?php
2
3  use Illuminate\Database\Migrations\Migration;
4  use Illuminate\Database\Schema\Blueprint;
5  use Illuminate\Support\Facades\Schema;
6
7  return new class extends Migration
8  {
9      /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14      public function up()
15      {
16          //
17      }
18
19      /**
20      * Reverse the migrations.
21      *
22      * @return void
23      */
24      public function down()
25      {
26          //
27      }
28 };
29
```

## 5.1 Migraciones

Vamos a crear nuestras tablas:

Y volvemos a escribir:

```
php artisan migrate
```

```
7   return new class extends Migration
8  {
9    /**
10     * Run the migrations.
11     *
12     * @return void
13     */
14    public function up()
15    {
16      Schema::create('cursos', function(Blueprint $table){
17        $table->id();
18        $table->string('name');
19        $table->text('descripcion');
20        $table->timestamps(); //created_up y update_up
21      });
22    }
23
24    /**
25     * Reverse the migrations.
26     *
27     * @return void
28     */
29    public function down()
30    {
31      Schema::dropIfExists('cursos'); //Si existe cursos, la elimina
32    }
33  };
34
```

## 5.1 Migraciones

Si nos fijamos, en migrations tenemos:

			id	migration	batch
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	1 2014_10_12_000000_create_users_table	1
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	2 2014_10_12_100000_create_password_resets_table	1
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	3 2019_08_19_000000_create_failed_jobs_table	1
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	4 2019_12_14_000001_create_personal_access_tokens_ta...	1
<input type="checkbox"/>	 Editar	 Copiar	 Borrar	5 2023_02_13_200826_cursos	2

En el lote (batch) nos sale el 2, eso significa que se ha creado en el lote 2.

Si yo quisiera revertir algún cambio, lo va a revertir pero en cuestión de lotes.

Por ejemplo, vamos a revertir todos los cambios producidos en el lote 2. Para ello escribimos:

```
php artisan migrate:rollback
```

## 5.1 Migraciones

Ya no existe la tabla y en migraciones, también desaparece. Si lo volvemos a ejecutar, desaparecerán las del lote 1. Si volvemos a migrar todo, nos saldrá lote 1 a todos.

Vamos a eliminar el documento que hemos creado para **hacerlo de otra forma**.

Primero hacemos un rollback para borrar los registros en nuestra base de datos. Elimino el archivo y vuelvo a hacer un migrate para que se creen las que teníamos.

Escribimos:

```
php artisan make:migration create_cursos_table
```

si escribo esto, veamos qué nos crea:



```
7  return new class extends Migration
8  {
9      /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
16         Schema::create('cursos', function (Blueprint $table) {
17             $table->id();
18             $table->timestamps();
19         });
20     }
21
22     /**
23      * Reverse the migrations.
24      *
25      * @return void
26      */
27     public function down()
28     {
29         Schema::dropIfExists('cursos');
30     }
31 };
```

## 5.1 Migraciones

Si seguimos esto, Laravel nos ayuda a la hora de crear nuestras migraciones. Con esta estructura lo que hago es agregar las columnas de antes:

Si volvemos a ejecutar: `php artisan migrate`

Se ejecuta la última. Y vemos:

	← ↑ →		id	migration	batch			
<input type="checkbox"/>		Editar		Copiar		Borrar	16 2014_10_12_000000_create_users_table	1
<input type="checkbox"/>		Editar		Copiar		Borrar	17 2014_10_12_100000_create_password_resets_table	1
<input type="checkbox"/>		Editar		Copiar		Borrar	18 2019_08_19_000000_create_failed_jobs_table	1
<input type="checkbox"/>		Editar		Copiar		Borrar	19 2019_12_14_000001_create_personal_access_tokens_ta...	1
<input type="checkbox"/>		Editar		Copiar		Borrar	20 2023_02_13_202819_create_cursos_table	2

```
return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('cursos', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->text('descripcion');
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('cursos');
    }
};
```

## 5.1 Migraciones

### Modificación de tablas: insertar una columna

Qué ocurriría si a la tabla users quiero añadirle un nuevo campo. Por ejemplo, quiero almacenar la URL de su avatar.

```
    *
    * Run the migrations.
    *
    * @return void
    */
public function up()
{
    Schema::create('users', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
        $table->timestamp('email_verified_at')->nullable();
        $table->string('password');
        $table->string('avatar');
        $table->rememberToken();
        $table->timestamps();
    });
}
```

Si ejecutamos el comando para migrar: nos pone que no existen migraciones.

## 5.1 Migraciones

### Modificación de tablas: insertar una columna

Esto ocurre porque ya hay un registro de que esa tabla ya existe por lo tanto, esa columna no se va agregar. Cómo podemos hacerlo:

Una opción sería ejecutando 2 veces el `rollback`.

Pero hay otra opción y es ejecutando el comando: `php artisan migrate:fresh`

Va a recorrer uno a uno las migraciones que ya se realizaron y va a ejecutar el método `down`, eliminando todas las tablas que he creado y una vez que ya eliminó todas las tablas, vuelve a recorrer una a una todas las tablas ejecutando el método `up`, creando las tablas correspondientes

Hay que tener cuidado con este método porque es un método destructivo.

Esto es, elimina todas las tablas y las vuelve a crear pero los registros que haya, desaparecen.

Por tanto, este comando solo es aconsejable usarlo cuando nuestro proyecto está en desarrollo, cuando no tenemos registro ninguno.

## 5.1 Migraciones

### Modificación de tablas: insertar una columna

Vamos a ver entonces cómo modificar una columna cuando tenemos registros.

En primer lugar, borramos la columna de avatar y ponemos: `php artisan migrate:fresh`

Nos vamos a nuestra base de datos e insertamos un nuevo registro en `users`.

	<input type="text"/> T <input type="button" value="→"/>	<input type="button" value="▼"/>	<b>id</b>	<b>name</b>	<b>email</b>	<b>email_verified_at</b>	<b>password</b>	<b>remember_token</b>	<b>created_at</b>	<b>updated_at</b>
<input type="checkbox"/>	<input type="button" value="Editar"/>	<input type="button" value="Copiar"/>	<input type="button" value="Borrar"/>	1	pilar	pilarjm@gmail.com	NULL	12345678	NULL	NULL

Seleccionar todo    Para los elementos que están marcados:

Vamos ahora a la consola y escribimos:

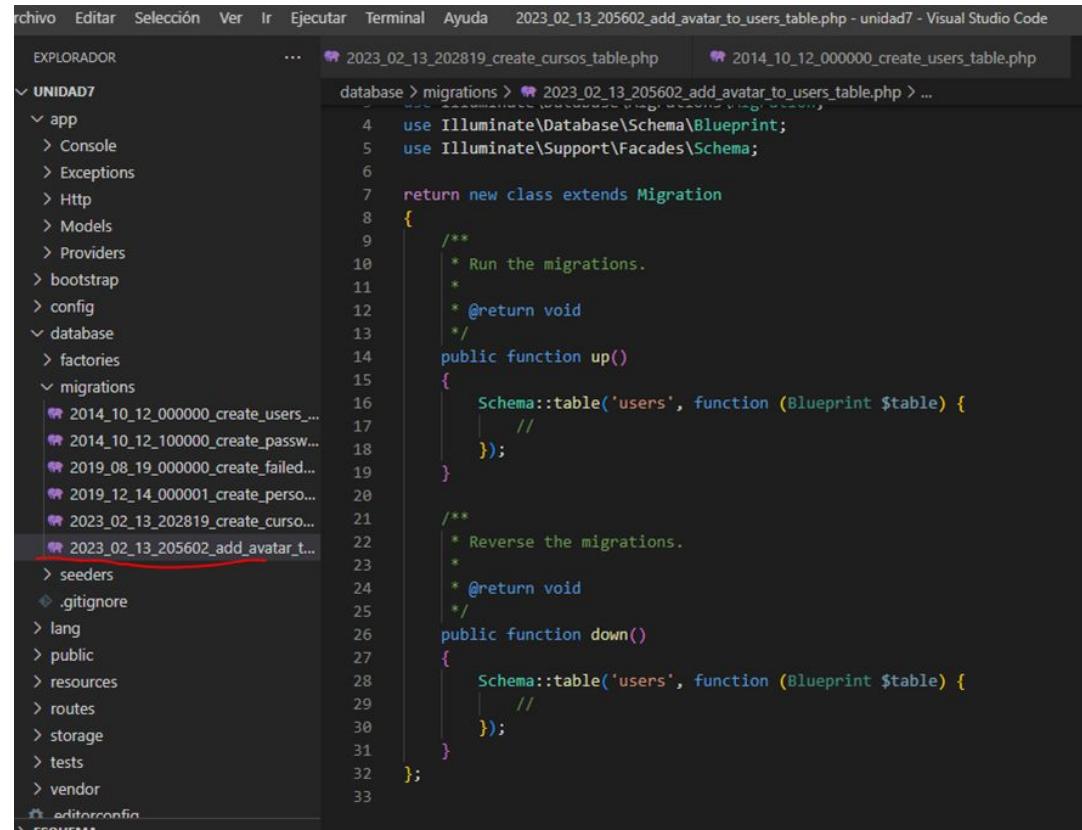
```
php artisan make:migration add_avatar_to_users_table
```

avatar será el nombre de la tabla y users table a la tabla que va.

## 5.1 Migraciones

### Modificación de tablas: insertar una columna

Lo que hace es crearme una migración con cierto código ya escrito.



```
EXPLORADOR          ...  2023_02_13_202819_create_cursos_table.php  2014_10_12_000000_create_users_table.php
UNIDAD7
  app
    > Console
    > Exceptions
    > Http
    > Models
    > Providers
    > bootstrap
    > config
  database
    > factories
    > migrations
      2014_10_12_000000_create_users...
      2014_10_12_100000_create_passw...
      2019_08_19_000000_create_failed...
      2019_12_14_000001_create_perso...
      2023_02_13_202819_create_curso...
      2023_02_13_205602_add_avatar_t...
    > seeders
    .gitignore
    > lang
    > public
    > resources
    > routes
    > storage
    > tests
    > vendor
    editorconfig
  terminal
  Ayuda  2023_02_13_205602_add_avatar_to_users_table.php - unidad7 - Visual Studio Code
```

```
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('users', function (Blueprint $table) {
            //
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('users', function (Blueprint $table) {
            //
        });
    }
};
```

## 5.1 Migraciones

### Modificación de tablas: insertar una columna

En la clase Schema, llama al método table.

Este método se va a utilizar cuando queramos modificar alguna tabla.

Primer parámetro: nombre de la tabla que quiero modificar.

Dentro vamos a poner las modificaciones, por ejemplo agregarle una columna.

Si migramos, nos aparecerá una columna detrás de updated\_at.

```
<?php

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

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('users', function (Blueprint $table) {
            $table->string('avatar')->nullable();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('users', function (Blueprint $table) {
            $table->dropColumn('avatar');
        });
    }
};
```

## 5.1 Migraciones

### Modificación de tablas: insertar una columna

Si queremos poner la columna detrás de donde yo quiera, usamos el método:

Y se pondrá detrás de email



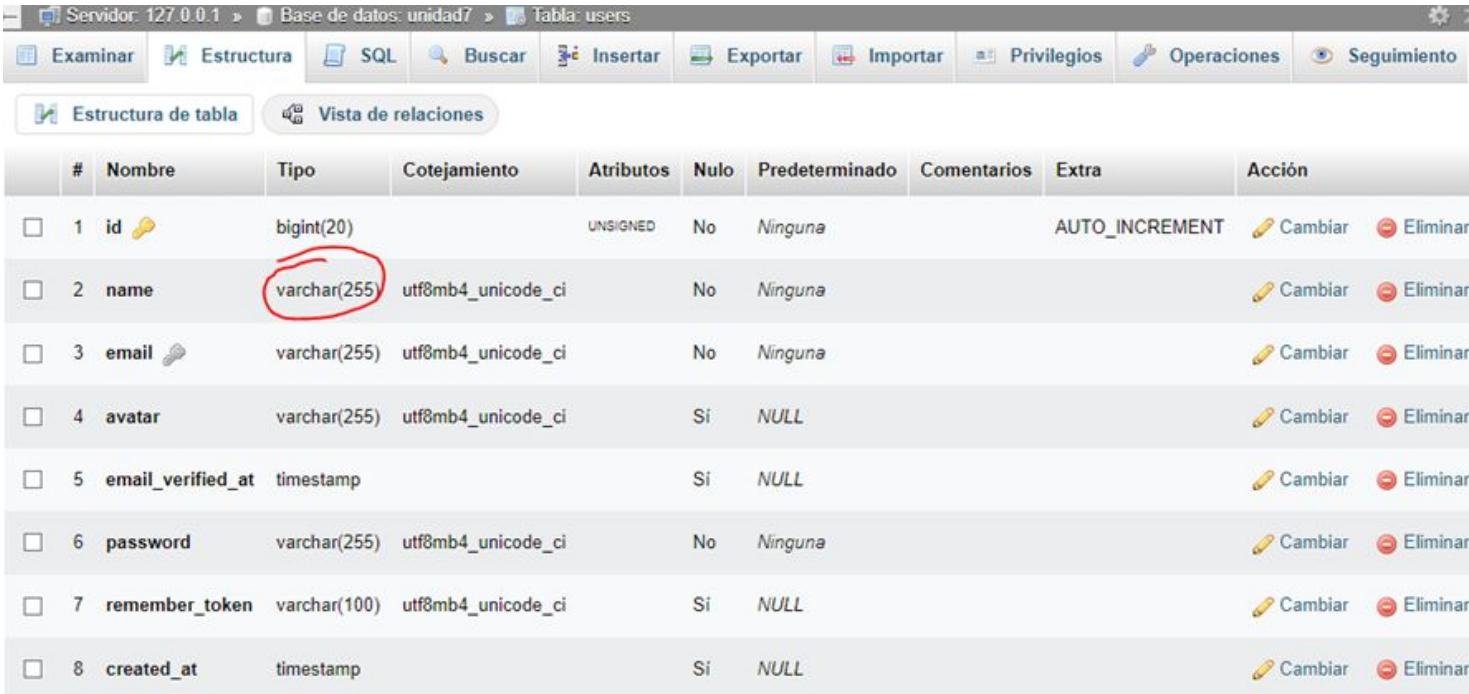
```
7  return new class extends Migration
8  {
9      /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
16         Schema::table('users', function (Blueprint $table) {
17             $table->string('avatar')->nullable()->after('email');
18         });
19     }
20
21     /**
22     * Reverse the migrations.
23     *
24     * @return void
25     */
26     public function down()
27     {
28         Schema::table('users', function (Blueprint $table) {
```

	id	name	email	avatar	email_verified_at	password	remember_token	created_at	updated_at
	Editar	Copiar	Borrar	1	Pilar	pilar@gmail.com	NULL	NULL	NULL
<input type="checkbox"/>	<input type="checkbox"/> Editar	<input type="checkbox"/> Copiar	<input type="checkbox"/> Borrar	1	Pilar	pilar@gmail.com	NULL	NULL	NULL
<input type="checkbox"/>	<input type="checkbox"/> Seleccionar todo	Para los elementos que están marcados:		<input type="checkbox"/> Editar	<input type="checkbox"/> Copiar	<input type="checkbox"/> Borrar	<input type="checkbox"/> Exportar		

## 5.1 Migraciones

### Modificación de tablas: modificar el tipo de una columna

Por ejemplo, en la tabla users, en la columna name quiero modificar el tipo. Modificarlo a que solo acepte 100 caracteres. Cómo lo vamos a hacer



#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	bigint(20)		UNSIGNED	No	Ninguna		AUTO_INCREMENT	Cambiar  Eliminar
2	name	varchar(255)	utf8mb4_unicode_ci		No	Ninguna			Cambiar  Eliminar
3	email	varchar(255)	utf8mb4_unicode_ci		No	Ninguna			Cambiar  Eliminar
4	avatar	varchar(255)	utf8mb4_unicode_ci		Si	NULL			Cambiar  Eliminar
5	email_verified_at	timestamp			Si	NULL			Cambiar  Eliminar
6	password	varchar(255)	utf8mb4_unicode_ci		No	Ninguna			Cambiar  Eliminar
7	remember_token	varchar(100)	utf8mb4_unicode_ci		Si	NULL			Cambiar  Eliminar
8	created_at	timestamp			Si	NULL			Cambiar  Eliminar

## 5.1 Migraciones

### Modificación de tablas: modificar el tipo de una columna

En primer lugar, instalamos la dependencia necesaria, para ello escribimos:

```
composer require doctrine/dbal
```

Luego, creamos una migración:

```
php artisan make:migration  
cambiar_propiedades_to_users_table
```

Y en el archivo agregamos en el método up y down: 

Y realizamos la migración:  
php artisan migrate

