



## Laravel. Nivel I

julio, 2019



## Objetivos del nivel

- Aprender a instalar el framework Laravel
- Crear modelos, vista y controladores
- Aprender a manipular una base de datos MySQL usando la librería ORM Eloquent

## Prerrequisitos del nivel

- PHP Nivel III

## Acerca de este manual

Este manual pertenece al Centro de Asesoramiento y Desarrollo Informático C.A. (CADIF1). Para obtener más información sobre este u otros cursos visite nuestro sitio Web [www.cadif1.com](http://www.cadif1.com), escribanos a la dirección de correo [cadi@cadif1.com](mailto:cadi@cadif1.com) o visítenos en nuestra sede ubicada en la Av. Pedro León Torres con calle 59, Centro Comercial Sotavento, piso 2 oficina 27, Barquisimeto estado Lara, Venezuela. Tlf. 0251-7179247, 0251-4410268.

Las marcas mencionadas en este manual son propiedad de sus respectivos dueños.  
Copyright 2019. Todos los derechos reservados.



## Contenido del nivel

### Capítulo 1. Conociendo Laravel

- 1.1.- Qué es Laravel.
- 1.2.- Instalación.
- 1.3.- Estructura Del Proyecto.

### Capítulo 2. Primeras Configuraciones

- 2.1.- Directorio Config.
- 2.2.- Idioma de la Aplicación.
- 2.3.- Modo Mantenimiento.

### Capítulo 3. El Patrón Mvc

- 3.1.- Concepto.
- 3.2.- El Patrón Mvc en Aplicaciones Web.
- 3.3.- El Patrón Mvc y Laravel.

### Capítulo 4. Enrutamiento

- 4.1.- Las Rutas.
- 4.2.- El Archivo Web.php.
- 4.3.- Estableciendo Rutas.

### Capítulo 5. Controladores

- 5.1.- Crear el Controlador.
- 5.2.- Enrutar un Método Del Controlador.
- 5.3.- Enrutar un Controlador Completo.

### Capítulo 6. Vistas. Parte 1

- 6.1.- Creado Vistas.
- 6.2.- Vista- Controlador.
- 6.3.- Pasando Datos a Las Vistas.

## Capítulo 7. Vistas. Parte 2

- 7.1.- Agregando Hojas de Estilo.
- 7.2.- Heredando Vistas.

## Capítulo 8. Configuración de la bd

- 8.1.- Archivo de Configuración.
- 8.2.- Migraciones.

## Capítulo 9. Query Builder

- 9.1.- Definición.
- 9.2.- Recuperando Tablas.
- 9.3.- Recuperando Datos Específicos.

## Capítulo 10. Eloquent

- 10.1.- Definición.
- 10.2.- Modelos.
- 10.3.- Tinker.

## Capítulo 11. Leer Registros

- 11.1.- Index.
- 11.2.- Ruta Show.
- 11.3.- Método Show.

## Capítulo 12. Agregar Registros

- 12.1.- Introducción.
- 12.2.- Index.
- 12.3.- Método Create.
- 12.4.- Formulario.

## Capítulo 13. Modificar Registros

- 13.1.- Introducción.
- 13.2.- Métodos Edit y Update.

## Capítulo 14. Eliminar Registros

- 14.1.- Introducción.
- 14.2.- Formulario de Eliminado.
- 14.3.- Método Destroy.

## Capítulo 15. Validaciones

- 15.1.- Método Validate().
- 15.2.- Form Requests.

## Capítulo 1. CONOCIENDO LARAVEL

### 1.1.- Qué es Laravel

Laravel es un framework de PHP de código libre, creado en 2011 por Taylor Otwell y distribuido bajo la licencia MIT, pensado en desarrollar aplicaciones de una manera rápida con sintaxis elegante y expresiva. Desde su liberación ha ido ganando terreno entre los distintos frameworks de desarrollo de Php.

Un framework es un marco base o estructura conceptual y tecnológica de soporte, definido normalmente con artefactos o módulos concretos de software, que puede servir de base para la organización y desarrollo de software.



El sitio oficial de Laravel es <https://laravel.com>. Laravel tiene una gran influencia de frameworks como Ruby on Rails. Entre las características más resaltantes están:

- Soporte para MVC
- Sistema de ruteo
- Motor de plantillas Blade
- Peticiones Fluent
- Eloquent ORM
- Basado en Composer

Para trabajar con Laravel (la versión 5.8 que se usó para este curso) hay algunos pre requisitos (además de tener conocimientos en Php):

- Servidor web (cualquiera)
- PHP >= 7.1.3
- Habilitar la extensión BCMath PHP
- Habilitar la extensión Ctype PHP
- Habilitar la extensión JSON PHP
- Habilitar la extensión OpenSSL PHP
- Habilitar la extensión PDO PHP
- Habilitar la extensión Mbstring PHP
- Habilitar la extensión Tokenizer PHP
- Habilitar la extensión XML PHP

## 1.2.- Instalación

El repositorio oficial del framework en gitup es <https://github.com/laravel/laravel>. Laravel se puede instalar de 2 formas:

- 1- descargando el archivo <https://github.com/laravel/laravel/archive/master.zip> y descomprimiéndolo en la carpeta pública de apache.
- 2- usando Composer para descargar todo lo que haga falta.

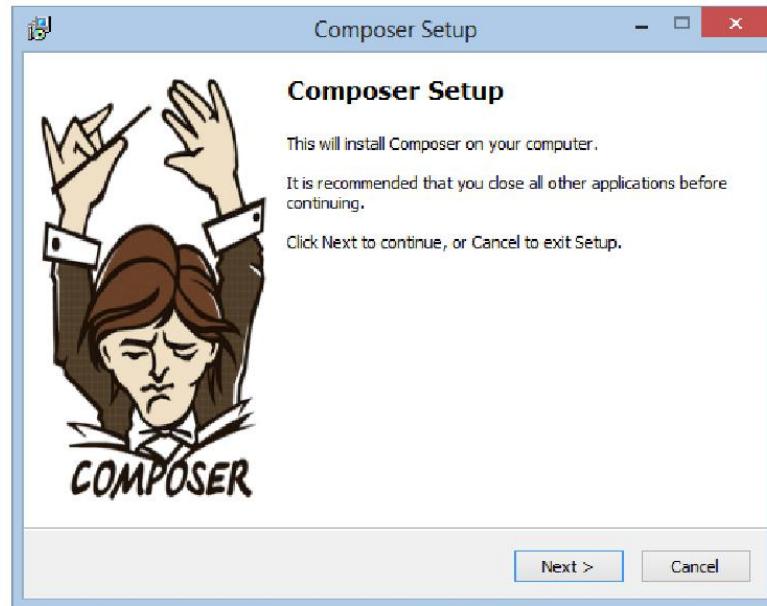
La forma recomendada de hacerlo en la documentación de la página oficial (<https://laravel.com/docs/5.8>) es usando Composer.