```
9  /**
10  * Run the migrations.
11  *
12  * @return void
13  */
14  public function up()
15  {
16      Schema::table('users', function (Blueprint $table) {
17          $table->string('name', 150)->nullable()->change();
18      });
19  }
20
21 /**
22 * Reverse the migrations.
23 *
24 * @return void
25 */
26 public function down()
27 {
28     Schema::table('users', function (Blueprint $table) {
29         $table->string('name', 255)->nullable(false)->change();
30     });
31 }
```

## 5.1 Migraciones

### Modificación de tablas: modificar el tipo de una columna

Observamos que nuestra tabla de datos se ha modificado

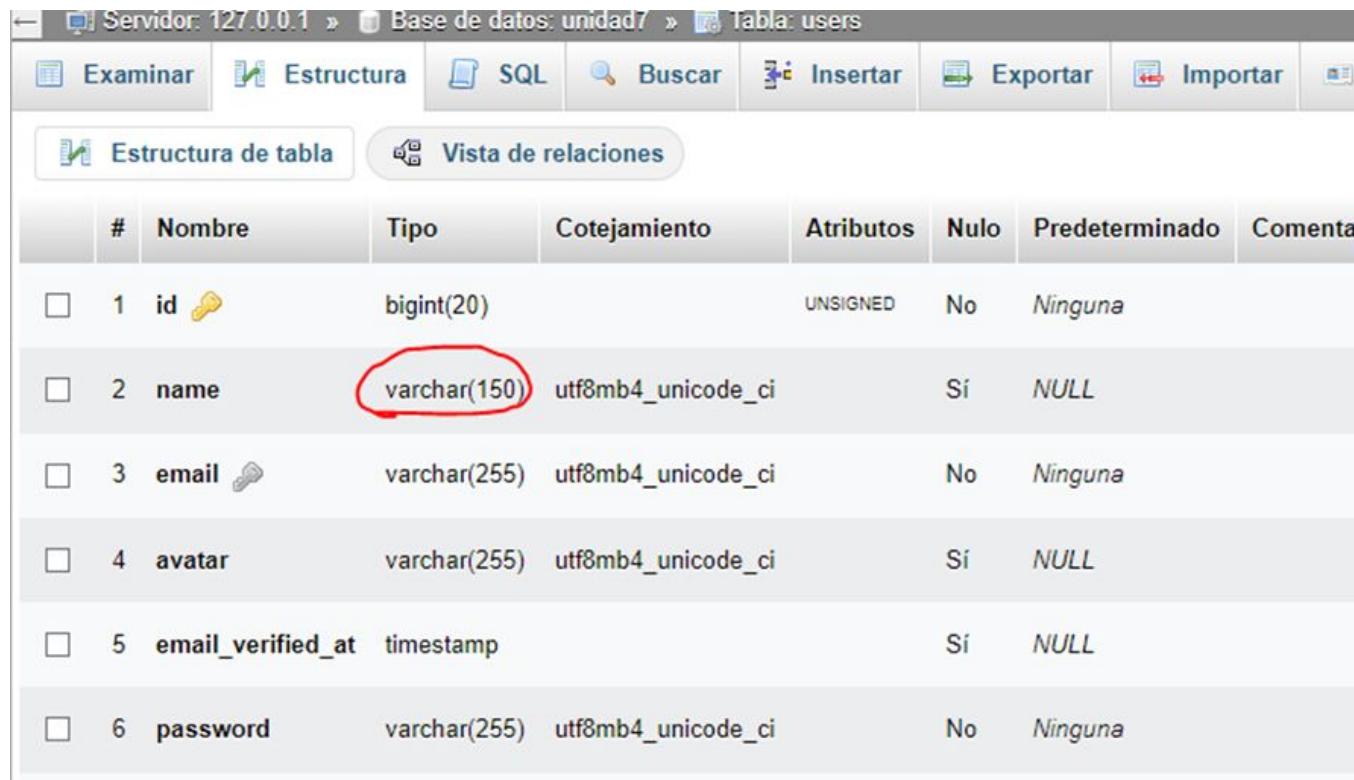
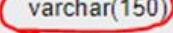


Tabla: users								
	#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comenta
<input type="checkbox"/>	1	id 	bigint(20)		UNSIGNED	No	Ninguna	
<input type="checkbox"/>	2	name	varchar(150) 	utf8mb4_unicode_ci		Sí	NULL	
<input type="checkbox"/>	3	email 	varchar(255)	utf8mb4_unicode_ci		No	Ninguna	
<input type="checkbox"/>	4	avatar	varchar(255)	utf8mb4_unicode_ci		Sí	NULL	
<input type="checkbox"/>	5	email_verified_at	timestamp			Sí	NULL	
<input type="checkbox"/>	6	password	varchar(255)	utf8mb4_unicode_ci		No	Ninguna	

## 5.2 ORM Eloquent

### Modelos

Vamos a ver cómo introducir, recuperar, actualizar y eliminar registros de nuestra base de datos.

Lo vamos a hacer con **ORM eloquent**

ORM es un modelo de programación que nos permite tratar cada registro como si fuera un objeto. Cada objeto con su respectiva propiedad (columnas de la tabla)

→ El ORM de Laravel es **eloquent**.

Para trabajar con eloquent necesitamos crear **Modelos**.

Los modelos se encuentran dentro de App>Models

## 5.2 ORM Eloquent

### Modelos

Creemos un modelo para nuestra tabla cursos

Para ello escribimos `php artisan make:model Curso`

Para poner el nombre, usamos la convención de Laravel, donde un modelo de nombre `Curso` (en singular y comenzando con mayúscula) se relaciona con una tabla llamada `cursos` (en plural y en minúsculas).

*Por ejemplo, si yo creo un modelo llamado User, eloquent va a interpretar que él va a administrar una tabla users.*

Esta convención de nombres en singular y plural se realiza para **palabras en inglés**.

Después veremos cómo no usar la convención

## 5.2 ORM Eloquent

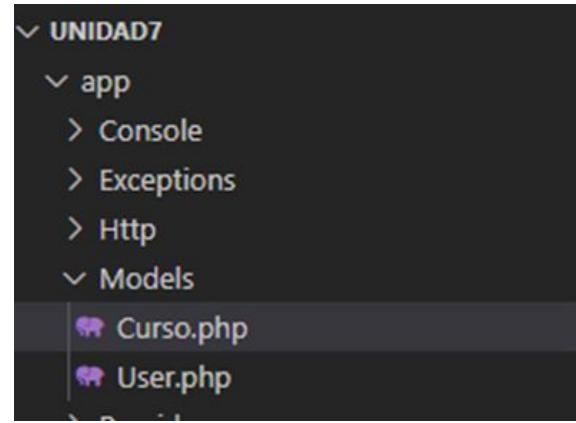
### Modelos

Al ejecutar el comando anterior, nos va a crear el modelo curso:

El modelo user, ya viene creado por defecto.

Si abrimos curso.php:

```
app > Models > Curso.php > ...
1  <?php
2
3  namespace App\Models;
4
5  use Illuminate\Database\Eloquent\Factories\HasFactory;
6  use Illuminate\Database\Eloquent\Model;
7
8  class Curso extends Model
9  {
10     use HasFactory;
11 }
12 |
```



## 5.2 ORM Eloquent

### Modelos

Ahora, vamos a utilizar una herramienta que nos permite usar eloquent desde una línea de comandos: **Tinker**.

Vamos a poder utilizar todos los comandos de Eloquent pero desde la terminal.

Escribimos: `php artisan tinker` (y la terminal cambiará un poco)

Para salir `exit`

Primero, indico que quiero hacer uso de ese modelo. En este caso, Curso. Escribo:

## 5.2 ORM Eloquent

### Modelos

```
Use App\Models\Curso;
```

```
PS C:\xampp\htdocs\unidad7> php artisan tinker  
Psy Shell v0.11.12 (PHP 8.1.10 - cli) by Justin Hileman  
> use App\Models\Curso;
```

Ahora creamos una instancia de esta clase. Definimos la variable:

```
$curso = new Curso;
```

Y lo llenamos de propiedades:

```
$curso->name = 'Laravel';  
$curso->descripción = 'El mejor framework de PHP';  
$curso;
```

Para guardarla:

```
$curso->save();
```

Si vamos a nuestra base de datos:



	id	name	descripcion	created_at	updated_at
<input type="checkbox"/>	1	Laravel	El mejor framework de PHP	2023-02-16 11:58:03	2023-02-16 11:58:03

## 5.2 ORM Eloquent

### Modelos

Ya hemos creado un registro sin necesidad de meter ninguna sentencia SQL.

Si ponemos en la consola \$curso;

Me devuelve:



```
> $curso;
= App\Models\Curso {#3709
    name: "Laravel",
    descripcion: "El mejor framework de PHP",
    updated_at: "2023-02-16 11:58:03",
    created_at: "2023-02-16 11:58:03",
    id: 1,
}
```

Para modificar una propiedad:

```
$curso->descripción = 'El mejor framework de todos';
```

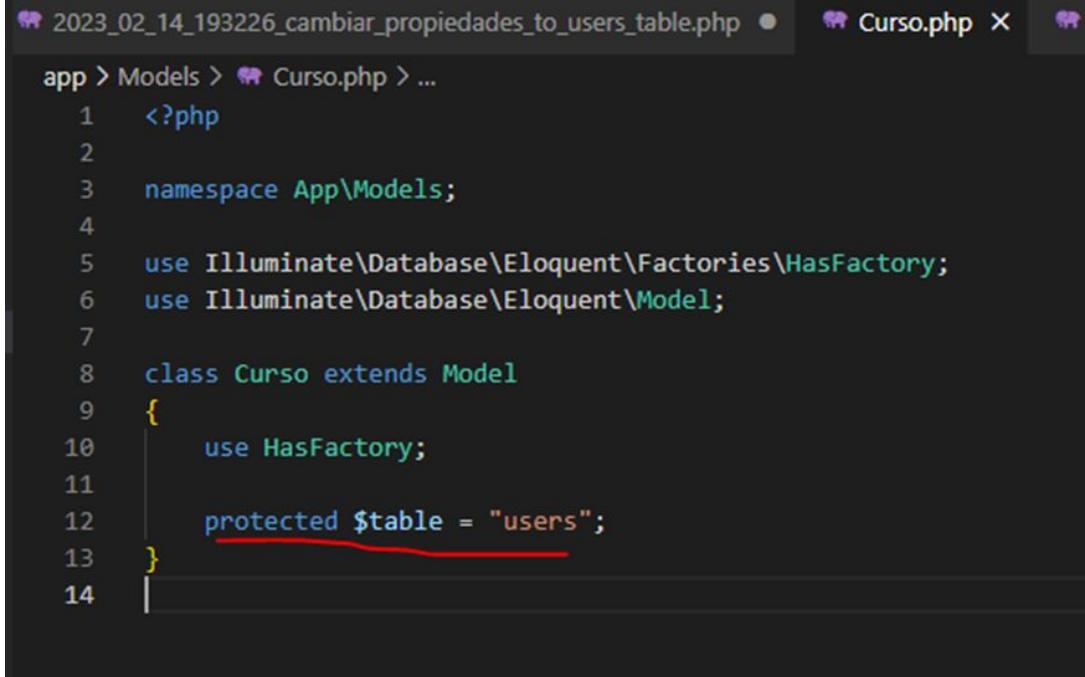
## 5.2 ORM Eloquent

### Modelos

Si no queremos usar la convención que hemos dicho antes.

Si yo quiero que el modelo `Curso`, no se encargue de administrar la tabla `curso` si no otra, por ejemplo `users`.

Pues agregamos:



```
2023_02_14_193226_cambiar_propiedades_to_users_table.php • Curso.php X
app > Models > Curso.php > ...
1 <?php
2
3 namespace App\Models;
4
5 use Illuminate\Database\Eloquent\Factories\HasFactory;
6 use Illuminate\Database\Eloquent\Model;
7
8 class Curso extends Model
9 {
10     use HasFactory;
11
12     protected $table = "users";
13 }
14
```

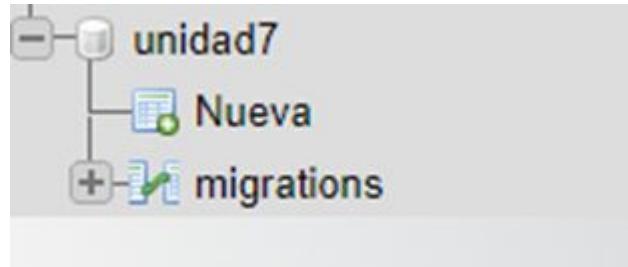
## 5.2 ORM Eloquent

### Llenado de tablas

Vamos a llenar de datos nuestra tabla. Y lo vamos a llenar con datos de prueba (llamados **seeders**) que son datos que se van a agregar a través de unos **factories** que tiene Laravel.

Antes de nada, vamos a eliminar todas las tablas que hemos creado en la BD:

```
Php artisan migrate:reset
```



Lo eliminamos porque actualmente, queremos eliminar las 2 migraciones que hemos creado. (avatar y propiedades).

En la migración: `create_cursos` vamos a añadir una nueva columna:

## 5.2 ORM Eloquent

### Llenado de tablas

```
database > migrations > 2023_02_13_202819_create_cursos_table.php > class > up
 5  use Illuminate\Support\Facades\Schema;
 6
 7  return new class extends Migration
 8  {
 9    /**
10     * Run the migrations.
11     *
12     * @return void
13     */
14    public function up()
15    {
16        Schema::create('cursos', function (Blueprint $table) {
17            $table->id();
18            $table->string('name');
19            $table->text('descripcion');
20            $table->text('categoria');
21            $table->timestamps();
22        });
23    }
24}
```

No lo migramos aún

## 5.2 ORM Eloquent

### Llenado de tablas: SEEDERS

Nos dirigimos a la carpeta seeders, justo debajo de migrations.

Y aquí escribimos el siguiente código:



Y escribimos:

php artisan migrate:fresh

```
1  <?php
2
3  namespace Database\Seeders;
4
5  // use Illuminate\Database\Console\Seeds\WithoutModelEvents;
6  use Illuminate\Database\Seeder;
7  use App\Models\Curso;
8
9  class DatabaseSeeder extends Seeder
10 {
11     /**
12      * Seed the application's database.
13      *
14      * @return void
15      */
16     public function run()
17     {
18         //Creando los datos que pasaremos a la base de datos
19
20         $cursos = [
21             [
22                 1 => [
23                     'name'=>'Nombre del curso',
24                     'descripcion' => 'Descripcion de uno de los cursos',
25                     'categoria' => 'Categoria del curso'
26                 ],
27                 [
28                     2 => [
29                         'name'=>'Nombre del curso dos',
30                         'descripcion' => 'Descripcion de uno de los cursos',
31                         'categoria' => 'Categoria del curso dos'
32                     ],
33                 ];
34
35             //Recorriendo los cursos y enviandolo a la base de datos
36
37             foreach($cursos as $key => $value){
38                 | Curso::create($value);
39             }
40         ]
41     }
42 }
```

## 5.2 ORM Eloquent

### Llenado de tablas: SEEDERS

Problema: imagina que tenemos muchísimos archivos, tendríamos un código gigantesco. Y eso es lo que evita Laravel. Lo ideal sería entonces es que todo el código de DatabaseSeeder.php esté aparte.

Es lo que vamos a hacer.

```
php artisan make:seeder CursoSeeder
```

Y en CursoSeeder.php copio lo que había puesto en DatabaseSeeder.php.

En DatabaseSeeder.php escribo:



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

```
database > seeders > DatabaseSeeder.php > ...
1  <?php
2
3  namespace Database\Seeders;
4
5  // use Illuminate\Database\Console\Seeds\WithoutModelEvents;
6  use Illuminate\Database\Seeder;
7  use App\Models\Curso;
8
9  class DatabaseSeeder extends Seeder
10 {
11     /**
12      * Seed the application's database.
13      *
14      * @return void
15      */
16     public function run()
17     {
18         $this->call(CursoSeeder::class);
19     }
20 }
21 }
```

## 5.2 ORM Eloquent

### Llenado de tablas: FACTORIES

Acabamos de ver cómo introducir datos manualmente, pero el problema se complica cuando queremos meter por ejemplo 100 registros. Por esta razón, vamos a usar la herramienta de **factories**.

**Factories:** herramienta para llenar nuestra BD con datos de prueba.

Debemos especificarle qué tipo de datos y la cantidad de registros.

El factory creado se ubica en database>factories

**Creamos un Factory:** `_ php artisan make:factory CursoFactory -model=Curso`

Agregamos lo siguiente al archivo dentro del método `definition()`:

## 5.2 ORM Eloquent

### Llenado de tablas: FACTORIES

```
1 <?php
2
3 namespace Database\Factories;
4
5 use Illuminate\Database\Eloquent\Factories\Factory;
6
7 /**
8 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Curso>
9 */
10 class CursoFactory extends Factory
11 {
12     /**
13      * Define the model's default state.
14      *
15      * @return array<string, mixed>
16      */
17     public function definition()
18     {
19         return [
20             'name' => $this->faker->sentence(), //name se llenará con una frase
21             'description' => $this->faker->paragraph(), //description se llenará con un párrafo
22             'categoria' => $this->faker->randomElement(['Desarrollo web', 'Diseño web']) //lo que hace es elegir entre esas 2
23         ];
24     }
25 }
26 
```

## 5.2 ORM Eloquent

### Llenado de tablas: FACTORIES

Para utilizarlo, vamos a `CursoSeeder.php`, eliminamos lo que habíamos creado y ponemos:



Crea 50 registros.

Escribimos: `php artisan migrate:fresh -seed`

Y si nos vamos a la BD en phpmyadmin, vemos que se han creado los 50 registros con pruebas.

```
database > seeders > 🗂 CursoSeeder.php > 🛡 CursoSeeder > ⚙ run
1  <?php
2
3  namespace Database\Seeders;
4
5  use Illuminate\Database\Console\Seeds\WithoutModelEvents;
6  use Illuminate\Database\Seeder;
7  use App\Models\Curso;
8
9  class CursoSeeder extends Seeder
10 {
11     /**
12      * Run the database seeds.
13      *
14      * @return void
15      */
16     public function run()
17     {
18         Curso::factory(50)->create();
19     }
20 }
21 }
```

## 5.2 ORM Eloquent

### Llenado de tablas: FACTORIES

Si hacemos lo mismo con Users



Si nos vamos a la BD en phpmyadmin, vemos que se han creado los 10 registros con pruebas.

	<input type="button" value="←"/> <input type="button" value="→"/>		<input type="button" value="id"/>	<input type="button" value="name"/>	<input type="button" value="email"/>	<input type="button" value="avatar"/>	<input type="button" value="em"/>
<input type="checkbox"/>	<input type="button" value="Editar"/>	<input type="button" value="Copiar"/>	<input type="button" value="Borrar"/>	1 Mr. Gilbert Yundt	bogdan.dashawn@example.com	NULL	2021-13-12
<input type="checkbox"/>	<input type="button" value="Editar"/>	<input type="button" value="Copiar"/>	<input type="button" value="Borrar"/>	2 Dahlia King Sr.	neoma56@example.net	NULL	2021-13-12
<input type="checkbox"/>	<input type="button" value="Editar"/>	<input type="button" value="Copiar"/>	<input type="button" value="Borrar"/>	3 Cleta Macejkovic	xkeebler@example.com	NULL	2021-13-12
<input type="checkbox"/>	<input type="button" value="Editar"/>	<input type="button" value="Copiar"/>	<input type="button" value="Borrar"/>	4 Mr. Milo ...	mckenna25@example.net	NULL	2021-13-12

```
database > seeders > 🌸 CursoSeeder.php > 🛡 CursoSeeder > ⚒ run
 1  <?php
 2
 3  namespace Database\Seeders;
 4
 5  use Illuminate\Database\Console\Seeds\WithoutModelEvents;
 6  use Illuminate\Database\Seeder;
 7  use App\Models\Curso;
 8  use App\Models\User;
 9
10 class CursoSeeder extends Seeder
11 {
12     /**
13      * Run the database seeds.
14      *
15      * @return void
16      */
17     public function run()
18     {
19
20         Curso::factory(50)->create();
21
22         User::factory(10)->create();
23
24     }
25 }
```

## 5.2 ORM Eloquent

### Generador de consultas de Eloquent

Consultas simples: Como aún no hemos visto vistas, lo vamos a ver utilizando la **consola de Tinker**:

```
php artisan tinker
```

Primero, accedemos al modelo Curso:

```
use App\Models\Curso;
```

```
$curso = Curso::all();
```

Lo que hace es devolver todos los registros de la tabla

```
> use App\Models\Curso;
> $curso = Curso::all();
= Illuminate\Database\Eloquent\Collection {#4707
  all: [
    App\Models\Curso {#4709
      id: 1,
      name: "Itaque nesciunt saepe ab vitae aut eveniet.",
      descripcion: "Natus et sed ea fuga omnis. Nulla ratione quaerat earum id. Quia porro at et non sed totam nihil et.",
      categoria: "Diseño web",
      created_at: "2023-02-16 13:37:28",
      updated_at: "2023-02-16 13:37:28",
    },
    App\Models\Curso {#4710
      id: 2,
      name: "Et rerum quae vitae nemo.",
      descripcion: "Praesentium ut deserunt asperiores rerum et animi earum. Aut velit nam eaque explicabo dolorum vel. Ad quas saepe omnis aut.",
      categoria: "Diseño web",
      created_at: "2023-02-16 13:37:28",
      updated_at: "2023-02-16 13:37:28",
    },
    App\Models\Curso {#4711
      id: 3,
      name: "Quis voluptate et veniam et.",
      descripcion: "Numquam magni nesciunt est molestiae ipsa ipsa asperiores. Ea quia sit distin
```

## 5.2 ORM Eloquent

### Generador de consultas de Eloquent

#### Consultas simples:

- Si no quiero que me devuelvan todos los registros, solo los que tienen la categoría diseño web:  
`$cursos = Curso::where('categoria', 'Diseño web')->get();`
- Las categorías las generan en orden ascendentes, en referencia a su id. Si yo quiero que el orden sea invertido:  
`$cursos = Curso::where('categoria', 'Diseño web')->orderBy('id', 'desc')->get();`
- Lo que hace el método orderBy es ordenar por id de manera descendente. Si yo quiero ordenar por nombre, de manera ascendente:  
`$cursos = Curso::where('categoria', 'Diseño web')->orderBy('name', 'asc')->get();`
- Si quiero que me devuelva el primer registro:  
`$cursos = Curso::where('categoria', 'Diseño web')->orderBy('name', 'asc')->first();`

## 5.2 ORM Eloquent

### Generador de consultas de Eloquent

#### Consultas simples:

- Si observamos, cada vez que ejecutamos una consulta, me devuelve todos los campos, es decir, el campo id, name, descripción, etc. Si quiero que me devuelva un campo en particular, por ejemplo, el name y la descripción:

```
$cursos = Curso::select('name', 'descripcion')->get();
```

- Si yo por ejemplo, quiero que al consultar la base de datos, sea por ejemplo name pero como un title:  
\$cursos = Curso::select('name as title', 'description')->get();

- Si queremos una cantidad de registros en particular: se hace con take(5) devuelve 5 registros.

```
$cursos = Curso::select('name as title', 'description')->take(5)->get();
```

**Nota:** Curso::where('id', 5)->first() es equivalente a Curso::find(5)

## 5.2 ORM Eloquent

### Generador de consultas de Eloquent

#### Consultas más complejas

`where()` puede contener consultas más complejas

Por ejemplo, si queremos los registros de id desde el 1 al 44.

```
$cursos = Curso::where('id', '<', 45)->get();
```

Podemos hacer lo mismo con `<` y con `<>` (diferente).

También podemos usar con `where()` caracteres especiales.

Por ejemplo, la siguiente consulta lo que nos hace es devolver todos los registros que en alguna parte de su nombre tengan la palabra dorime, da igual que tenga algo adelante o algo atrás. Esto se hace:

```
$cursos = Curso::where('name', 'like', '% dorime %')->get();
```

## 5.2 ORM Eloquent

### Mutadores y accesores

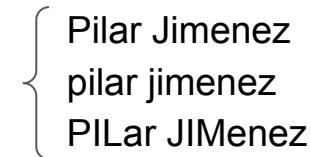
La mayoría de información que vamos a almacenar en nuestra BD, nos llega desde formularios, por ejemplo un formulario de registro.

Puede ocurrir que el usuario escriba su nombre de diferentes maneras:

Entonces, esto genera 3 registros. Y esto, en ocasiones, puede ser un inconveniente

Entonces, lo que vamos a hacer es que indistintamente de cómo el usuario escriba su nombre, Eloquent va a transformar esa cadena en minúsculas y lo guarde en la BD en minúscula como un único registro.

Cómo lo hacemos: a través de **Mutadores**



Pilar Jimenez  
pilar jimenez  
PILar JIMenez

## 5.2 ORM Eloquent

### Mutadores

Estos **mutadores** se colocarán dentro de los **Modelos** encargados de almacenar la información que les llega y transformarla antes de guardarla

Nos vamos a: Models > User.php

Y vamos a agregar un mutador

Lo primero que tenemos que hacer es agregar:



```
app > Models > User.php > ...
1  <?php
2
3  namespace App\Models;
4
5  // use Illuminate\Contracts\Auth\MustVerifyEmail;
6  use Illuminate\Database\Eloquent\Factories\HasFactory;
7  use Illuminate\Foundation\Auth\User as Authenticatable;
8  use Illuminate\Notifications\Notifiable;
9  use Laravel\Sanctum\HasApiTokens;
10
11 use Illuminate\Database\Eloquent\Casts\Attribute;
```

## 5.2 ORM Eloquent

### Mutadores

Y agregamos un nuevo método dentro de la clase User que debe tener el mismo nombre que el atributo que quiera modificar:

```
47 protected function name():Attribute  
48 {  
49     return new Attribute(  
50         set:function($value){ //la variable captura lo que el usuario mete en name  
51             return strtolower($value); //la funcion strolower modifica a minusculas  
52         }  
53     );  
54 }  
55 }  
56  
57 }
```

Esto lo que hace es que cuando haya un registro, antes de guardarlo en nuestra base de datos se ponga en minúscula.

## 5.2 ORM Eloquent

### Mutadores

Para probarlo, ingresamos en el Tinker:

```
php artisan tinker  
use App\Models\User  
$user = new User();
```

Si ingresamos en nuestra base de datos:

Vemos que el nombre se ha almacenado en minúscula.

```
> use App\Models\User  
> $user = new User();  
= App\Models\User {#3709}  
  
> $user->name = "PiLAr JiMeNeZ";  
= "PiLAr JiMeNeZ"  
  
> $user->email = "pilar@gmail.com";  
= "pilar@gmail.com"  
  
> $user->password = bcrypt("12345678");  
= "$2y$10$krahmrvoWozJ7A3B2z0pF.kZzENrgG4vLJ3LwTh0Qgl85ETkZpw2a"  
  
> $user->save();  
= true
```

<input type="checkbox"/>	Editar	Copiar	Borrar	11	pilar jimenez	pilar@gmail.com	NULL	NULL	\$2y\$10\$krahmrvoWozJ7A3B2z0pF.kZzENrgG4vLJ3LwT
--------------------------	--------	--------	--------	----	------------------	-----------------	------	------	--

## 5.2 ORM Eloquent

### Accesores

Es muy común que queramos lograr el efecto contrario y es que cuando hagamos una consulta a nuestra base de datos, queramos que la representación de un registro venga transformado. Por ejemplo, el nombre así escrito: Pilar Jiménez

Para ello, vamos a hacer uso de Accesores

Nos volvemos a ir a User.php  
y escribimos donde antes:

```
46 | protected function name():Attribute
47 | {
48 |     return new Attribute(
49 |         get: function($value) {
50 |             return ucwords($value); //transforma
51 |         },
52 |
53 |         set:function($value){ //la variable captura lo que el usuario mete en name
54 |             return strtolower($value); //la funcion strolower modifica a minusculas
55 |         }
56 |
57 |     );
58 |
59 | }
```

## 5.2 ORM Eloquent

### Accesores

Para comprobarlo, abrimos la consola y el tinker

```
php artisan tinker
```

```
Use App\Models\User;
```

```
$user = User::first();
```

```
$user->name;
```

```
> $user = User::first();
= App\Models\User {#4669
    id: 1,
    name: "Mr. Gilbert Yundt",
    email: "bogan.dashawn@example.com",
    avatar: null,
    email_verified_at: "2023-02-16 13:37:29",
    #password: "$2y$10$92IXUNpkj00r0Q5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi",
    #remember_token: "POk3ZPoi8H",
    created_at: "2023-02-16 13:37:29",
    updated_at: "2023-02-16 13:37:29",
}

> $user->name;
= "Mr. Gilbert Yundt"
```

## 5.2 ORM Eloquent

### Accesores

#### Novedades de PHP 8

##### Funciones flechas

Podemos hacer uso de la notación siguiente para escribir funciones:

```
get: fn($value) => ucwords($value)
```

```
46
47     protected function name():Attribute
48     {
49         return new Attribute(
50             get: fn($value) => ucwords($value),
51             set:function($value){ //la variable captu
52                 return strtolower($value); //la funci
53             }
54         );
55     }
56 }
57 }
```

```
protected function name():Attribute
{
    return new Attribute(
        get: fn($value) => ucwords($value),
        set: fn($value) => strtolower($value)
    );
}
```

## 6. CRUD

En informática, CRUD es el acrónimo de “Crear, Leer, Actualizar y Borrar” que se usa para referirse a las funciones básicas en bases de datos.

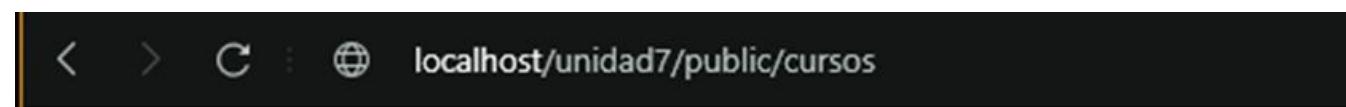
Vamos a crear nuestro primer CRUD en Laravel

En primer lugar, si vamos a

Ahora lo que quiero que me muestre no es ese mensaje, si no los cursos que tengo almacenados en mi base de datos.

Cómo hacemos esto:

Nos vamos a `Http>Controller>CursoController`



## 6. CRUD

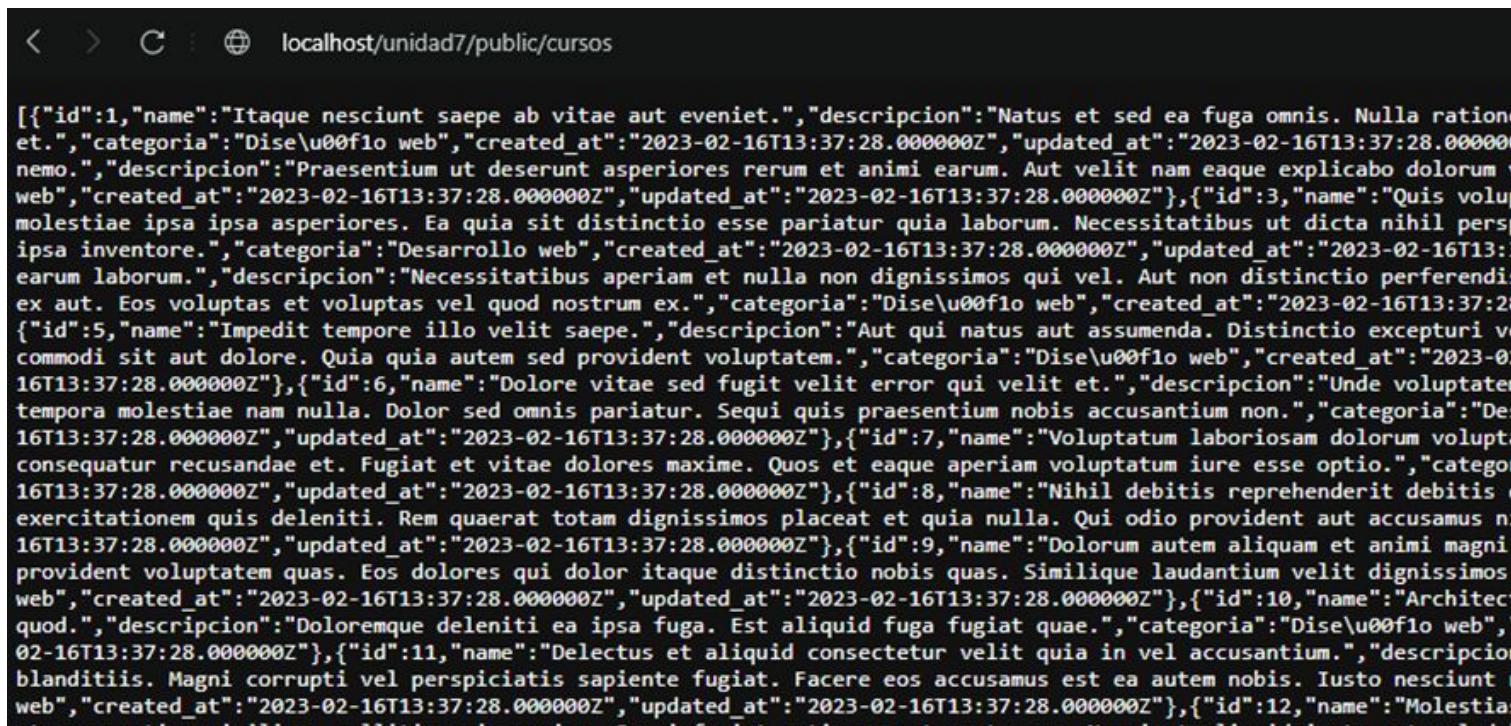
Y escribimos:

Lo que hacemos con esto es que en la variable cursos, tengo todos los registros

```
8  class CursoController extends Controller
9  {
10
11
12      public function index(){ //Hace referencia a cursos
13
14          $cursos= Curso::all();
15          return $cursos
16
17          return view('cursos.index');
18
19      }
}
```

## 6. CRUD

Si lo comprobamos:



The screenshot shows a browser window with the URL `localhost/unidad7/public/cursos`. The page content is a JSON array of 12 objects, each representing a course. The objects have fields: id, name, descripcion, categoria, created\_at, and updated\_at. The names of the courses are generated in Latin, and the descriptions are also in Latin, describing various aspects of education and technology.

```
[{"id":1,"name":"Itaque nesciunt saepe ab vitae aut eveniet.", "descripcion":"Natus et sed ea fuga omnis. Nulla ratione et.", "categoria":"Dise\u00f1o web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}, {"id":2,"name":"Praesentium ut deserunt asperiores rerum et animi earum. Aut velit nam eaque explicabo dolorum v", "descripcion":"Praesentium ut deserunt asperiores rerum et animi earum. Aut velit nam eaque explicabo dolorum v", "categoria":"Dise\u00f1o web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}, {"id":3,"name":"Quis voluptas molestiae ipsa ipsa asperiores. Ea quia sit distinctio esse pariatur quia laborum. Necessitatibus ut dicta nihil persp", "descripcion":"Quis voluptas molestiae ipsa ipsa asperiores. Ea quia sit distinctio esse pariatur quia laborum. Necessitatibus ut dicta nihil persp", "categoria":"Desarrollo web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}, {"id":4,"name":"ipsa inventore.", "descripcion":"ipsa inventore.", "categoria":"Desarrollo web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}, {"id":5,"name":"Impedit tempore illo velit saepe.", "descripcion":"Aut qui natus aut assumenda. Distinctio excepturi vel commodi sit aut dolore. Quia quia autem sed provident voluptatem.", "categoria":"Dise\u00f1o web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}, {"id":6,"name":"Dolore vitae sed fugit velit error qui velit et.", "descripcion":"Unde voluptatem tempora molestiae nam nulla. Dolor sed omnis pariatur. Sequi quis praesentium nobis accusantium non.", "categoria":"Desarrollo web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}, {"id":7,"name":"Voluptatum laboriosam dolorum voluptatum consequatur recusandae et. Fugiat et vitae dolores maxime. Quos et eaque aperiam voluptatum iure esse optio.", "descripcion":"Voluptatum laboriosam dolorum voluptatum consequatur recusandae et. Fugiat et vitae dolores maxime. Quos et eaque aperiam voluptatum iure esse optio.", "categoria":"Dise\u00f1o web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}, {"id":8,"name":"Nihil debitis reprehenderit debitis et exercitationem quis deleniti. Rem quaerat totam dignissimos placeat et quia nulla. Qui odio provident aut accusamus nisi.", "descripcion":"Nihil debitis reprehenderit debitis et exercitationem quis deleniti. Rem quaerat totam dignissimos placeat et quia nulla. Qui odio provident aut accusamus nisi.", "categoria":"Dise\u00f1o web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}, {"id":9,"name":"Dolorum autem aliquam et animi magni provident voluptatem quas. Eos dolores qui dolor itaque distinctio nobis quas. Similique laudantium velit dignissimos.", "descripcion":"Dolorum autem aliquam et animi magni provident voluptatem quas. Eos dolores qui dolor itaque distinctio nobis quas. Similique laudantium velit dignissimos.", "categoria":"Dise\u00f1o web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}, {"id":10,"name":"Architect quod.", "descripcion":"Doloremque deleniti ea ipsa fuga. Est aliquid fuga fugiat quae.", "categoria":"Dise\u00f1o web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}, {"id":11,"name":"Delectus et aliquid consectetur velit quia in vel accusantium.", "descripcion":"Delectus et aliquid consectetur velit quia in vel accusantium.", "categoria":"Dise\u00f1o web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}, {"id":12,"name":"Molestias blanditiis. Magni corrupti vel perspiciatis sapiente fugiat. Facere eos accusamus est ea autem nobis. Iusto nesciunt et.", "descripcion":"Molestias blanditiis. Magni corrupti vel perspiciatis sapiente fugiat. Facere eos accusamus est ea autem nobis. Iusto nesciunt et.", "categoria":"Dise\u00f1o web", "created_at":"2023-02-16T13:37:28.00000Z", "updated_at":"2023-02-16T13:37:28.00000Z"}]
```

## 6. CRUD

Obviamente, queremos que este contenido tenga un determinado formato.

Para ello, en primer lugar pongo:

Y nos vamos a la vista  
index.blade.php:

¿Cómo voy mostrando unos datos en una  
tabla? A través de un **bucle for**.

Blade tiene una directiva que es **@foreach**  
que nos permite hacer esto.