Composer es el manejador de dependencias de PHP, que permite manejar las dependencias de los proyectos, ya sea la descarga de un framework o componentes más sencillos como un generador de PDFs.

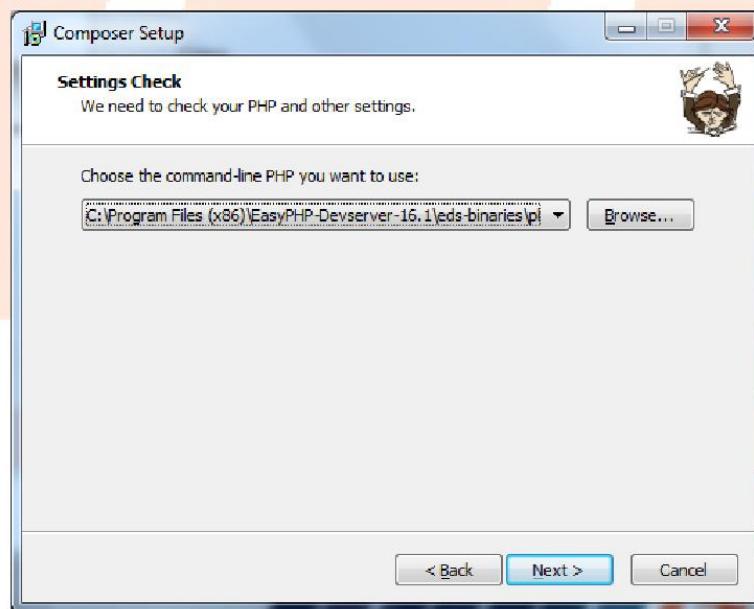
Composer corre en la línea de comandos e instala dependencias para una aplicación. También permite a los usuarios instalar aplicaciones que están disponibles en "Packagist" que es el repositorio principal de paquetes. Se puede visitar el sitio del repositorio en la url <https://packagist.org/>.



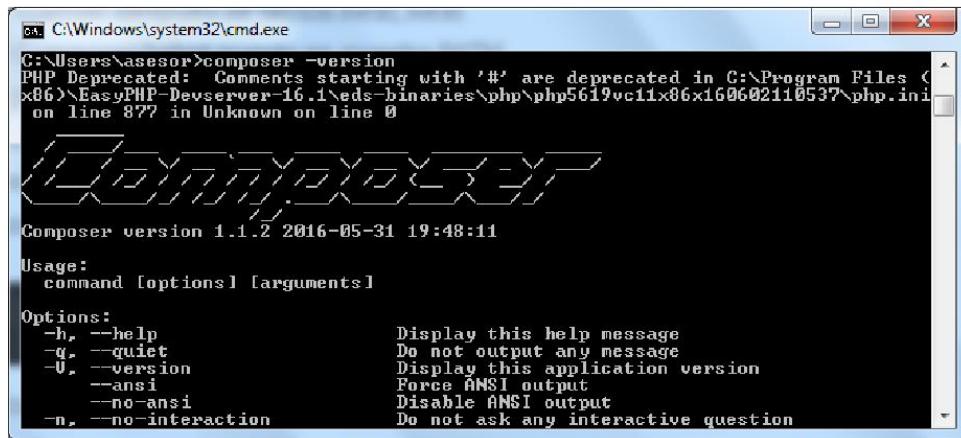
Para instalar Composer debe descargar el instalador desde la página oficial <https://getcomposer.org/>. Para Windows existe un instalador que facilita el trabajo. Antes de instalar Composer debe estar instalado Php (con cualquier paquete como EasyPhp o Wamp).



Durante la instalación de Composer se debe indicar el directorio donde está ubicado el ejecutable de Php:



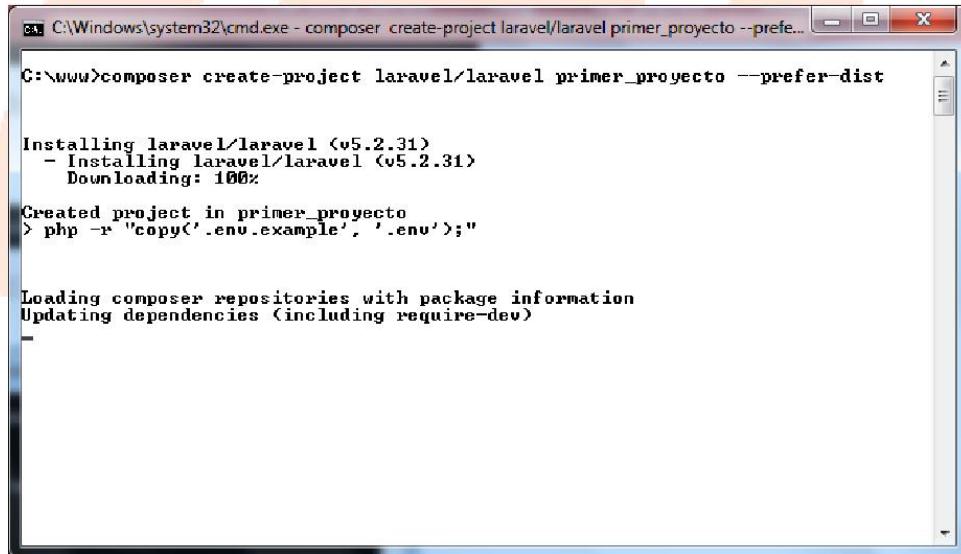
Al finalizar el proceso de instalación se puede comprobar que Composer se haya instalado correctamente abriendo una ventana de comandos y escribiendo el comando: composer -version, lo que mostrará un resultado como el siguiente:



```
C:\Windows\system32\cmd.exe
C:\Users\asesor>composer -version
PHP Deprecated: Comments starting with '#' are deprecated in C:\Program Files (x86)\EasyPHP-Devserver-16.1\eds-binaries\php\php5619vc11x86x160602110537\php.ini
on line 877 in Unknown on line 0

Composer version 1.1.2 2016-05-31 19:48:11
Usage:
  command [options] [arguments]
Options:
  -h, --help           Display this help message
  -q, --quiet          Do not output any message
  -V, --version         Display this application version
  --ansi               Force ANSI output
  --no-ansi             Disable ANSI output
  -n, --no-interaction Do not ask any interactive question
```