```
Http > Controllers > CursoController.php > ...
<?php

namespace App\Http\Controllers;

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

class CursoController extends Controller
{
    public function index(){ //Hace referencia a cursos
        $cursos= Curso::all();
        return view('cursos.index', compact('cursos'));
    }
}
```

## 6. CRUD

Si ponemos:



Nos imprimiría



### Bienvenidos a la página cursos

- {"id":1,"name":"Itaque nesciunt saepe ab vitae aut eveniet.", "des et.", "categoria":"Dise\u00f1o web", "created\_at": "2023-02-16T13:37:28.00000Z"}
- {"id":2,"name":"Et rerum quae vitae nemo.", "descripcion":"Praes aut.", "categoria":"Dise\u00f1o web", "created\_at": "2023-02-16T13:37:28.00000Z"}
- {"id":3,"name":"Quis voluptate et veniam et.", "descripcion":"Nu nihil perspiciatis. Consequatur perspiciatis quod ipsa et magnam 16T13:37:28.00000Z"}
- {"id":4,"name":"Nihil enim at quae et architecto earum laborum. iste eligendi harum placeat rerum est ex aut. Eos voluptas et volu 16T13:37:28.00000Z"}
- {"id":5,"name":"Impedit tempore illo velit saepe.", "descripcion": "sit aut dolore. Quia quia autem sed provident voluptatem.", "categ
- {"id":6,"name":"Dolore vitae sed fugit velit error qui velit et.", "d omnis pariatur. Sequi quis praesentium nobis accusantium non."}
- {"id":7,"name":"Voluptatum laboriosam dolorum voluptate quis t eaque aperiam voluptatum iure esse optio.", "categoria":"Dise\u00f1o web"}
- {"id":8,"name":"Nihil debitis reprehenderit debitis dicta rem iure odio provident aut accusamus nihil." "categoria":"Dise\u00f1o web"}

```
resources > views > cursos > 📄 index.blade.php > ...
1   @extends('layouts.plantilla')
2
3   @section('title','Cursos')
4
5   @section('content')
6
7       <h1>Bienvenidos a la página cursos</h1>
8       <ul>
9           @foreach ($cursos as $curso)
10              <li>{{$curso}}</li>
11          @endforeach
12
13
14      </ul>
15
16  @endsection
17
```

## 6. CRUD

Si quiero que solo imprima algo determinado, por ejemplo el **nombre**:

Nos imprimiría

### Bienvenidos a la página cursos

- Itaque nesciunt saepe ab vitae aut eveniet.
- Et rerum quae vitae nemo.
- Quis voluptate et veniam et.
- Nihil enim at quae et architecto earum laborum.
- Impedit tempore illo velit saepe.
- Dolore vitae sed fugit velit error qui velit et.
- Voluptatum laboriosam dolorum voluptate quis tempora.
- Nihil debitis reprehenderit debititis dicta rem iure.
- Dolorum autem aliquam et animi magni ipsum unde.
- Architecto pariatur distinctio sunt modi at velit quod.
- Delectus et aliquid consectetur velit quia in vel accusantium.
- Molestias non expedita adipisci dolorem.
- Natus nobis nemo enim qui quasi voluptas.
- Ut ipsa eos dolore sit rem aliquid et.
- Molestias ex dolor accusanti ut accusanti laboriosum.

resources > views > cursos > index.blade.php > ul

```
1  @extends('layouts.plantilla')
2
3  @section('title', 'Cursos')
4
5  @section('content')
6
7      <h1>Bienvenidos a la página cursos</h1>
8      <ul>
9          @foreach ($cursos as $curso)
10             <li>{{$curso->name}}</li>
11         @endforeach
12     </ul>
13
14
15     @endsection
16
```



## 6. CRUD

Sin embargo, nosotros ahora mismo solo tenemos 50 registros, si tuviéramos muchos más 50000, tardaría muchísimo en procesar.

Entonces deberíamos mostrar esos registros pero **paginados**.

Nos vamos a `CursoController` y escribimos:



```
public function index(){ //Hace referencia a cursos
    $cursos= Curso::paginate(); ←
    return view('cursos.index', compact('cursos'));
}
```

Nos muestra ahora solo 15 cursos:

Bienvenidos a la página cursos

- Itaque nesciunt saepe ab vitae aut eveniet.
- Et rerum quae vitae nemo.
- Quis voluptate et veniam et.
- Nihil enim at quae et architecto earum laborum.
- Impedit tempore illo velit saepe.
- Dolore vitae sed fugit velit error qui velit et.
- Voluptatum laboriosam dolorum voluptate quis tempora.
- Nihil debitis reprehenderit debitis dicta rem iure.
- Dolorum autem aliquam et animi magni ipsum unde.
- Architecto paratus distinctio sunt modi at velit quod.
- Delectus et aliquid consectetur velit quia in vel accusantium.
- Molestias non expedita adipisci dolorem.
- Natus nobis nemo enim qui quasi voluptas.
- Ut ipsa eos dolore sit rem aliquid et.
- Molestiae ex dolor occaecati ut occaecati laboriosam.

## 6. CRUD

Si yo quisiera ver los demás registros, en la URL debería agregar a la URL:

localhost/unidad7/public/cursos?page=2

Y así con todas las demás.

Evidentemente, no queremos ir poniendo eso en la URL y además el usuario no tiene por qué saber que hay más páginas.

Lo que nos pide el cuerpo es poner un botón y que se vayan pasando.

Nos vamos a index.blade.php

```
resources > views > cursos > 🗂 index.blade.php > ...
1   @extends('layouts.plantilla')
2
3   @section('title','Cursos')
4
5   @section('content')
6
7       <h1>Bienvenidos a la página cursos</h1>
8       <ul>
9           @foreach ($cursos as $curso)
10              <li>{{$curso->name}}</li>
11          @endforeach
12
13      </ul>
14
15      {{$cursos->links()}}
16
17  @endsection
```

## 6. CRUD

Si lo probamos:

Si bajamos un poco, vemos: 2 grandes flechas y la cantidad de páginas. Que si lo pulsamos, nos mueve de página.

Esto es porque tanto el número de páginas como las flechas tienen unos estilos preestablecidos que usan los estilos de la librería **Tailwind CSS**.

Lo veremos más adelante.

# Bienvenidos a la página cursos

- Nam sint dolorem est ea blanditiis deleniti.
- Rerum molestiae laboriosam neque exercitationem.
- Minus quae quos non animi.
- Inventore voluptatem aperiam quidem sit quibusdam voluptate.
- Porro eligendi qui enim doloribus rerum atque.
- Ad eum similique unde qui.
- Reprehenderit et consequatur beatae vel.
- Qui in illum totam maiores et.
- Rerum et sed voluptatibus facilis et.
- Perspiciatis sed eius dolores magnam facere sapiente.
- Vero ut est illum deserunt excepturi dolores.
- Ducimus enim mollitia ut qui labore.
- Ut nesciunt et ab suscipit.
- Voluptatem alias qui laborum aut.
- Qui et enim ex reiciendis ipsum voluptatem.

[« Previous](#) [Next »](#)

Showing 16 to 30 of 50 results

## 6. CRUD

Ahora lo que quiero es que debajo de Bienvenido a la página principal de cursos, me aparezca un enlace para crear cursos y me redirija a la página que creamos anteriormente: cursos/create.

En index.blade.php



Pero Laravel nos recomienda que hagamos otra cosa, también para evitar problemas y es ponerle un **nombre identificativo a las Rutas** (Route) que tenemos creadas, cómo hacemos esto: nos vamos a web.php

```
Route::get('/', HomeController::class);
```

```
Route::controller(CursoController::class)->group(function(){
    Route::get('cursos', 'index')->name('cursos.index');
    Route::get('cursos/create', 'create')->name('cursos.create');
    Route::get('cursos/{curso}', 'show')->name('cursos.show');
});
```

```
resources > views > cursos > index.blade.php > ...
1   @extends('layouts.plantilla')
2
3   @section('title','Cursos')
4
5   @section('content')
6
7       <h1>Bienvenidos a la página cursos</h1>
8       <a href= 'cursos/create'> Crear curso </a>
9       <ul>
10      @foreach ($cursos as $curso)
11          <li>{{$curso->name}}</li>
12      endforeach
13  
```

## 6. CRUD

En index.blade.php escribimos:



El método route y dentro de route el nombre de la ruta a la que hacemos referencia, en este caso cursos.create

De esta forma si modificamos la URL en un futuro, no tendremos inconvenientes y no tenemos que estar cambiando la URL en todos los archivos.

```
resources > views > cursos > index.blade.php > ...
1  @extends('layouts.plantilla')
2
3  @section('title','Cursos')
4
5  @section('content')
6
7      <h1>Bienvenidos a la página cursos</h1>
8      <a href= {{route('cursos.create')}}> Crear curso </a>
9      <ul>
10         @foreach ($cursos as $curso)
11             <li>{{$curso->name}}</li>
12         @endforeach
13
14     </ul>
15
16     {{$cursos->links()}}
17
18 @endsection
```

# 6. CRUD

## Enlaces de visualización

Ahora vamos a hacer que cada uno de los cursos sea un enlace y cuando haga click, me redirija a una página que diga:  
Bienvenido al curso *tal*.

Para ello, vamos a usar una url que ya hemos creado:

Vamos a `index.blade.php`  
Y escribimos:

```
resources > views > cursos > index.blade.php > ul
1   @extends('layouts.plantilla')
2
3   @section('title', 'Cursos')
4
5   @section('content')
6
7       <h1>Bienvenidos a la página cursos</h1>
8       <a href= "{{route('cursos.create')}}"> Crear curso </a>
9       <ul>
10          @foreach ($cursos as $curso)
11              <li>
12                  <a href="{{route('cursos.show', $curso->id)}}> {{$curso->name}}</a>
13              </li>
14          @endforeach
15      </ul>
16
17      {{$cursos->links()}}
18
19
20     @endsection
21
```

Al método `route` se le pasa la URL (`cursos.show`) y algo identificativo del registro, por ejemplo la `id`

## 6. CRUD

### Enlaces de visualización

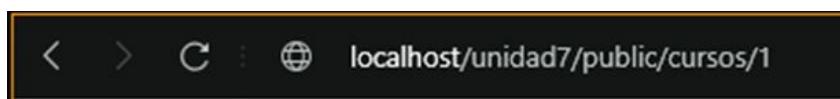


# Bienvenidos a la página cursos

[Crear curso](#)

- [Itaque nesciunt saepe ab vitae aut eveniet.](#)
- [Et rerum quae vitae nemo.](#)
- [Quis voluptate et veniam et.](#)
- [Nihil enim at quae et architecto earum laborum.](#)
- [Impedit tempore illo velit saepe.](#)
- [Dolore vitae sed fugit velit error qui velit et.](#)
- [Voluptatum laboriosam dolorum voluptate quis tempora.](#)
- [Nihil debitis reprehenderit debitis dicta rem iure.](#)
- [Dolorum autem aliquam et animi magni ipsum unde.](#)

Si pinchamos sobre el primero, por ejemplo:



# Bienvenido al curso: 1

## 6. CRUD

### Enlaces de visualización

Como estamos mandando el id, tendríamos que irnos a rutas y cambiar curso por id

```
Route::get('/', HomeController::class);

Route::controller(CursoController::class)->group(function(){
    Route::get('cursos', 'index')->name('cursos.index');
    Route::get('cursos/create', 'create')->name('cursos.create');
    Route::get('cursos/{curso}', 'show')->name('cursos.show');
});

|
```

```
Route::get('/', HomeController::class);

Route::controller(CursoController::class)->group(function(){
    Route::get('cursos', 'index')->name('cursos.index');
    Route::get('cursos/create', 'create')->name('cursos.create');
    Route::get('cursos/{id}', 'show')->name('cursos.show');
});

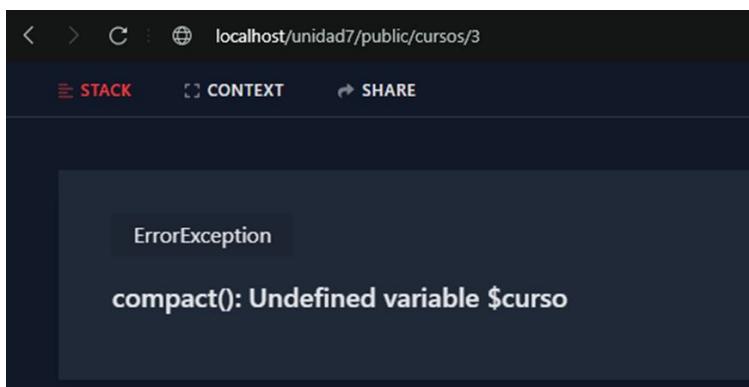
|
```

# 6. CRUD

## Enlaces de visualización

Y lo mismo en el controlador:

Pero qué pasa, que nos da un error si lo probamos:



```
pp > Http > Controllers > CursoController.php > CursoController > create
4
5  use App\Models\Curso;
6  use Illuminate\Http\Request;
7
8  class CursoController extends Controller
9  {
10
11     public function index(){ //Hace referencia a cursos
12
13         $cursos= Curso::paginate();
14         return view('cursos.index', compact('cursos'));
15     }
16
17
18     public function create(){ //Hace referencia a cursos/create
19         return view('cursos.create');
20     }
21
22
23     public function show($id){ //Hace referencia a cursos/muestra
24         return view('cursos.show', compact('curso'));
25     }
26 }
```

## 6. CRUD

### Enlaces de visualización

Lo que vamos a hacer es recuperar el registro por la id, cómo hacemos esto:

```
21
22
23     public function show($id){ //Hace referencia a cursos/muestra
24
25         $curso = Curso::find($id);
26         return view('cursos.show', compact('curso'));
27     }
28 }
```

Y si lo comprobamos:

## 6. CRUD

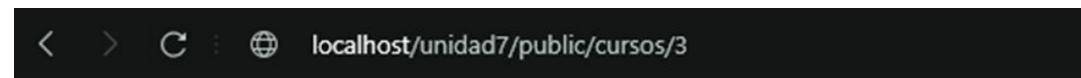
### Enlaces de visualización

Nos muestra el curso con sus características pero horribles.

Y esto es porque en show.blade.php

Estamos diciendo que me imprima la variable \$curso pero en este caso, es un objeto.

```
resources > views > cursos > 📄 show.blade.php > ...
1 @extends('layouts.plantilla')
2
3 @section('title','Curso' . $curso)
4
5 @section('content') <!-- content porque fue el nombre que
6 |       |
7 |       <h1>Bienvenido al curso: {{$curso}} </h1>
8 |
9 @endsection
10 |
```



**Bienvenido al curso: {"id":3,"name":"Q  
magni nesciunt est molestiae ipsa ipsa as  
laborum. Necessitatibus ut dicta nihil pe  
magnam debitis ipsa inventore.","catego  
16T13:37:28.000000Z","updated\_at":"2**

Y yo solo quiero que me imprima el nombre:

```
resources > views > cursos > 📄 show.blade.php > ...
1 @extends('layouts.plantilla')
2
3 @section('title','Curso' . $curso)
4
5 @section('content') <!-- content porque fue el nombre que
6 |       |
7 |       <h1>Bienvenido al curso: {{$curso->name}} </h1>
8 |
9 @endsection
10 |
```

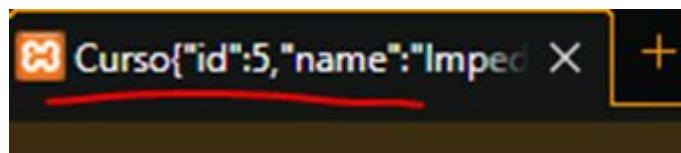
## 6. CRUD

### Enlaces de visualización

Si lo probamos:



Si nos fijamos, también hay que corregirlo en el title:



```
resources > views > cursos > 🗂 show.blade.php > ...
1  @extends('layouts.plantilla')
2
3  @section('title','Curso' . $curso->name)
4
5  @section('content') <!-- content porque fue el nombre
6
7      <h1>Bienvenido al curso: {{$curso->name}} </h1>
8
9  @endsection
10 |
```

## 6. CRUD

### Enlaces de visualización

Como al final lo que tenemos es un objeto, podemos también hacer que se nos imprima otra información, por ejemplo la categoría (línea 8) y la descripción (línea 9)

```
resources > views > cursos > show.blade.php > ...
1  @extends('layouts.plantilla')
2
3  @section('title','Curso' . $curso->name)
4
5  @section('content') <!-- content porque fue el nombre que puse
6
7      <h1>Bienvenido al curso: {{$curso->name}} </h1>
8  →     <p><strong>Categoría:</strong> {{$curso->categoria}}</p>
9  →     <p>{{$curso->descripcion}}</p>
10    @endsection
```

< > C : localhost/unidad7/

## Bienvenido al curso: Imp

Categoría: Diseño web

Aut qui natus aut assumenda. Distinctio excepturi velit

## 6. CRUD

### Enlaces de visualización

Hilando un poco más, si queremos poner un **botón** que vuelva a la lista de cursos, podríamos poner:

```
resources > views > cursos > 📄 show.blade.php > ...
1   @extends('layouts.plantilla')
2
3   @section('title','Curso' . $curso->name)
4
5   @section('content') <!-- content porque fue el nombre que pus
6
7   →     <h1>Bienvenido al curso: {{$curso->name}} </h1>
8   →     <a href="{{route('cursos.index')}}"> Volver a cursos</a>
9     <p><strong>Categoría:</strong> {{$curso->categoria}}</p>
10    <p>{{$curso->descripcion}}</p>
11    @endsection
```

# Bienvenido al curs

[Volver a cursos](#)

Categoría: Diseño web

Praesentium ut deserunt asperiores reru

## 6. CRUD

Vamos a ver cómo agregar y actualizar registros de nuestra BD desde un formulario. Esto lo habíamos hecho pero usando la terminal de Tinker.

### Agregar registros

Lo que queremos hacer es que cuando pinchemos sobre Crear nuevo curso, en la página nos aparezca un formulario. Y la información de ese formulario será la que almacenemos en nuestra base de datos.

Primero creamos el formulario

Abrimos `create.blade.php` y creamos un formulario:

# 6. CRUD

## Agregar registros

Formulario:



Si lo probamos



**En esta página podrás crear un curso**

Nombre:

Descripción:

Categoría:

```
resources > views > cursos > create.blade.php > ...
4
5  @section('content') <!-- content porque fue el nombre que puse en 1
6
7      <h1>En esta página podrás crear un curso</h1>
8      <form action="">
9          <label>
10             Nombre:
11             <br>
12             <input type="text" name="name">
13         </label>
14
15         <br>
16         <label>
17             Descripción:
18             <br>
19             <textarea name="descripcion" rows="5"></textarea>
20         </label>
21
22         <br>
23         <label>
24             Categoría:
25             <br>
26             <input type="text" name="categoria">
27         </label>
28     </form>
29
30 @endsection
```

## 6. CRUD

### Agregar registros

La información que se rellene, se va a mandar a algún lado para que se procese y para que se pueda guardar en la base de datos. Esto se indicará en el action del form.

Vamos a crear la ruta:

Routes > web.php

Usamos post porque es información que se va a mandar a través de un formulario.

Hemos indicado que el que se va a encargar es el método store pero aún no lo hemos creado.

```
15 |
16 */
17
18 Route::get('/', HomeController::class);
19
20 Route::controller(CursoController::class)->group(function(){
21     Route::get('cursos', 'index')->name('cursos.index');
22     Route::get('cursos/create', 'create')->name('cursos.create');
23     Route::post('cursos', 'store')->name('cursos.store');
24     Route::get('cursos/{id}', 'show')->name('cursos.show');
25 });
26 });
```

## 6. CRUD

### Agregar registros

Lo creamos: nos vamos al controlador:  
CursoController y debajo de create:



```
public function store(){  
    //  
}
```

En  
create.blade.php,  
le indicamos la ruta:



```
resources > views > cursos > create.blade.php > ...  
4  
5     @section('content') <!-- content porque fue el nombre que puse en la plantilla -->  
6  
7         <h1>En esta página podrás crear un curso</h1>  
8         <form action="{{route('cursos.store')}}" method = "POST">  
9             <label>  
10                Nombre:  
11                <br>  
12                <input type="text" name="name">
```



## 6. CRUD

### Agregar registros

Además, debajo del formulario,  
agregamos un botón



```
28 |         <br>
29 |     <button type="submit">Enviar formulario</button>
30 |   </form>
31 |
32 @endsection
33 |
```

Pero si lo probamos:



419 | PAGE EXPIRED

## 6. CRUD

### Agregar registros

Esto es porque Laravel exige que cada vez que enviamos información a través del método post, enviamos también un token junto al formulario por seguridad.

Cómo se hace esto:

Con la directiva de Blade:  
@csrf que lo que hace es agregarle un input oculto con un nombre llamado token y se va a encargar de generar un token.

```
resources > views > cursos > 📄 create.blade.php > ⚙ form
  1 @extends('layouts.plantilla')
  2
  3 @section('title','Cursos create')
  4
  5 @section('content') <!-- content porque fue el nombre que puse en
  6
  7     <h1>En esta página podrás crear un curso</h1>
  8     <form action="{{route('cursos.store')}}" method ="POST"> |
  9      @csrf
 10     <label>
 11         Nombre:
 12     </label>
 13     <input type="text" name="name">
 14     <button type="submit" value="Crear">Crear</button>
 15 </form>
 16
 17 </div>
 18 </div>
```

## 6. CRUD

### Agregar registros

Cómo recuperar la información que estoy mandando:

Voy a CursoController.php y pongo:

```
21     }
22
23     public function store(Request $request){ //cualquier cosa que se envíe va a hacer referencia a request
24         return $request->all();
25
26     }
```

Nos sale:

```
{"_token":"Alvkm9zywgUtkRpJvhfJ0h6EPzS4rlN5gZfQ4Abe","name":"Css","descripcion":"asdad","categoria":"Desarrollo web"}
```

## 6. CRUD

### Agregar registros

Una vez recuperemos esto, podemos crear un nuevo registro, como hacemos esto:

```
$curso = new Curso();
```

Y quiero que el nombre del curso sea igual a lo que se ha enviado en el formulario con el nombre name.

Y hacemos esto por cada parte del formulario:



```
public function store(Request $request){ //cualquier  
    $curso = new Curso();  
  
    $curso->name = $request->name;  
    $curso->descripcion = $request->descripcion;  
    $curso->categoria = $request->categoria;  
  
    return $curso; |
```

Y nos sale:

```
{"name": "Css", "descripcion": "asdad", "categoria": "Desarrollo web"}
```

## 6. CRUD

### Agregar registros

Lo que vamos a hacer que se guarde, entonces:

Si lo volvemos a actualizar, no nos sale nada.

Pero si nos vamos a la base de datos:

```
public function store(Request $request){ //cualquier  
    $curso = new Curso();  
  
    $curso->name = $request->name;  
    $curso->descripcion = $request->descripcion;  
    $curso->categoria = $request->categoria;  
  
    $curso->save();  
}
```

← ↑ →	v	id	name	descripcion	categoria	created_at	updated_at		
□	Editar	Copiar	Borrar	51	Css	asdad	Desarrollo web	2023-02-17 18:07:03	2023-02-17 18:07:03

De esta forma, hemos logrado almacenar en nuestra base de datos un nuevo registro cuya información se mandó desde un formulario.

## 6. CRUD

### Agregar registros

Lo que vamos a hacer ahora es que después de mandar la información del formulario, no se quede en blanco, si no que nos redirija hacia alguna página.

Lo primero que tengo que tener en cuenta es **hacia dónde quiero redirigirlo**. Por ejemplo, vamos a hacer que, una vez creado, nos mande al registro ya creado.

Si lo probamos:

```
public function store(Request $request){ //cualquier cosa
    $curso = new Curso();

    $curso->name = $request->name;
    $curso->descripcion = $request->descripcion;
    $curso->categoria = $request->categoria;
    $curso->save();

    return redirect()->route('cursos.show', $curso);
}
```

## 6. CRUD

### Agregar registros

En esta página p...

Nombre:

Descripción:

El mejor framework de  
PHP

Categoría:



## Bienvenido al curso: laravel

[Volver a cursos](#)

**Categoría:** Desarrollo web

El mejor framework de PHP

## 6. CRUD

### Actualizar registros

Ahora vamos a ver cómo actualizar el registro que tenemos

Lo que vamos a hacer es crear un enlace que diga **editar curso** y cuando hagamos click, nos redirija a un formulario similar al que utilizamos para crearlo pero con la diferencia de que ya va a estar relleno con los datos para poder modificarlo.

```
resources > views > cursos > 🐘 show.blade.php > ...
1  @extends('layouts.plantilla')
2
3  @section('title','Curso' . $curso->name)
4
5  @section('content') <!-- content porque fue el nombre que puse en la pl...
6
7      <h1>Bienvenido al curso: {{$curso->name}} </h1>
8      <a href="{{route('cursos.index')}}"> Volver a cursos</a>
9      <br>
10     ↗<a href="{{ route('cursos.edit', $curso) }}>Editar curso</a>
11
```

Vamos a show.blade.php

## 6. CRUD

### Actualizar registros

Ahora **creamos la ruta**.  
Nos vamos a web.php



```
18     Route::get('/', HomeController::class);
19
20     Route::controller(CursoController::class)->group(function(){
21         Route::get('cursos', 'index')->name('cursos.index');
22         Route::get('cursos/create', 'create')->name('cursos.create');
23         Route::post('cursos', 'store')->name('cursos.store');
24         Route::get('cursos/{id}', 'show')->name('cursos.show');
25         Route::get('cursos/{id}/edit', 'edit')->name('cursos.edit');
26
27     });
28
```

Y como hemos introducido edit,  
creamos el **método edit** en el  
controlador. Nos vamos a  
CursoController y:



```
40
41     public function edit($id) {
42         $curso = Curso::find($id);
43         return $curso;
44     }
45 }
```

## 6. CRUD

### Actualizar registros

Podemos re-escribir el método de la siguiente manera:

Donde Laravel entiende que quiero que ese \$id sea una instancia de la clase Curso cuyo id sea lo que estoy pasando por la URL.

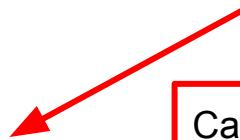
```
public function show($id){ //Hace referencia a cursos/muestra  
    $curso = Curso::find($id);  
    return view('cursos.show', compact('curso'));  
}  
  
public function edit($id) {  
    $curso = Curso::find($id);  
    return $curso;  
}  
}
```

```
}
```

```
public function show(Curso $curso){ //Hace referencia a cursos/muestra  
    return view('cursos.show', compact('curso'));  
}  
  
public function edit(Curso $curso) {  
    return view('cursos.edit', compact('curso'));  
}
```

Cambio id por  
Curso \$curso



¡Y ahorraremos código!

## 6. CRUD

### Actualizar registros

Y en la ruta, lo cambiamos también:

```
Route::controller(CursoController::class)->group(function(){
    Route::get('cursos', 'index')->name('cursos.index');
    Route::get('cursos/create', 'create')->name('cursos.create');
    Route::post('cursos', 'store')->name('cursos.store');
    Route::get('cursos/{id}', 'show')->name('cursos.show');
    Route::get('cursos/{id}/edit', 'edit')->name('cursos.edit');

});
```



```
Route::controller(CursoController::class)->group(function(){
    Route::get('cursos', 'index')->name('cursos.index');
    Route::get('cursos/create', 'create')->name('cursos.create');
    Route::post('cursos', 'store')->name('cursos.store');
    Route::get('cursos/{curso}', 'show')->name('cursos.show');
    Route::get('cursos/{curso}/edit', 'edit')->name('cursos.edit');

});
```

# 6. CRUD

## Actualizar registros

Ahora mismo solo nos muestra el registro. Pero yo quiero que me **edite lo que tengo:**

Vamos a crear la vista:  
edit.blade.php y aquí va a haber lo mismo que en la vista create.blade

Copiamos y pegamos.

Y modificamos:



```
1 @extends('layouts.plantilla')
2
3 @section['title', 'Cursos edit']
4
5 @section('content')
6
7 <h1>En esta página podrás editar un curso</h1>
8 <form action="{{route('cursos.store')}}" method ="POST">
9     @csrf
10    <label>
11        Nombre:
12        <br>
13        <input type="text" name="name" value="{{$curso->name}}>
14    </label>
15
16    <br>
17    <label>
18        Descripción:
19        <br>
20        <textarea name="descripcion" rows="5" >{{$curso->descripcion}}</textarea>
21    </label>
22
23    <br>
24    <label>
25        Categoría:
26        <br>
27        <input type="text" name="categoria" value="{{$curso->categoria}}>
28    </label>
29    <br>
30    <button type="submit">Actualizar formulario</button>
31
32
33 @endsection
```

## 6. CRUD

### Actualizar registros

Ahora lo que tenemos que crear es una ruta: web.php

```
18     Route::get('/', HomeController::class);
19
20     Route::controller(CursoController::class)->group(function(){
21         Route::get('cursos', 'index')->name('cursos.index');
22         Route::get('cursos/create', 'create')->name('cursos.create');
23         Route::post('cursos', 'store')->name('cursos.store');
24         Route::get('cursos/{curso}', 'show')->name('cursos.show');
25         Route::get('cursos/{curso}/edit', 'edit')->name('cursos.edit');
26         Route::put('cursos/{curso}', 'update')->name('cursos.update');
27     });
28 
```

Usamos el método put en este caso porque Laravel nos recomienda que, si es un archivo de editar, sea mejor put que get o post.

## 6. CRUD

### Actualizar registros

Como hemos puesto la ruta, tengo que apuntar a esta ruta desde el formulario. Nos vamos al formulario edit.blade.php:

```
@section('content')

    <h1>En esta página podrás editar un curso</h1>
    <form action="{{route('cursos.update', $curso)}}" method = "POST">

        @csrf
        @method('put') ←
```

Utilizamos la directiva de Blade @method porque en teoría tendríamos que poner: "method=PUT" pero html solo conoce dos directivas, get y post. Por eso, tenemos que usar @method

## 6. CRUD

### Actualizar registros

Por último, tenemos que crear el método update en el controlador. Nos vamos a CursoController.php

```
42
43     public function update(Request $request, Curso $curso) {
44         $curso->name = $request->name;
45         $curso->description= $request->description;
46         $curso->categoria= $request->categoria;
47         $curso->save();
48         return redirect()->route('cursos.show', $curso);
49     }
50 }
```

## 6. CRUD

### Validar formularios

Ahora mismo, si tenemos un formulario y no lo rellenamos, nos sale el siguiente error:

unidad7/public/cursos/create

Para evitar esto, tenemos que crear una **validación**.

Que lo que hace es comprobar si los datos que ha introducido el usuario son correctos y en caso de que no, vuelva al formulario y nos diga dónde está el error.

The screenshot shows a Laravel error page with the following details:

- URL: localhost/unidad7/public/cursos
- Buttons: BACK, CONTEXT, SHARE
- Exception Type: Illuminate\Database\QueryException
- Message: SQLSTATE[23000]: Integrity constraint violation
- Stack Trace: INSERT INTO `cursos` (`name`, `descripcion`, `categoria`) VALUES ('', ' ', '')

## 6. CRUD

### Validar formularios

Para ello, nos vamos a `CursoController.php` y en el método `store` del controlador agregamos:

```
22
23     public function store(Request $request){ //cualquier cosa
24
25         $request->validate([
26             'name' => 'required|max:10',
27             'descripcion' => 'required|min:10',
28             'categoria' => 'required'
29         ]);
30     }
```

**Regla de validación**

Si alguna regla de validación que acabamos de poner falla se detiene el flujo del programa y nos retoma nuevamente al formulario.

En este caso, se requiere el nombre (con un máximo de 10 caracteres), la descripción (con un mínimo de 10 caracteres) y la categoría. Para poner diferentes validaciones se usa |

## 6. CRUD

### Validar formularios

Para que el usuario sepa lo que ha pasado, debajo de los campos debe aparecer un mensaje.

Para ello, usamos la directiva de Blade. @error @enderror que actúa como un if.

Nos vamos a la vista, en este caso: `create.blade.php` y debajo de la etiqueta de nombre, ponemos:



```
@section('content') <!-- content porque fue el nombre que p  
  
<h1>En esta página podrás crear un curso</h1>  
<form action="{{route('cursos.store')}}" method ="POST">  
    @csrf  
    <label>  
        Nombre:  
        <br>  
        <input type="text" name="name">  
    </label>  
  
    @error('name')  
        <br>  
        <small>*{{$message}}</small>  
        <br>  
    @enderror
```

## 6. CRUD

### Validar formularios

Si lo comprobamos:

En esta página p

Nombre:

\*The name field is required.

Descripción:

Categoría:

Enviar formulario

## 6. CRUD

### Validar formularios

Para evitar perder los datos que ya se han rellenado en el formulario, cuando la validación falla, actualizamos la vista `create.blade.php` y escribimos:

```
8   <form action= "{{route('cursos.store')}}" method = POST >
9     @csrf
10    <label>
11      Nombre:
12      <br>
13      <input type="text" name="name" value={{old('name')}}>
14    </label>
15  |
```

## 6. CRUD

### Validar formularios

```
<label>
    Descripción:
    <br>
    <textarea name="descripcion" rows="5">{{old('descripcion')}}</textarea>
</label>
```

```
<label>
    Categoría:
    <br>
    <input type="text" name="categoria" value="{{old('categoria')}}"> 
</label>
```

Con esto, ya tendríamos validado nuestro formulario.

## 6. CRUD

### Validar formularios

Vamos a hacer lo mismo para el formulario de editar.

Nos vamos a CursoController y la regla de validación de la función store la copiamos en la función update



Nos vamos a la vista de editar:  
edit.blade.php

```
public function update(Request $request, Curso $curso) {  
  
    $request->validate([  
        'name' => 'required|max:10',  
        'descripcion' => 'required|min:10',  
        'categoria' => 'required'  
    ]);  
  
    $curso->name = $request->name;  
    $curso->descripcion= $request->descripcion;  
    $curso->categoria= $request->categoria;  
    $curso->save();  
    return redirect()->route('cursos.show', $curso);  
}
```

Y copiamos las directivas de Blade @error @enderror de antes, debajo de cada etiqueta.

Al igual que antes, para evitar que los datos que estén, se vayan cuando la validación falla, tenemos que poner:

## 6. CRUD

### Validar formularios

```
<label>
    Nombre:
    <br>
    <input type="text" name="name" value="{{old('name', $curso->name)}}">
</label>
```

```
1>
Descripción:
<br>
<textarea name="descripcion" rows="5" >{{old('descripcion', $curso->descripcion)}}</textarea>
el>
```

## 6. CRUD

### Validar formularios

```
Categoría:  
<br>  
<input type="text" name="categoria" value="{{old('categoria',$curso->categoria)}}>  
>
```

## 6. CRUD

### Form Request

Para crear validaciones más complejas, tenemos que crear un **Form Request** (Solicitud de formulario)

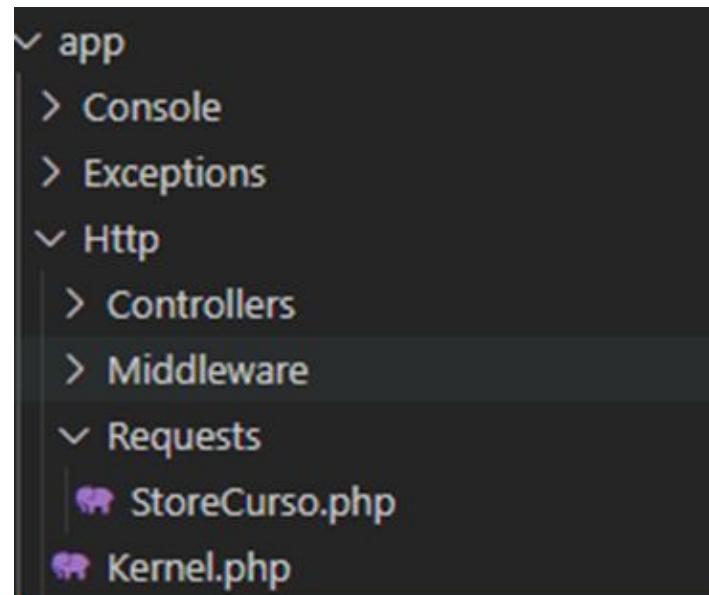
Para crearlas: `php artisan make:request StoreCurso`

Lo va a colocar dentro de `app > Http > Request`

Si lo abrimos:

Vemos que hay 2 métodos **authorize** y **rules**

En **authorize** debe contener la lógica necesaria para verificar si el usuario que está intentando ingresar a nuestra BD tiene los permisos necesarios para hacerlos. (administrador, creador de contenido, editor, etc).



## 6. CRUD

### Form Request

Lo vamos a dejar por ahora. Solo que cambiamos false por true.