Luego de instalado el Composer, se debe ejecutar el comando:



```
C:\Windows\system32\cmd.exe - composer create-project laravel/laravel primer_proyecto --prefer-dist
C:\www>composer create-project laravel/laravel primer_proyecto --prefer-dist

Installing laravel/laravel (v5.2.31)
- Installing laravel/laravel (v5.2.31)
  Downloading: 100%
Created project in primer_proyecto
> php -r "copy('.env.example', '.env');"

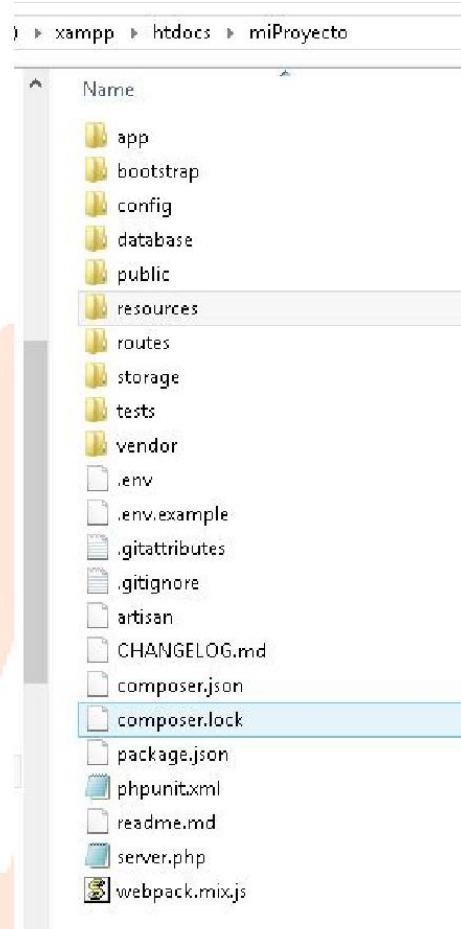
Loading composer repositories with package information
Updating dependencies (including require-dev)
-
```

### 1.3.- Estructura Del Proyecto

Luego de instalado Laravel, se puede probar si todo está bien ejecutando el archivo index.php que está contenido en la carpeta "public". Si todo está bien debería aparecer lo siguiente:



La estructura por defecto de una aplicación Laravel está hecha para proveer un excelente punto de partida tanto para aplicaciones pequeñas como para aplicaciones grandes. El programador tiene la libertad de organizar la aplicación como prefiera, debido a que Laravel no tiene mayores restricciones para la ubicación de las clases. La siguiente imagen muestra las carpetas que por defecto posee un proyecto Laravel:

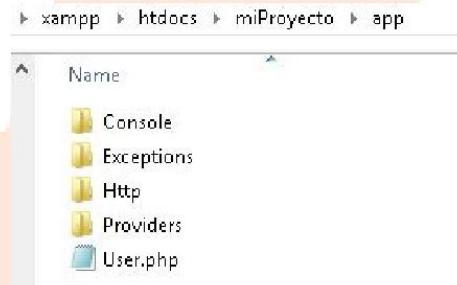


Los directorios que se encuentran en la raíz del proyecto son:

- app: contiene el código principal de la aplicación.
- bootstrap: En esta carpeta se incluye el código que se carga para procesar cada una de las llamadas a nuestro proyecto.
- config: Aquí se encuentran todos los archivos de configuración de la aplicación.
- database: En esta carpeta se incluye todo lo relacionado con la definición de la base de datos de nuestro proyecto (migrations, seeds, factory).

- public: Es el punto de entrada para todas las solicitudes que ingresan a su aplicación y configura la carga automática. Es la única carpeta que debería ser visible en nuestro servidor web. Contiene los recursos (imágenes, js, css, etc).
- resources: Contiene las vistas blade utilizadas por la aplicación.
- routes: Contiene las rutas de la aplicación.
- storage: En esta carpeta Laravel almacena toda la información interna necesaria para la ejecución de la web, como son los archivos de sesión, la caché, la compilación de las vistas, meta información y los logs del sistema. Normalmente no tendremos que tocar nada dentro de esta carpeta, únicamente se suele acceder a ella para consultar los logs.
- tests: Esta carpeta se utiliza para almacenar los ficheros con las pruebas automatizadas. Laravel incluye un sistema que facilita todo el proceso de pruebas con PHPUnit.
- vendor: Contiene todas las dependencias utilizadas por el framework pero instaladas con composer.

El directorio "app" contiene el corazón de la aplicación. Este a su vez está dividido en varios sub directorios importantes que serán revisados con detalle más adelante:



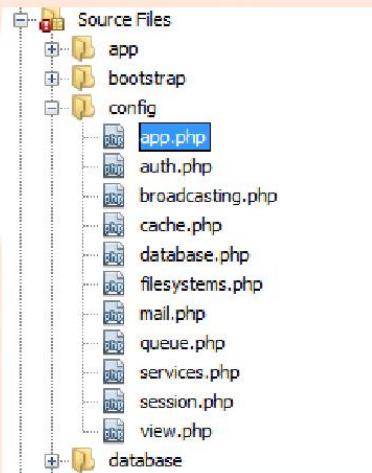
## Capítulo 2. PRIMERAS CONFIGURACIONES

### 2.1.- Directorio Config

Todos los archivos de configuración para el marco de Laravel se almacenan en el directorio config. Cada opción está documentada, así que no dude en consultar los archivos y familiarizarse con las opciones disponibles para usted.