```
15  ,
14  public function authorize()
15  {
16      return true;
17  }
```

Nos vamos a `CursoController` y la validación de la función `store` la cortamos y la pegamos en la `function rules` de `StoreCurso.php`

```
24  public function rules()
25  {
26      return [
27          'name' => 'required|max:10',
28          'descripcion' => 'required|min:10',
29          'categoria' => 'required'
30      ];
31  }
32
33 }
```

## 6. CRUD

### Form Request

Y ahora, agregamos en el controlador:

Y en su método store cambiamos  
el tipo Request por StoreCurso

```
pp > Http > Controllers > CursoController.php > ...
1   <?php
2
3   namespace App\Http\Controllers;
4
5   use App\Models\Curso;
6   use Illuminate\Http\Request;
7   use App\Http\Requests\StoreCurso;
8
```

```
24  public function store(StoreCurso $request){ //cualq
25
26
27   $curso = new Curso();
28
29   $curso->name = $request->name;
30   $curso->descripcion = $request->descripcion;
31   $curso->categoria = $request->categoria;
```

De esta forma \$request recibe todos los datos del formulario y también hace las validaciones. Todas las reglas de validación irán en un archivo aparte: StoreCurso.

## 6. CRUD

### Form Request

Para validaciones más complejas:

Por ejemplo, para traducir/cambiar los mensajes de validación, creamos en StoreCurso.php

Lo que hace es que cuando falla el 'name', me devuelve y pone 'nombre del curso'

```
33
34     public function attributes() {
35         return [
36             'name' => 'nombre del curso',
37         ];
38     }
39 }
```

Nombre:  
  
\*The nombre del curso field is required.

Descripción:  
  
\*The descripcion field is required.

Categoría:  
  
\*The categoria field is required.

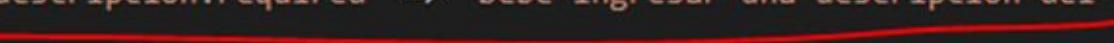
Enviar formulario

## 6. CRUD

### Form Request

Para generar un mensaje más personalizado, agregamos también otro método, message() indicando el elemento y la validación

```
34     public function attributes() {
35         return [
36             'name' => 'nombre del curso',
37         ];
38     }
39
40     public function messages() {
41         return [
42             'descripcion.required' => 'Debe ingresar una descripción del curso',
43         ];
44     }
45
46 }
```



## 6. CRUD

### Asignación masiva

Por ejemplo, si nos vamos a:

App>http>Controller  
CursoController.php

Vemos que en la función store



```
public function store(StoreCurso $request){ //cualquier cosa  
  
    $curso = new Curso();  
  
    $curso->name = $request->name;  
    $curso->descripcion = $request->descripcion;  
    $curso->categoria = $request->categoria;  
    $curso->save();  
  
    return redirect()->route('cursos.show', $curso);
```

Tenemos varias asignaciones, en el que se mandan 3 campos.

Pero imaginemos un formulario con 40 propiedades, tendríamos que agregar 40 propiedades a ese objeto.  
Sería horrible.

Laravel con la asignación masiva nos permite reemplazar ese código por:

## 6. CRUD

### Asignación masiva

```
public function store(StoreCurso $request){ //cualquier cosa que se envíe va  
/*  
$curso = new Curso();  
  
$curso->name = $request->name;  
$curso->descripcion = $request->descripcion;  
$curso->categoria = $request->categoria;  
$curso->save();  
  
*/  
$curso = Curso::create($request->all());  
return redirect()->route('cursos.show', $curso);  
}
```

## 6. CRUD

### Asignación masiva

Por cuestiones de seguridad, para que no se agreguen campos no deseados en nuestra tabla, debemos indicarle al modelo cuáles son las propiedades deseadas (autorizadas) para crear el objeto. Eso lo hacemos:

Agregando en  
app\Models\Curso.php

```
app > Models > Curso.php > ...
1  <?php
2
3  namespace App\Models;
4
5  use Illuminate\Database\Eloquent\Factories\HasFactory;
6  use Illuminate\Database\Eloquent\Model;
7
8  class Curso extends Model
9  {
10     use HasFactory;
11     protected $fillable = [ 'name', 'descripcion', 'categoria' ];
12
13
14 }
```

## 6. CRUD

### Asignación masiva

Si la cantidad de propiedades es mucha, tendríamos en `Curso.php` muchísimo código.

En este caso, podemos optar por usar:



Podemos dejar el array vacío y aún así nos permitirá la asignación masiva.

```
app > Models > Curso.php > ...
1  <?php
2
3  namespace App\Models;
4
5  use Illuminate\Database\Eloquent\Factories\HasFactory;
6  use Illuminate\Database\Eloquent\Model;
7
8  class Curso extends Model
9  {
10     use HasFactory;
11     protected $guarded = [ 'status' ];
12
13 }
```

## 6. CRUD

### Asignación masiva

Ahora, en CursoController.php

```
38 ✓    public function update(Request $request, Curso $curso) {  
39  
40 ✓        $request->validate([  
41            'name' => 'required|max:10',  
42            'descripcion' => 'required|min:10',  
43            'categoria' => 'required'  
44        ]);  
45  
46        /*  
47            $curso->name = $request->name;  
48            $curso->descripcion= $request->descripcion;  
49            $curso->categoria= $request->categoria;  
50            $curso->save();  
51  
52        */  
53  
54        $curso->update($request->all());  
55        return redirect()->route('cursos.show', $curso);  
56    }  
57
```

## 6. CRUD

### Eliminar un registro de la base de datos

Lo primero que tenemos que hacer es crear una ruta que se va a encargar de eliminar ese registro.

Nos vamos a web.php

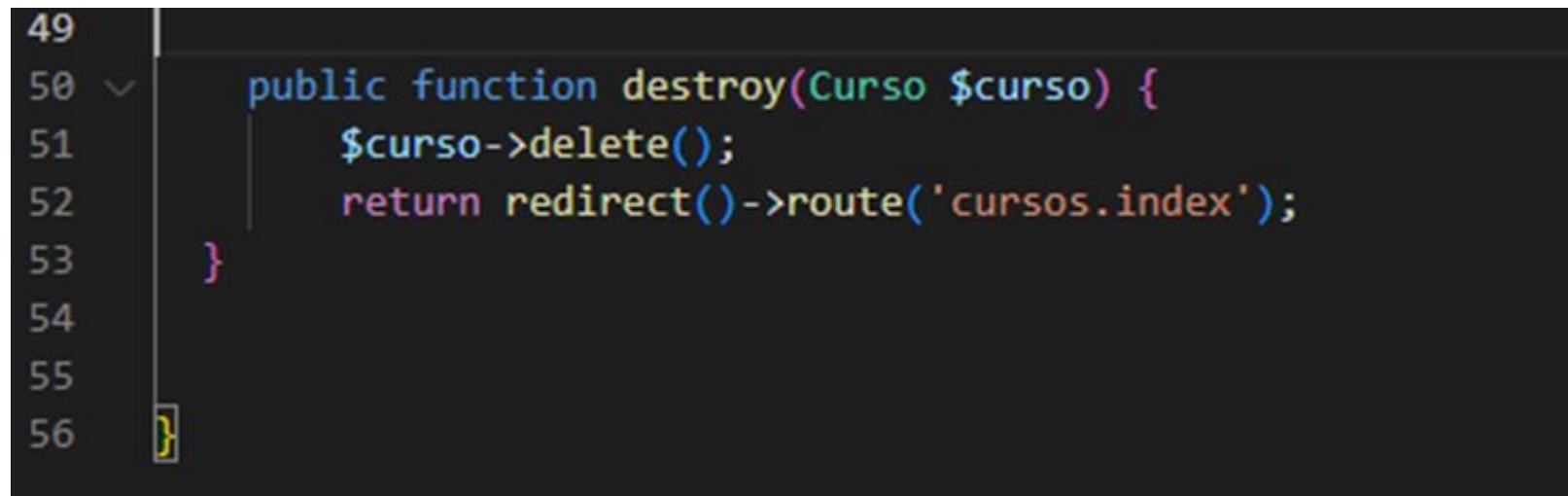
```
20 Route::controller(CursoController::class)->group(function(){
21     Route::get('cursos', 'index')->name('cursos.index');
22     Route::get('cursos/create', 'create')->name('cursos.create');
23     Route::post('cursos', 'store')->name('cursos.store');
24     Route::get('cursos/{curso}', 'show')->name('cursos.show');
25     Route::get('cursos/{curso}/edit', 'edit')->name('cursos.edit');
26     Route::put('cursos/{curso}', 'update')->name('cursos.update');
27     Route::delete('cursos/{curso}', 'destroy')->name('cursos.destroy');
28 });
29 
```



## 6. CRUD

### Eliminar un registro de la base de datos

Como hemos creado una ruta, tenemos que crear el método en el controlador que se va a encargar de procesar esa ruta. Nos vamos a CursoController:



```
49
50     public function destroy(Curso $curso) {
51         $curso->delete();
52         return redirect()->route('cursos.index');
53     }
54
55
56 }
```

The screenshot shows a code editor with a dark theme. It displays a portion of the `CursoController.php` file. The `destroy()` method is highlighted, showing its implementation. The code uses PHP syntax with annotations for variables (`$curso`) and routes (`'cursos.index'`). Line numbers 49 through 56 are visible on the left side of the editor.

## 6. CRUD

### Eliminar un registro de la base de datos

Y ahora nos tenemos que ir a la vista (show.blade.php) para poner un botón que diga, eliminar el curso:

Lo hacemos a través de un formulario porque en la ruta estamos utilizando el método delete.

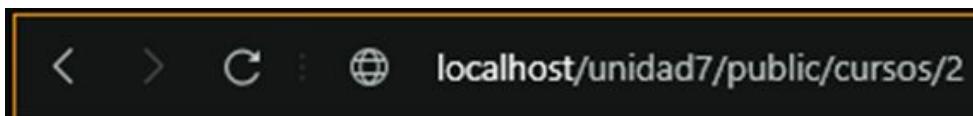
Si usáramos un botón, estaríamos usando get.

```
14
15  <form action="{{ route('cursos.destroy', $curso) }}" method="POST">
16    @csrf
17    → @method('delete') !
18    <button type="submit">Eliminar</button>
19  </form>
20
21
22 @endsection
```

## 6. CRUD

### Eliminar un registro de la base de datos

Si lo probamos:



# Bienvenido al curso: Et re

[Volver a cursos](#)

[Editar curso](#)

**Categoría:** Diseño web

Praesentium ut deserunt asperiores rerum et animi earum

[Eliminar](#)

## 6. CRUD

### Cómo crear rutas con route resource

En web.php

8 rutas:

La primera es la que se encarga de la página principal y las otras 7 de nuestro CRUD de Cursos.

Pero cuando nosotros hagamos un proyecto, vamos a generar más de un CRUD, entonces vamos a generar muchísimas rutas.

Por lo que el archivo estaría cargado.

Esto se puede solucionar.

```
18 Route::get('/', HomeController::class);
19
20 Route::controller(CursoController::class)->group(function(){
21     Route::get('cursos', 'index')->name('cursos.index');
22     Route::get('cursos/create', 'create')->name('cursos.create');
23     Route::post('cursos', 'store')->name('cursos.store');
24     Route::get('cursos/{curso}', 'show')->name('cursos.show');
25     Route::get('cursos/{curso}/edit', 'edit')->name('cursos.edit');
26     Route::put('cursos/{curso}', 'update')->name('cursos.update');
27     Route::delete('cursos/{curso}', 'destroy')->name('cursos.destroy');
28 });
29 );
```

## 6. CRUD

### Cómo crear rutas con route resource

Si abrimos consola y escribimos: php artisan route:list

```
PS C:\xampp\htdocs\unidad7> php artisan route:list
```

GET HEAD	/	.....	HomeController
POST	_ignition/execute-solution	.....	ignition.executeSolution > Spatie\LaravelIgnition > ExecuteSolutionController
GET HEAD	_ignition/health-check	.....	ignition.healthCheck > Spatie\LaravelIgnition > HealthCheckController
POST	_ignition/update-config	.....	ignition.updateConfig > Spatie\LaravelIgnition > UpdateConfigController
GET HEAD	api/user	.....	
GET HEAD	cursos	.....	cursos.index > CursoController@index
POST	cursos	.....	cursos.store > CursoController@store
GET HEAD	cursos/create	.....	cursos.create > CursoController@create
GET HEAD	cursos/{curso}	.....	cursos.show > CursoController@show
PUT	cursos/{curso}	.....	cursos.update > CursoController@update
DELETE	cursos/{curso}	.....	cursos.destroy > CursoController@destroy
GET HEAD	cursos/{curso}/edit	.....	cursos.edit > CursoController@edit
GET HEAD	sanctum/csrf-cookie	.....	sanctum.csrf-cookie > Laravel\Sanctum > csrfCookieController@show

## 6. CRUD

### Cómo crear rutas con route resource

Si escribimos en web.php el siguiente código:



Estamos resumiendo 7 rutas que se encargan de curso, en una sola.

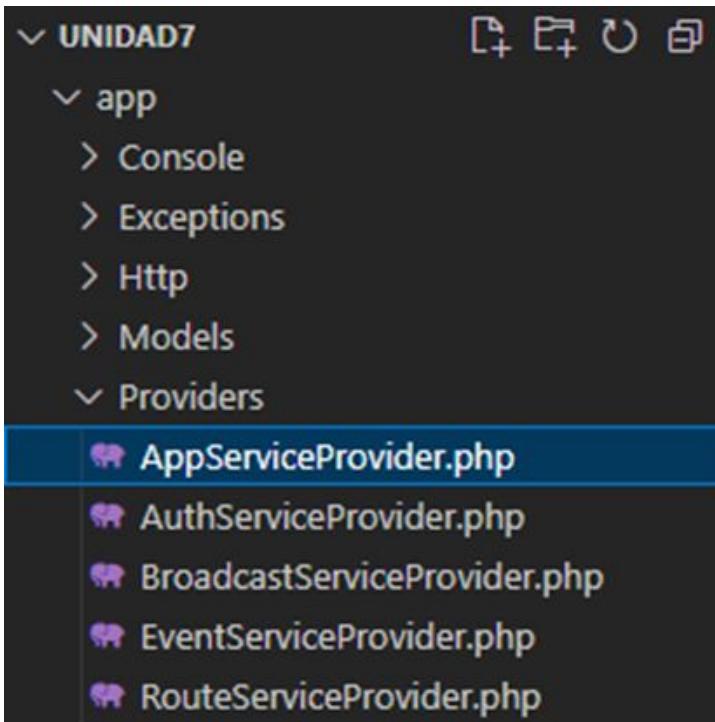
Esa única línea de código se encarga de administrar todas las rutas de Curso.

Y esto es por las convenciones de Laravel.

```
routes > web.php
12 | Here is where you can register web routes for your application. These
13 | routes are loaded by the RouteServiceProvider within a group which
14 | contains the "web" middleware group. Now create something great!
15 |
16 */
17
18 Route::get('/', HomeController::class);
19
20 { /*Route::controller(CursoController::class)->group(function(){
21     Route::get('cursos', 'index')->name('cursos.index');
22     Route::get('cursos/create', 'create')->name('cursos.create');
23     Route::post('cursos', 'store')->name('cursos.store');
24     Route::get('cursos/{curso}', 'show')->name('cursos.show');
25     Route::get('cursos/{curso}/edit', 'edit')->name('cursos.edit');
26     Route::put('cursos/{curso}', 'update')->name('cursos.update');
27     Route::delete('cursos/{curso}', 'destroy')->name('cursos.destroy');
28 });
29 */
30
31 Route::resource('cursos', CursoController::class);
32
33 |
```

## 6. CRUD

### Cómo crear rutas con route resource



Y agregamos:

```
app > Providers > AppServiceProvider.php > ...
1  <?php
2
3  namespace App\Providers;
4
5  use Illuminate\Support\ServiceProvider;
6  use Illuminate\Support\Facades\Route;
7
```

Y dentro del método boot();

```
25  public function boot()
26  {
27      Route::resourceVerbs([
28          'create' => 'crear',
29          'edit' => 'editar'
30      ]);
31 }
```

## 6. CRUD

### Cómo crear rutas con route resource

Si lo probamos: php artisan route:list

GET HEAD	/	
POST	_ignition/execute-solution	ignition.executeSolution > Spatie\LaravelIgnition\Execute
GET HEAD	_ignition/health-check	ignition.healthCheck > Spatie\LaravelIgnition\H
POST	_ignition/update-config	ignition.updateConfig > Spatie\LaravelIgnition\Up
GET HEAD	api/user	
GET HEAD	cursos	cursos.index > C
POST	cursos	cursos.store > C
GET HEAD	cursos/crear	cursos.create > Cu
GET HEAD	cursos/{curso}	cursos.show > I
PUT PATCH	cursos/{curso}	cursos.update > Cu
DELETE	cursos/{curso}	cursos.destroy > Cur
GET HEAD	cursos/{curso}/editar	cursos.edit >
GET HEAD	sanctum/csrf-cookie	sanctum.csrf-cookie > Laravel\Sanctum\CsrfC

## 6. CRUD

### Cómo crear rutas con route resource

< > C



localhost/unidad7/public/cursos/crear

< > C :



localhost/unidad7/public/cursos/9/editar

# En esta página podrás

Nombre:

Descripción:

# En esta página podrás editar tu

Nombre:

Descripción:

Categoría:

## 7. Buenas prácticas

### Generar URLs amigables

Por ejemplo, ahora mismo si probamos y nos metemos en cualquier curso:

Esto es poco descriptivo tanto para nosotros como para los buscadores.

Vamos a cambiar esto



A screenshot of a web browser window. The address bar at the top shows a URL: "localhost/unidad7/public/cursos/3". This URL is circled in red. Below the address bar, the main content area displays the text "Bienvenido al curso: Quis v". Underneath this, there are two blue hyperlinks: "Volver a cursos" and "Editar curso". Further down, the text "Categoría: Desarrollo web" is displayed. At the bottom, there is a line of Latin text: "Numquam magni nesciunt est molestiae ipsa ipsa asperiore debitibus ipsa inventore."

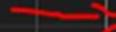
## 7. Buenas prácticas

### Generar URLs amigables

Para ello, nos vamos a database -> migrations

Y abrimos la migración que dice create\_cursos\_table

Y añadimos un campo llamado slug

```
14     public function up()
15     {
16         Schema::create('cursos', function (Blueprint $table) {
17             $table->id();
18             $table->string('name');
19             $table->string('slug'); 
20             $table->text('descripcion');
21             $table->text('categoria');
22             $table->timestamps();
23         });
24     }
25 }
```

## 7. Buenas prácticas

### Generar URLs amigables

Abrimos ahora el factories:  
CursoFactory.php

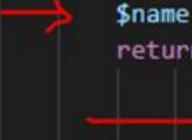
Y escribimos en primer lugar: →

Y después, en definition():

database > factories > CursoFactory.php > CursoFactory > definition

```
1  <?php
2
3  namespace Database\Factories;
4
5  use Illuminate\Database\Eloquent\Factories\Factory;
6  use App\Models\Curso;
7  use Illuminate\Support\Str;
8
```

```
19  public function definition()
20  {
21      $name = $this->faker->sentence();
22      return [
23          'name' => $name, //name se llenará con una frase
24          'slug' => Str::slug($name, '-'),
25          'descripcion' => $this->faker->paragraph(), //description se llenará con un párrafo
26          'categoria' => $this->faker->randomElement(['Desarrollo web', 'Diseño web']) //lo
27      ];
28  }
29 }
```



## 7. Buenas prácticas

### Generar URLs amigables

Como estamos trabajando y modificando migraciones:

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

Si nos vamos a phpmyadmin y en la tabla cursos:

	<input type="checkbox"/>	<input type="checkbox"/> Editar	<input type="checkbox"/> Copiar	<input type="checkbox"/> Borrar	id	name	slug
	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	1	Ipsa facilis et nihil est necessitatibus rerum nes...	ipsa-facilis-et-nihil-est- necessitatibus-rerum-nes...

Es lo mismo que el nombre pero separado por guiones.

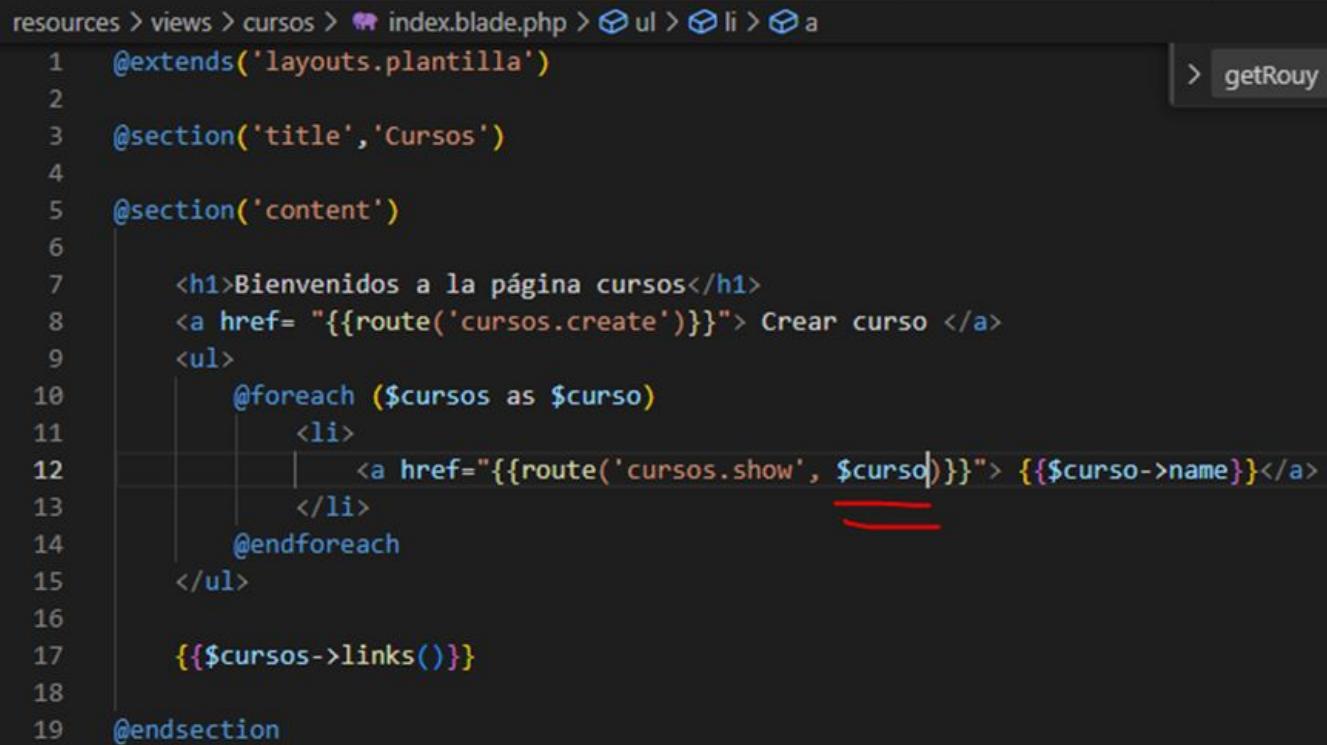
Vamos a usar este campo `slug` para crear nuestras urls con nombre.

## 7. Buenas prácticas

### Generar URLs amigables

En primer lugar, vamos a `index.blade.php` y cambiamos:

```
resources > views > cursos > index.blade.php > ul > li > a
1   @extends('layouts.plantilla')
2
3   @section('title','Cursos')
4
5   @section('content')
6
7       <h1>Bienvenidos a la página cursos</h1>
8       <a href="{{route('cursos.create')}}"> Crear curso </a>
9       <ul>
10          @foreach ($cursos as $curso)
11              <li>
12                  <a href="{{route('cursos.show', $curso)}}> {{$curso->name}}</a>
13              </li>
14          @endforeach
15      </ul>
16
17      {{$cursos->links()}}
18
19  @endsection
```



## 7. Buenas prácticas

### Generar URLs amigables

El método que nos permite recuperar el `id` a partir de `$cursos` es `getRouteKeyName()`

Lo ponemos en `Curso.php` de esta manera:

```
app > Models > Curso.php > ...
1  <?php
2
3  namespace App\Models;
4
5  use Illuminate\Database\Eloquent\Factories\HasFactory;
6  use Illuminate\Database\Eloquent\Model;
7
8  class Curso extends Model
9  {
10     use HasFactory;
11     protected $guarded = [ 'status' ];
12
13     public function getRouteKeyName()
14     {
15         //return $this->getKeyName();
16         return 'slug';
17     }
18
19 }
```

## 7. Buenas prácticas

### Navegabilidad a nuestro sitio web: menú de navegaciones

En primer lugar, tenemos que darle un nombre a la ruta principal /

Nos vamos a web.php:

```
17  
18 Route::get('/', HomeController::class)->name('home');  
19
```

La plantilla que se encarga de administrar esta ruta es views>layaouts> plantilla.blade.php

## 7. Buenas prácticas

### Navegabilidad a nuestro sitio web: menú de navegaciones

Pues nos vamos a ella y escribimos un header en el body .

Como hemos metido la ruta nosotros, creamos esa ruta.

```
resources > views > layouts > plantilla.blade.php > html > body > header > h1
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta http-equiv="X-UA-Compatible" content="IE=edge">
6       <meta name="viewport" content="width=device-width, initial-scale=1.0">
7       <title>@yield('title')</title>
8       <!-- favicon -->
9       <!-- estilos -->
10      </head>
11      <body>
12          <!-- header -->
13          <header>
14              <h1>La cabecera correspondiente</h1>
15              <nav>
16                  <ul>
17                      <li><a href="{{ route('home') }}>Home</a></li>
18                      <li><a href="{{ route('cursos.index') }}>Cursos</a></li>
19                      <li><a href="{{ route('nosotros') }}>Nosotros</a></li>
20                  </ul>
21              </nav>
22          </header>
23          <!-- nav -->
24          @yield('content')
```

## 7. Buenas prácticas

### Navegabilidad a nuestro sitio web: menú de navegaciones

Volvemos de nuevo a web.php y escribimos:

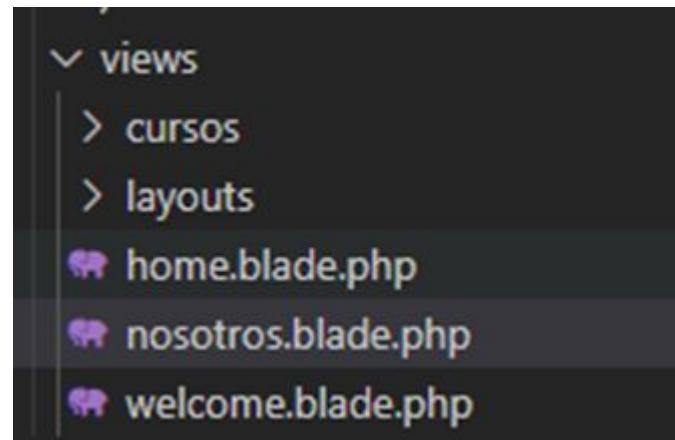
```
34     Route::view('nosotros', 'nosotros')->name('nosotros'); |  
35
```

Esta forma de definir la ruta solo la vamos a usar cuando queremos **mostrar contenido estático**, es decir, que no nos vamos a conectar con nuestra base de datos.

Primer parámetro: url, segundo parámetro: nombre.

Como estamos haciendo uso de una vista que no la hemos creado aún, la creamos:

Nos vamos a resources\views y creamos un:  
nosotros.blade.php



## 7. Buenas prácticas

### Navegabilidad a nuestro sitio web: menú de navegaciones

Y escribimos:



Si lo comprobamos:

```
< > C : localhost/unidad7/public/cursos
```

### La cabecera correspondiente

- [Home](#)
- [Cursos](#)
- [Nosotros](#)

### Bienvenidos a la página cursos

[Crear curso](#)

```
resources > views > 🏠 nosotros.blade.php > ...
1   @extends('layouts.plantilla')
2
3   @section('title', 'Nosotros')
4
5   @section('content')
6       <h1>Acerca de nosotros</h1>
7
8   @endsection
```

## 7. Buenas prácticas

### Navegabilidad a nuestro sitio web: menú de navegaciones

Vamos a activar los links, es decir, que cuando ponga el ratón sobre un link, interactúe.

Nos vamos a plantilla.blade.php.y escribimos:

```
21 header -->
22 <div>
23 <h1>La cabecera correspondiente</h1>
24 <nav>
25   <ul>
26     <li><a href="{{ route('home') }}" class="{{ request()->routeIs('home') ? 'active' : '' }}>Home</a>
27
28     </li>
29     <li><a href="{{ route('cursos.index') }}" class="{{ request()->routeIs('cursos.*') ? 'active' : '' }}>Cursos</a>
30     </li>
31     <li><a href="{{ route('nosotros') }}" class="{{ request()->routeIs('nosotros') ? 'active' : '' }}>Nosotros</a>
32     </li>
33   </ul>
34 </nav>
35 </div>
```

Ponemos ? porque es como si fuera un if, : es como si fuera else  
El \* es para que cuando ingresemos en cursos, no se nos vaya lo que acabamos de crear



## 7. Buenas prácticas

### Navegabilidad a nuestro sitio web: menú de navegaciones

Luego, definimos el estilo para ese active en el header:

Observamos que, llegados a este punto, nuestra plantilla.blade.php se está ensuciando un poco bastante.

**¡Lo vamos a solucionar!**

```
resources > views > layouts > 🌐 plantilla.blade.php > ⚙️ html > ⚙️ head
1   <!DOCTYPE html>
2   <html lang="en">
3   <head>
4       <meta charset="UTF-8">
5       <meta http-equiv="X-UA-Compatible" content="IE=edge">
6       <meta name="viewport" content="width=device-width">
7       <title>@yield('title')</title>
8       <!-- favicon -->
9       <!-- estilos -->
10
11  <style>
12      .active {
13          color: red;
14          font-weight: bold;
15      }
16  </style>
17
18
19  </head>
20  <body>
```

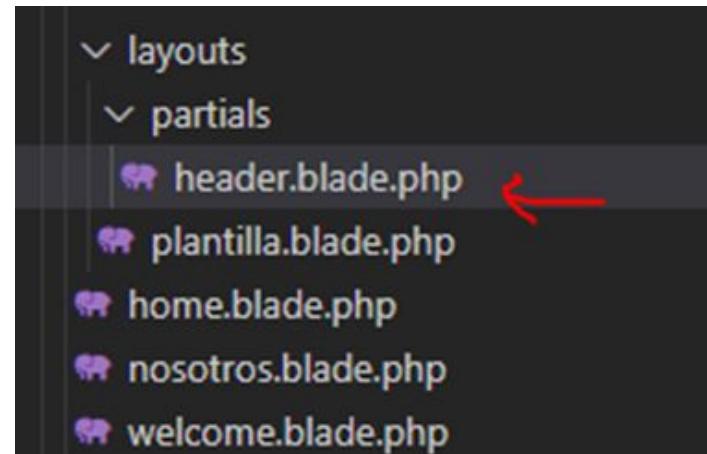
## 7. Buenas prácticas

### Navegabilidad a nuestro sitio web: menú de navegaciones

Lo que hacemos es crear una carpeta dentro de layouts, llamada partials

Y dentro creamos un archivo header.blade.php

Y el contenido header que acabamos de crear en plantilla.blade.php lo cortamos y lo pegamos en header.blade.php



```
resources > views > layouts > partials > header.blade.php > header
1  <header>
2    <h1>La cabecera correspondiente</h1>
3    <nav>
4      <ul>
5        <li><a href="{{ route('home') }}" class="{{ request()->routeIs('home') ? 'active' : '' }}>Home</a>
6
7        </li>
8        <li><a href="{{ route('cursos.index') }}" class="{{ request()->routeIs('cursos.*') ? 'active' : '' }}>Cursos</a>
9        </li>
10       <li><a href="{{ route('nosotros') }}" class="{{ request()->routeIs('nosotros') ? 'active' : '' }}>Nosotros</a>
11       </li>
12     </ul>
13   </nav>
14 </header>
```

## 7. Buenas prácticas

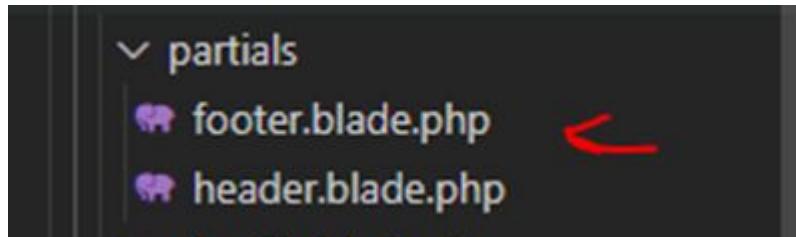
### Navegabilidad a nuestro sitio web: menú de navegaciones

Para incluir ese header dentro de nuestra plantilla, nos volvemos a `plantilla.blade.php` y:

```
19 </head>
20 <body>
21     <!-- header -->
22
23     @include('layouts.partials.header')
24
25     <!-- nav -->
```

Hacemos lo mismo si queremos agregar un **footer a nuestra página.**

Creamos en `partials`: `footer.blade.php`



## 7. Buenas prácticas

### Navegabilidad a nuestro sitio web: menú de navegaciones

Y ponemos en ese archivo:

```
resources > views > layouts > partials > footer.blade.php > footer  
1  <footer>  
2  |   Este es nuestro pie de página  
3  </footer>
```

Y en plantilla.blade.php

```
25      <!-- nav -->  
26      @yield('content')  
27      <!-- footer -->  
28  
29      → @include('layouts.partials.footer')  
30          <!-- script -->  
31      </body>  
32      </html>
```

## 7. Buenas prácticas

### Enviar emails con Laravel

Antes de enviar emails, tenemos que conectarnos a un proveedor de correos electrónicos

Laravel nos permite usar proveedores basados en [SMTP](#) y además proveedores que utilizan otros protocolos basados en API (más rápidos y aconsejados por Laravel)

[SMTP: Protocolo para transferencia simple de correo.](#)

### Configuración para el envío de correos electrónicos

Se encuentra en .env y en la parte de mail:

```
30  
31 MAIL_MAILER=smtp  
32 MAIL_HOST=mailpit  
33 MAIL_PORT=1025  
34 MAIL_USERNAME=null  
35 MAIL_PASSWORD=null  
36 MAIL_ENCRYPTION=null  
37 MAIL_FROM_ADDRESS="hello@example.com"  
38 MAIL_FROM_NAME="${APP_NAME}"  
39
```

Vamos a configurarlo

## 7. Buenas prácticas

### Enviar emails con Laravel

Mailtrap es un servidor SMTP falso para que los equipos de desarrollo prueben, vean y compartan correos electrónicos enviados desde los entornos de desarrollo web, evitando usar nuestro mail personal.

Entonces nos vamos a mailtrap.io y nos logueamos con nuestro correo

Ahí nos aparecerá un correo de demostración en el que salen las credenciales necesarias para conectarnos.

[Hide Credentials ^](#)

#### SMTP

Host: sandbox.smtp.mailtrap.io  
Port: 25 or 465 or 587 or 2525  
Username: bc4c8f0dd9153f  
Password: c41fd1f1d30ef2  
Auth: PLAIN, LOGIN and CRAM-MD5  
TLS: Optional (STARTTLS on all ports)

#### POP3

Host: pop3.mailtrap.io  
Port: 1100 or 9950  
Username: bc4c8f0dd9153f  
Password: c41fd1f1d30ef2  
Auth: USER/PASS, PLAIN, LOGIN, APOP and CRAM-MD5  
TLS: Optional (STARTTLS on all ports)

Y copio la información del host, del port, username y password

```
30
31 MAIL_MAILER=smtp
32 MAIL_HOST=sandbox.smtp.mailtrap.io
33 MAIL_PORT=2525
34 MAIL_USERNAME=bc4c8f0dd9153f
35 MAIL_PASSWORD=c41fd1f1d30ef2
36 MAIL_ENCRYPTION=tls
37 MAIL_FROM_ADDRESS="pilarmatesprofe@gmail.com"
38 MAIL_FROM_NAME="Pilar profe"
39 |
```

## 7. Buenas prácticas

### Enviar emails con Laravel

Ahora, creamos un controlador: `php artisan make:mail ContactanosMailable`

En `App>Mail` se ha creado un `ContactanosMailable.php`

Aquí podemos personalizar lo que va a ir en nuestro correo electrónico, por ejemplo agregando una propiedad que sea el asunto con el cual se enviarán los emails

```
31     public function envelope()
32     {
33         return new Envelope(
34             subject: 'Información de contacto',
35         );
36     }
37 }
```

El método `content()` se va a encargar de traer una vista:

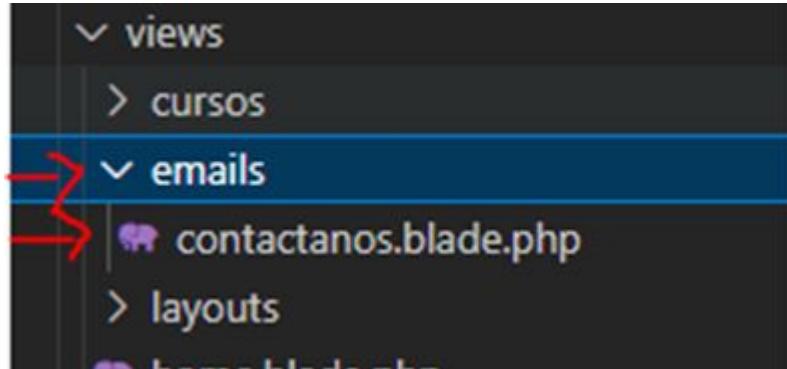
```
45     return new Content(
46         view: 'emails.contactanos',
47     );
48 }
```

## 7. Buenas prácticas

### Enviar emails con Laravel

Creamos la **vista**: en  
resources\views\emails\contactanos.blade.php

Creamos la **ruta**: nos vamos a web.php



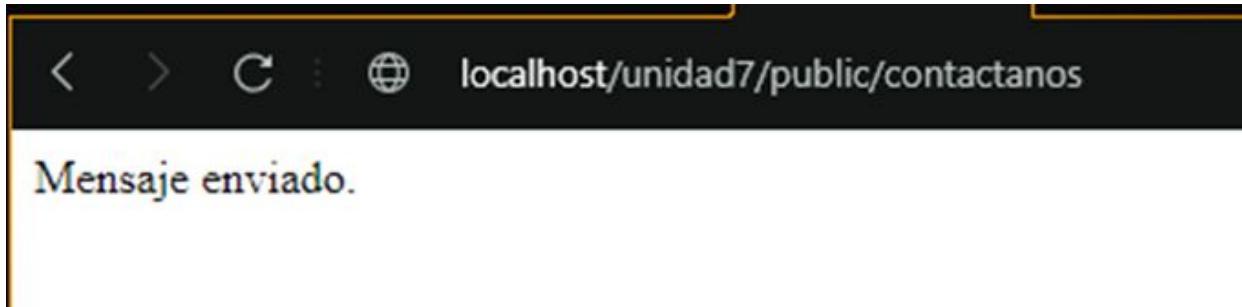
```
15
16 Route::get('contactanos', function () {
17     $correo = new ContactanosMailable;
18     Mail::to('pilarmatesprofe@gmail.com')->send($correo);
19     return "Mensaje enviado.";
20 })->name('contactanos.index');
21
```

```
routes > web.php > ...
1  <?php
2
3  use Illuminate\Support\Facades\Route;
4  use App\Http\Controllers\HomeController;
5  use App\Http\Controllers\CursoController;
6  use App\Mail>ContactanosMailable;
7  use Illuminate\Support\Facades\Mail
8
```

## 7. Buenas prácticas

### Enviar emails con Laravel

Si lo probamos:



Y al estar utilizando Mailtrap ese correo ha sido enviado allí

A screenshot of the Mailtrap inbox interface. At the top, there's a navigation bar with "Inboxes" and "My Inbox" selected. Below that is a search bar and some filter icons. A list of emails is shown, with the first one being "Información de contacto" from "Pilar profe <pilarmatesprofe@gmail.com>". The email details show it was sent "a minute ago" to "pilarmatesprofe@gmail.com". To the right of the list, under the heading "Información de contacto", the email headers are displayed:

From: Pilar profe <pilarmatesprofe@gmail.com>  
To: <pilarmatesprofe@gmail.com>

Below the headers is a link "Show Headers". At the bottom, there are tabs for "HTML", "HTML Source", "Text", and "Raw", with "HTML" currently selected.

## 7. Buenas prácticas

### Formulario contacto

Ahora vamos a agregar a nuestro menú de navegación un link que diga Contáctanos y cuando le demos click, nos redirige a la vista que nos muestre un formulario y ahí el usuario pueda llenar sus datos y nos envíe un correo electrónico

Primero: agregamos un enlace al menú de navegación  
layouts/partials/  
header.blade.php

```
resources > views > layouts > partials > header.blade.php > header
1 <header>
2   <h1>La cabecera correspondiente</h1>
3   <nav>
4     <ul>
5       <li><a href="{{ route('home') }}" class="{{ request()->routeIs('home') ? 'active' : '' }}>
6         Home</a>
7       </li>
8       <li><a href="{{ route('cursos.index') }}" class="{{ request()->routeIs('cursos.*') ? 'active' : '' }}>
9         Cursos</a>
10      </li>
11      <li><a href="{{ route('nosotros') }}" class="{{ request()->routeIs('nosotros') ? 'active' : '' }}>
12        Nosotros</a>
13      </li>
14      <li>
15        <a href="{{ route('contactanos.index') }}"
16          class="{{ request()->routeIs('contactanos.index') ? 'active' : '' }}>
17            Contáctanos</a>
18        </li>
19      </ul>
```

## 7. Buenas prácticas

### Formulario contacto

Creamos un controlador para administrar la ruta:

```
php artisan make:controller  
ContactanosController
```

Le agregamos los métodos **index()** para mostrar un formulario y **store()** para procesar el formulario y enviar el correo electrónico



```
app > Http > Controllers > ContactanosController.php > ...  
1  <?php  
2  
3  namespace App\Http\Controllers;  
4  
5  [use Illuminate\Http\Request;  
6  use App\Mail>ContactanosMailable;  
7  use Illuminate\Support\Facades\Mail;  
8  
9  
10 class ContactanosController extends Controller  
11 {  
12     [public function index()  
13     | {  
14     |     return view('contactanos.index');  
15     | }  
16  
17     [public function store(Request $request) {  
18     | $correo = new ContactanosMailable($request->all());  
19     | Mail::to('pilar matesprofe@gmail.com')->send($correo);  
20     | return "Mensaje enviado.";  
21     }  
22 }
```

## 7. Buenas prácticas

### Formulario contacto

Configuramos la ruta en web.php

Como estamos haciendo uso de index, tenemos que crear una index.blade.php para la vista contactanos.

```
routes > 🌐 web.php
1  <?php
2
3  use Illuminate\Support\Facades\Route;
4  use App\Http\Controllers\HomeController;
5  use App\Http\Controllers\CursoController;
6  use App\Mail\ContactanosMailable;
7  use Illuminate\Support\Facades\Mail;
8  use App\Http\Controllers\ContactanosController; ←
9  use Illuminate\Http\Request;
10
11
12 Route::get('/', HomeController::class)->name('home');
13
14 Route::resource('cursos', CursoController::class);
15
16 Route::view('nosotros', 'nosotros')->name('nosotros');
17
18 → Route::get('contactanos', [ContactanosController::class, 'index'])->name('contactanos.index');
19
```

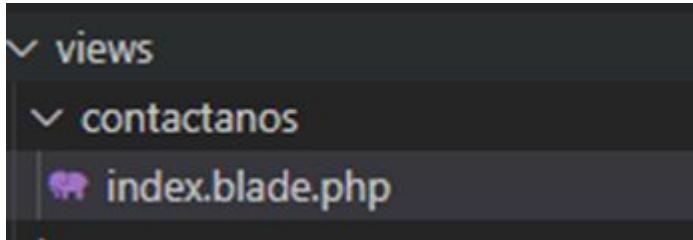
Entonces:

# 7. Buenas prácticas

## Formulario contacto

Creamos en resources\views una carpeta que se llame contactanos

Y dentro: index.blade.php



Y ahí, creamos el formulario:



```
resources > views > contactanos > 🌐 index.blade.php > ...
1   @extends('layouts.plantilla')
2   @section('title', 'Contáctanos')
3   @section('content')
4     <h1>Déjanos un mensaje</h1>
5     <form action="{{ route('contactanos.store') }}" method="POST">
6       @csrf
7       <label>
8         Nombre:
9         <br>
10        <input type="text" name="name">
11      </label>
12      <br>
13      <label>
14        Correo:
15        <br>
16        <input type="mail" name="correo">
17      </label>
18      <br>
19      <label>
20        Mensaje:
21        <br>
22        <textarea name="mensaje" rows="4" ></textarea>
23      </label>
24      <br>
25      <button type="submit">Enviar mensaje</button>
26    </form>
27  @endsection
```

## 7. Buenas prácticas

### Formulario contacto

Y como estamos haciendo referencia a la ruta, la creamos en web.php:

```
24  
25 Route::post('contactanos', [ContactanosController::class, 'store'])->name('contactanos.store');  
26
```

Si lo probamos:

### La cabecera correspondiente

- [Home](#)
- [Cursos](#)
- [Nosotros](#)
- [Contáctanos](#)

### Déjanos un mensaje

Nombre:

Correo:

Mensaje:

Este es nuestro pie de página

## 7. Buenas prácticas

### Formulario contacto

Ahora queremos rescatar el contenido del formulario para mostrarla en el correo que se envía.

Si agregamos una propiedad en ContactanosMailable.php



Cualquier propiedad que incluyamos en esta clase va a poder ser accedida desde el correo electrónico

```
app > Mail > ContactanosMailable.php > ContactanosMailable
6   use Illuminate\Contracts\Queue\ShouldQueue;
7   use Illuminate\Mail\Mailable;
8   use Illuminate\Mail\Mailables\Content;
9   use Illuminate\Mail\Mailables\Envelope;
10  use Illuminate\Queue\SerializesModels;
11
12 class ContactanosMailable extends Mailable
13 {
14     use Queueable, SerializesModels;
15
16
17     public $contacto = "Esta es la información de contacto";
18
19     /**

```

Entonces si vamos a la vista contactanos.blade.php, podemos imprimirla:

```
resources > views > emails > contactanos.blade.php > ...
1   <p>{{ $contacto }}</p>
2   |
```

## 7. Buenas prácticas

### Formulario contacto

La forma que tenemos de pasar información a nuestro correo es la siguiente:

Nos centramos en el controlador ContactanosController.

Lo que hace el método store es recuperar la información que se está mandando desde el formulario a través del objeto Request \$request, cuando instanciamos la clase ContactanosMailable, le pasamos esa información al constructor \$request->all()

```
app > Http > Controllers > ContactanosController.php > ...
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use Illuminate\Http\Request;
6  use App\Mail>ContactanosMailable;
7  use Illuminate\Support\Facades\Mail;
8
9
10 class ContactanosController extends Controller
11 {
12
13     public function index() {
14         return view('contactanos.index');
15     }
16
17     public function store(Request $request) {
18         $correo = new ContactanosMailable($request->all());
19         Mail::to('pilar matesprofe@gmail.com')->send($correo);
20         return "Mensaje enviado.";
21     }
22 }
```

## 7. Buenas prácticas

### Formulario contacto

Recibimos esa información en el constructor de ContactanosMailable.php a través de \$contacto:

```
12  class ContactanosMailable extends Mailable
13  {
14      use Queueable, SerializesModels;
15
16
17      public $contacto;
18
19
20      /**
21      * Create a new message instance.
22      *
23      * @return void
24      */
25
26      public function __construct($contacto)
27      {
28          → $this->contacto = $contacto;
29      }

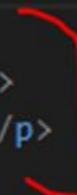
```

## 7. Buenas prácticas

### Formulario contacto

En la vista contactanos.blade.php ponemos entonces:

```
resources > views > emails > 📧 contactanos.blade.php > ...
1
2 |
3 <p><strong>Nombre: </strong> {{ $contacto['name'] }}</p>
4 <p><strong>Correo: </strong> {{ $contacto['correo'] }}</p>
5 <p><strong>Mensaje: </strong> {{ $contacto['mensaje'] }}</p>
6
```



## 7. Buenas prácticas

### Formulario contacto

Y ahora, **validamos la información** que se está enviando desde el formulario:

Nos centramos en el método `store` del controlador:  
`ContactanosController.php`



```
pp > Http > Controllers > ContactanosController.php > ...
6   use App\Mail\ContactanosMailable;
7   use Illuminate\Support\Facades\Mail;
8
9
10  class ContactanosController extends Controller
11  {
12
13  public function index() {
14      |     return view('contactanos.index');
15  }
16
17  public function store(Request $request) {
18
19      |     $request->validate([
20          |         'name' => 'required',
21          |         'correo' => 'required|email',
22      |         'mensaje' => 'required'
23      |     ]);
24
25      |     $correo = new ContactanosMailable($request->all());
26      |     Mail::to('pilarmatesprofe@gmail.com')->send($correo);
27      |     return "Mensaje enviado.";
28  }
29
30  |
```

## 7. Buenas prácticas

### Formulario contacto

Y ahora nos vamos a la vista index.blade.php de contactanos y aquí imprimo los posibles errores, con la directiva @error, @enderror de Blade, debajo de cada etiqueta.

resources > views > contactanos > index.blade.php > form

```
1  @extends('layouts.plantilla')
2  @section('title', 'Contáctanos')
3  @section('content')
4  <h1>Déjanos un mensaje</h1>
5  <form action="{{ route('contactanos.store') }}" method="POST">
6      @csrf
7      <label>
8          Nombre:
9          <br>
10         <input type="text" name="name">
11     </label>
12     <br>
13
14     @error('name')
15     <p><strong>{{ $message }}</strong></p>
16     @enderror
17
```

## 7. Buenas prácticas

### Formulario contacto

```
<label>
    Correo:
    <br>
    <input type="mail" name="correo">
</label>
<br>

@error('correo')
<p><strong>{{ $message }}</strong></p>
@enderror
```

```
.9
0   <label>
1     Mensaje:
2     <br>
3     <textarea name="mensaje" rows="4" ></textarea>
4   </label>
5   <br>
6
7   @error('mensaje')
8     <p><strong>{{ $message }}</strong></p>
9   @enderror
0
```

## 7. Buenas prácticas

### Formulario contacto

Por último, para volver a la vista después del mensaje enviado.

Nos vamos a ContactoController.php y cambiamos el return:



```
0 class ContactanosController extends Controller
1 {
2
3     public function index() {
4         return view('contactanos.index');
5     }
6
7     public function store(Request $request) {
8
9         $request->validate([
0             'name' => 'required',
1             'correo' => 'required|email',
2             'mensaje' => 'required'
3         ]);
4
5         $correo = new ContactanosMailable($request->all());
6         Mail::to('pilarmatesprofe@gmail.com')->send($correo);
7
8         return redirect()->route('contactanos.index')->with('info', 'Mensaje enviado');
9     }
0
1 }
```

With (info será la sesión y lo otro el mensaje)  
crea el mensaje de sesión

## 7. Buenas prácticas

### Formulario contacto

Entonces, en la vista:  
contactanos  
index.blade.php mostramos  
una alerta cuando haya un mensaje  
de sesión, esto se hace:

```
43
44 @if (session('info'))
45 <script>
46 |   |   alert('{{ session('info') }}')
47 </script>
48 @endif
49
```

Si lo probamos:

localhost/unidad7/public/contactanos

localhost dice

Mensaje enviado

Aceptar