El archivo de configuración principal del Framework está localizado en:

config/app.php



Entre las configuraciones más importantes que se pueden realizar en el archivo app.php se encuentran:

- Nombre de la aplicación
- Habilitar o deshabilitar modo de depuración
- Zona horaria
- Lenguaje de la aplicación
- Proveedores de servicios
- Alias

Cada una de estas variables de entorno están debidamente comentadas para dar mayor certeza de su funcionamiento. En la imagen se observa una vista previa y sin comentarios de parte del archivo de configuración app.php

```
*/  
  
'name' => env('APP_NAME', 'Laravel'),  
'env' => env('APP_ENV', 'production'),  
'debug' => env('APP_DEBUG', false),  
'url' => env('APP_URL', 'http://localhost'),  
'timezone' => 'UTC',  
'locale' => 'en',  
'fallback_locale' => 'en',  
'key' => env('APP_KEY'),  
'cipher' => 'AES-256-CBC',  
'log' => env('APP_LOG', 'single'),  
'log_level' => env('APP_LOG_LEVEL', 'debug'),  
'providers' => [  
    Illuminate\Auth\AuthServiceProvider::class,  
    Illuminate\Broadcasting\BroadcastServiceProvider::class,  
    Illuminate\Bus\BusServiceProvider::class,  
    Illuminate\Cache\CacheServiceProvider::class,  
    Illuminate\Foundation\Providers\ConsoleServiceProvider::class,  
    Illuminate\Cookie\CookieServiceProvider::class,  
    Illuminate\Database\DatabaseServiceProvider::class,  
    Illuminate\Encryption\EncryptionServiceProvider::class,  
    Illuminate\Filesystem\FilesystemServiceProvider::class
```

Para editar algunas de las variables, simplemente se debe modificar el valor presente en ella. En la imagen se muestra el cambio de nombre de la aplicación a "Cadif1App".

```
'name' => env('APP_NAME', 'Cadif1App'),
```

Laravel contiene un archivo en la raíz de la aplicación

llamado .env , el cual es un archivo donde el desarrollador puede definir variables de entorno para posteriormente ser usadas en los diferentes archivos de configuración. El archivo .env no debe comprometerse con el control de origen de la aplicación, ya que cada desarrollador / servidor que utilice su aplicación podría requerir una configuración de entorno diferente. Además, esto supondría un riesgo para la seguridad en caso de que un intruso obtenga acceso a su repositorio de control de origen, ya que cualquier credencial confidencial quedaría expuesta.

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=
APP_DEBUG=true
APP_LOG_LEVEL=debug
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=

BROADCAST_DRIVER=log
CACHE_DRIVER=file
SESSION_DRIVER=file
QUEUE_DRIVER=sync

REDIS_HOST=127.0.0.1
REDIS_PASSWORD=null
REDIS_PORT=6379

MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null

PUSHER_APP_ID=
PUSHER_APP_KEY=
PUSHER_APP_SECRET=
```

## 2.2.- Idioma de la Aplicación

La especificación del idioma de la aplicación está definido en el archivo app.php del directorio config. Por defecto el idioma de la aplicación es el inglés, pero esto puede ser modificado por el desarrollador.

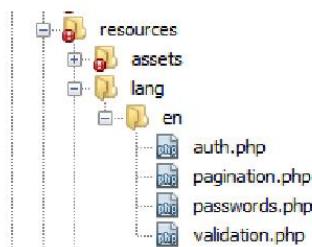
La variable "locale" define el idioma de la aplicación, y por otro lado la variable "fallback\_locale" especifica el idioma secundario.

```
'locale' => 'en',  
  
'fallback_locale' => 'en',
```

Las funciones de localización de Laravel proporcionan una forma conveniente de recuperar cadenas en varios idiomas, lo que le permite admitir fácilmente varios idiomas dentro de su aplicación. Las cadenas de idioma se almacenan en archivos dentro del directorio resources/ lang. Dentro de este directorio debe haber un subdirectorio para cada idioma admitido por la aplicación.

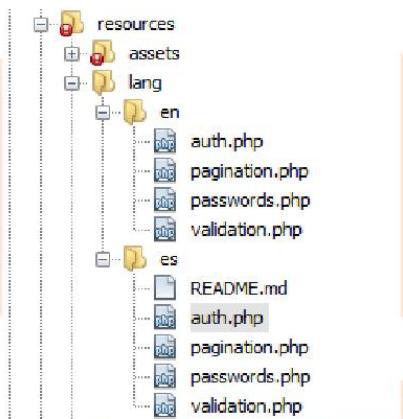
Todos los archivos de idioma devuelven una matriz de cadenas con clave. Por ejemplo:

```
return [  
    "welcome" => "Welcome to our application"  
];
```



Para cambiar el idioma de la aplicación a español se debe descargar la carpeta para el idioma español, copiarla dentro del directorio resource/lang y modificar el valor de la variable "locale" del archivo app.php del directorio config de la siguiente manera:

locale=>es

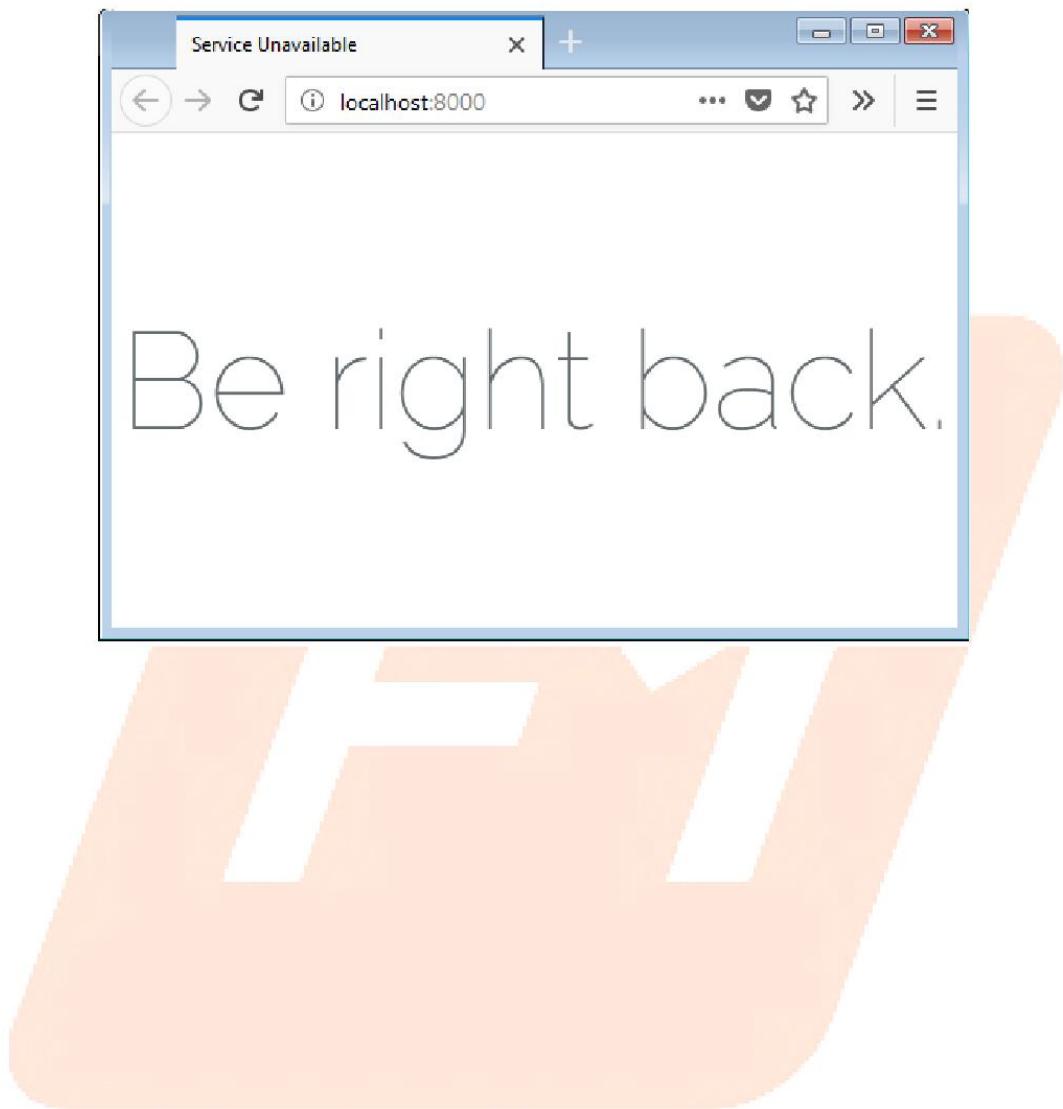


### 2.3.- Modo Mantenimiento

Cuando la aplicación se encuentre en modo de mantenimiento, se mostrará una vista personalizada para todas las solicitudes en su aplicación. Esto facilita la "desactivación" de su aplicación mientras se actualiza o cuando se realiza mantenimiento.

Para habilitar el modo de mantenimiento, ejecute el comando down Artisan:

```
php artisan down
```



## Capítulo 3. EL PATRÓN MVC

### 3.1.- Concepto

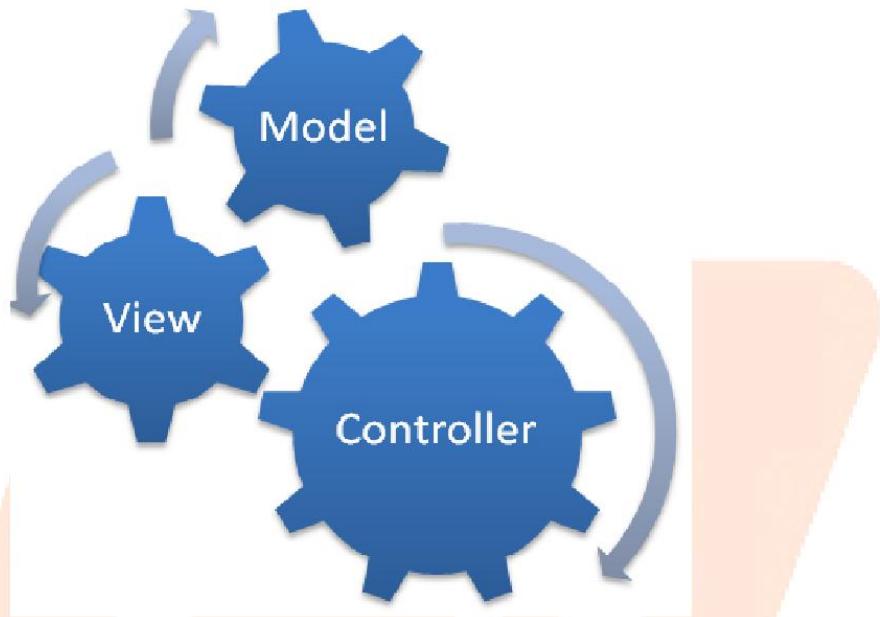
Durante el desarrollo de software es posible encontrarse con situaciones en las que el programador nota que está resolviendo un problema similar a otro(s) que ya ha resuelto, aunque el problema actual tenga ciertas particularidades que no tenían los anteriores. Lo ideal sería resolver el nuevo problema usando la misma estrategia que se usó en el problema anterior.

Los patrones son esquemas generales de solución para problemas que típicamente se pueden encontrar en el desarrollo de software. La idea es analizar un problema y verificar si este puede enmarcarse en alguno de los patrones ya definidos, en caso de ser así, se resuelve aplicando el patrón que encaja con el problema analizado.



Existe un patrón de diseño muy popular llamado Modelo-Vista-Controlador (MVC). Este patrón puede aplicarse en cualquier desarrollo de software, ya sea aplicaciones de consola, de escritorio, web, etc. La idea de este patrón es separar en 3 capaz todo el desarrollo del software, donde la interfaz del usuario queda separada de la lógica del negocio, de forma tal que se haga más fácil la implementación y el mantenimiento del software.

El patrón MVC está basado netamente en la programación orientada a objetos. En la práctica, cada una de las capas serán clases que tendrán responsabilidades diferentes.



En el patrón MVC cada uno de los elementos tiene una función particular:

- El modelo: contiene la(s) clase(s) que manejar la lógica del negocio. Dicho de otra forma, en el modelo es donde se resuelven los problemas específicos, como los cálculos, las conexiones a base de datos, las búsquedas, etc. El modelo debe estar hecho de forma genérica, es decir, no debe estar apegado a ningún tipo de interfaz: ventana, consola, página html, etc.
- la vista: tiene como responsabilidad servir de interfaz con el usuario, es la encargada de capturar los datos del usuario (lectura de entradas) y la encargada de mostrar los resultados al usuario.
- el controlador: es el intermediario entre la vista y el modelo. Su único objetivo es lograr que estas 2 capas que están separadas puedan comunicarse con el paso de mensajes entre los objetos.

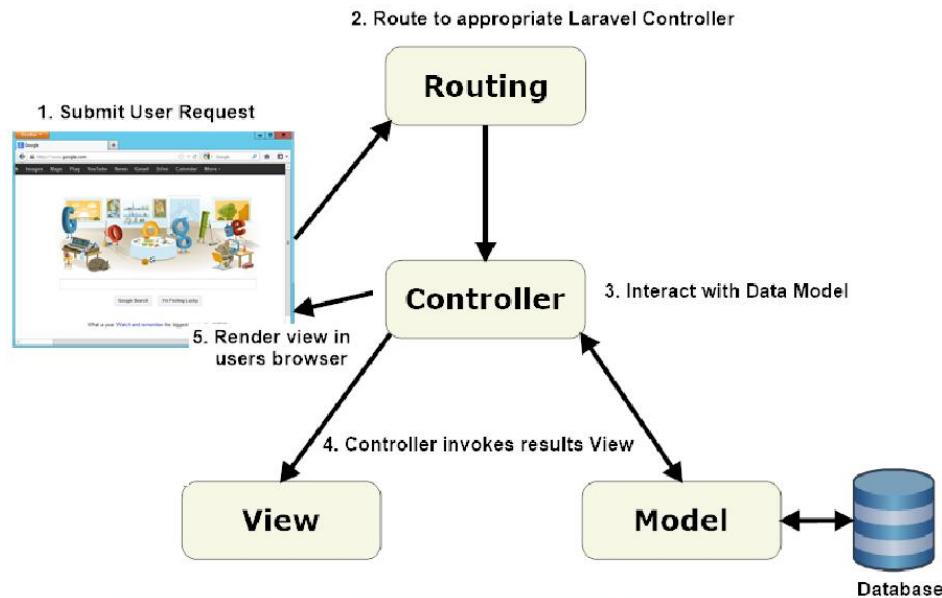
### 3.2.- El Patrón Mvc en Aplicaciones Web

En las aplicaciones web, el patrón MVC se aplica de la siguiente forma:



### 3.3.- El Patrón Mvc y Laravel

El esquema de funcionamiento del patrón MVC en Laravel es el siguiente:



En Laravel específicamente, el funcionamiento del patrón y la dinámica de ejecución de la aplicación es muy diferente a la forma de trabajar aplicaciones php sin framework. En forma general, la dinámica es la siguiente:

- el punto de entrada es el archivo public/index.php. Todas las solicitudes son enviadas por el servidor web directamente a este archivo. El index.php es el punto de partida para cargar todo el framework.
- el archivo index.php carga el autoloader generado por Composer y retorna una instancia de la aplicación Laravel ubicada en el script bootstrap/app.php. La primera acción tomada por la aplicación Laravel es crear una instancia del contenedor de servicios de la aplicación.
- luego, la solicitud entrante es enviada o al "HTTP kernel" a la "console kernel", dependiendo del tipo de solicitud que recibe la aplicación.
- El "HTTP kernel" además define una lista de "HTTP middleware", a través del cual tienen que pasar todas las solicitudes manejadas por la aplicación. Estos middleware manejan la lectura y escritura de las sesiones, determina si la aplicación está en mantenimiento, verifican los CSRF, entre otras cosas.
- el objeto aplicación carga los proveedores de servicios. Los proveedores de servicios están configurados en el archivo config/app.php. Los proveedores de

servicios son los responsables de levantar varios de los componentes del framework: base de datos, colas, validaciones y enrutamiento.

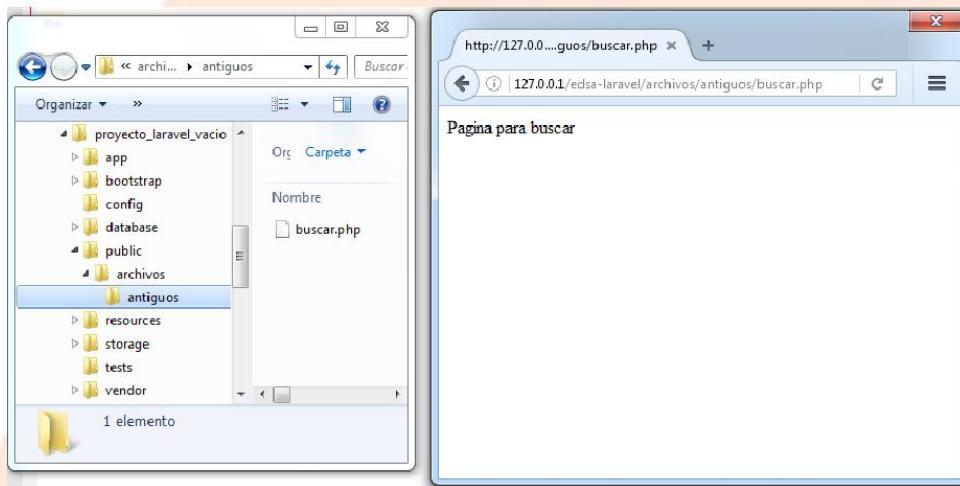
- Una vez ha sido levantada la aplicación y todos los servicios han sido registrados, la solicitud será enviada al enrutador para ser despachada a un controlador o a un middleware específico.



## Capítulo 4. ENRUTAMIENTO

### 4.1.- Las Rutas

En las aplicaciones web tradicionales, la URL con la cual se accede a los recursos tiene que ver con la estructura de carpetas físicas de la carpeta raíz del sitio y de la ubicación de los archivos dentro de esta. Por ejemplo, si se necesita acceder a un archivo que está en la carpeta "archivos/antiguos/buscar.php" que está en la carpeta raíz del sitio, se debe colocar en la URL: direcciónIp/carpeta/archivos/antiguos/buscar.php, como se muestra en la imagen:



Para resolver este problema, Laravel utiliza el patrón "Front Controller". El patrón de diseño "Front Controller" fue diseñado para las aplicaciones web. Este provee un punto de entrada único para manejar todas las solicitudes que se realizan a la aplicación. De esta forma, se evita estar atado a la estructura de las carpetas del sitio, lo que aporta mayor flexibilidad al momento de hacer cambios de nombres, así como también, se evita que los visitantes conozcan en qué lenguaje de programación fue programada la aplicación.

La aplicación del "Front Controller" se logra gracias al uso de archivos .htaccess de apache. En la carpeta "public" se encuentra este archivo y es quien le dice a apache que todas las solicitudes recibidas deben ser manejadas por el archivo index.php:



```
htaccess x
1 <IfModule mod_rewrite.c>
2     <IfModule mod_negotiation.c>
3         Options -MultiViews -Indexes
4     </IfModule>
5
6     RewriteEngine On
7
8     # Handle Authorization Header
9     RewriteCond %{HTTP:Authorization} .
10    RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
11
12    # Redirect Trailing Slashes If Not A Folder...
13    RewriteCond %{REQUEST_FILENAME} !-d
14    RewriteCond %{REQUEST_URI} (.+)/$
15    RewriteRule ^ %1 [L,R=301]
16
17    # Handle Front Controller...
18    RewriteCond %{REQUEST_FILENAME} !-d
19    RewriteCond %{REQUEST_FILENAME} !-f
20    RewriteRule ^ index.php [L]
21 </IfModule>
22
```

#### 4.2.- El Archivo Web.php

Como se explicó en el capítulo anterior, el archivo index.php es quien levanta la aplicación Laravel. Luego del proceso de inicio, las peticiones son enviadas al archivo web.php. Este archivo está ubicado en la carpeta "routes". Es el encargado de hacer el enrutamiento de todas las peticiones recibidas por el index.php, para ser manejadas apropiadamente. El archivo contiene una ruta mínima por defecto. En la imagen muestra el contenido original del archivo:

```
routes.php 3
1 <?php
2
3 /*
4 | -----
5 | Application Routes
6 | -----
7 |
8 | Here is where you can register all of the routes for an application.
9 | It's a breeze. Simply tell Laravel the URIs it should respond to
10| and give it the controller to call when that URI is requested.
11|
12 */
13
14 Route::get('/', function () {
15     return view('welcome');
16 });
17
18
```

En el archivo se muestra el uso de la clase `Route` y el llamado a un método estático con 2 parámetros: un string (que indica la ruta a la cual se responderá) y una función anónima (que responderá la acción). En el ejemplo anterior, se responderá a todos los llamados que se hagan a la raíz del sitio y se retornará una vista llamada "welcome" (el uso detallado de las vistas se explicará más adelante). El formato general de un route es:

```
Route::metodoEnvio("ruta", function ()
{
    código de respuesta al llamado de la ruta
})
```

Los métodos de envío pueden ser: get, post, put, patch, delete y options.

#### 4.3.- Estableciendo Rutas

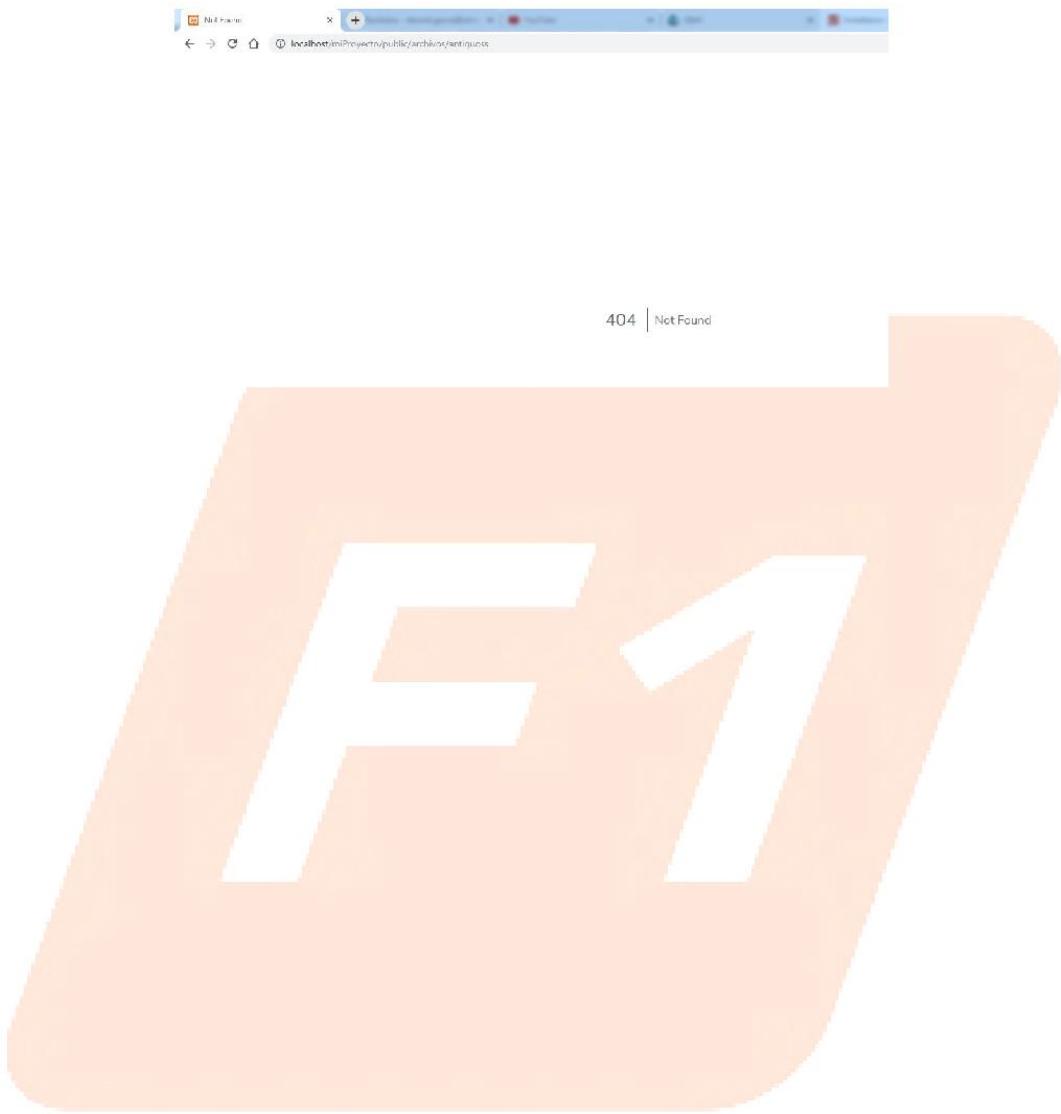
Se pueden crear tantas rutas como haga falta. El equivalente al ejemplo con la URL dependiente a la dirección física de los archivos es el siguiente:

The screenshot shows a code editor with two tabs: '.htaccess' and 'web.php'. The 'web.php' tab contains the following PHP code:

```
1 <?php
2
3 /*
4 | -----
5 | Web Routes
6 | -----
7 |
8 | Here is where you can register web routes for your
9 | routes are loaded by the RouteServiceProvider within
10 | contains the "web" middleware group. Now create som
11 |
12 */
13
14 Route::get('/', function () {
15     return view('welcome');
16 });
17
18 Route::get('/archivos/antiguos', function () {
19     return 'pagina para buscar';
20 });
```

Below the code editor is a browser window. The address bar shows 'localhost/miProyecto/public/archivos/antiguos'. The page content area displays the text 'pagina para buscar'.

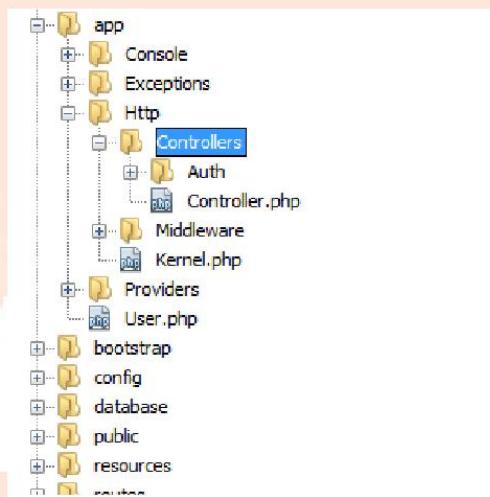
Si se hace referencia a una ruta que no existe, laravel devuelve una vista propia del framework con un código de estado 404 (página no encontrada). En el ejemplo se está haciendo la petición a la ruta “/archivos/antiguoss” que no está definida en el routes/web.php



## Capítulo 5. CONTROLADORES

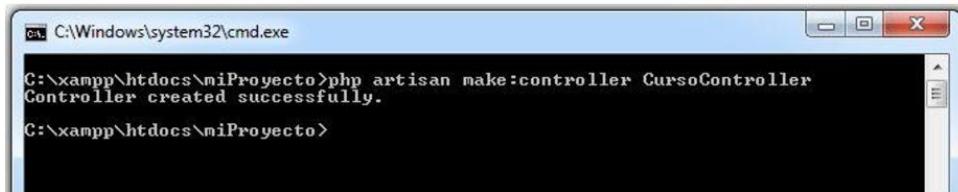
### 5.1.- Crear el Controlador

En lugar de definir toda la lógica de gestión de solicitudes con funciones en el archivo de rutas, se puede organizar este comportamiento utilizando clases de controlador. Los controladores pueden agrupar la lógica de manejo de solicitudes relacionada en una sola clase. Los controladores se almacenan en el directorio de la aplicación / Http / Controllers.



Para crear un controlador se debe usar la interfaz de línea de comandos (CLI por sus siglas en inglés de Command-line interface), el cual es un medio para la interacción con la aplicación donde los usuarios (en este caso los desarrolladores) dan instrucciones en forma de línea de texto simple o línea de comando. Simplemente se debe abrir la consola de Windows y ejecutar la siguiente instrucción:

```
php artisan make:controller nombreControlador
```



Los controladores de manera predeterminada contendrán su definición como se muestra en la imagen:

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

## 5.2.- Enrutar un Método Del Controlador

Un controlador debería responder o procesar las peticiones de los métodos HTTP:

- GET.
- POST.
- PUT.
- DELETE.
- PATCH.

En el archivo de rutas debe asociarse una ruta con un método en la clase controlador, como se muestra en el ejemplo, se asocia la ruta "cursos" usando el método HTTP "GET", con el método "index" de la clase "CursoController":

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

Route::get('cursos', 'CursoController@index');

controlador
CursoController.php
1 <?php
2 namespace App\Http\Controllers;
3 use Illuminate\Http\Request;
4
5 class CursoController extends Controller
6 {
7     public function index()
8     {
9         return "pagina de cursos";
10    }
}
```

### 5.3.- Enrutar un Controlador Completo

Asociando los métodos que servirán para la interacción con los modelos y la base de datos de la siguiente forma:

- GET: index, create, show, edit.
- POST: store.
- PUT: update.
- DELETE: destroy.
- PATCH: update.

Para esto se usa un tipo de ruta llamada resource:

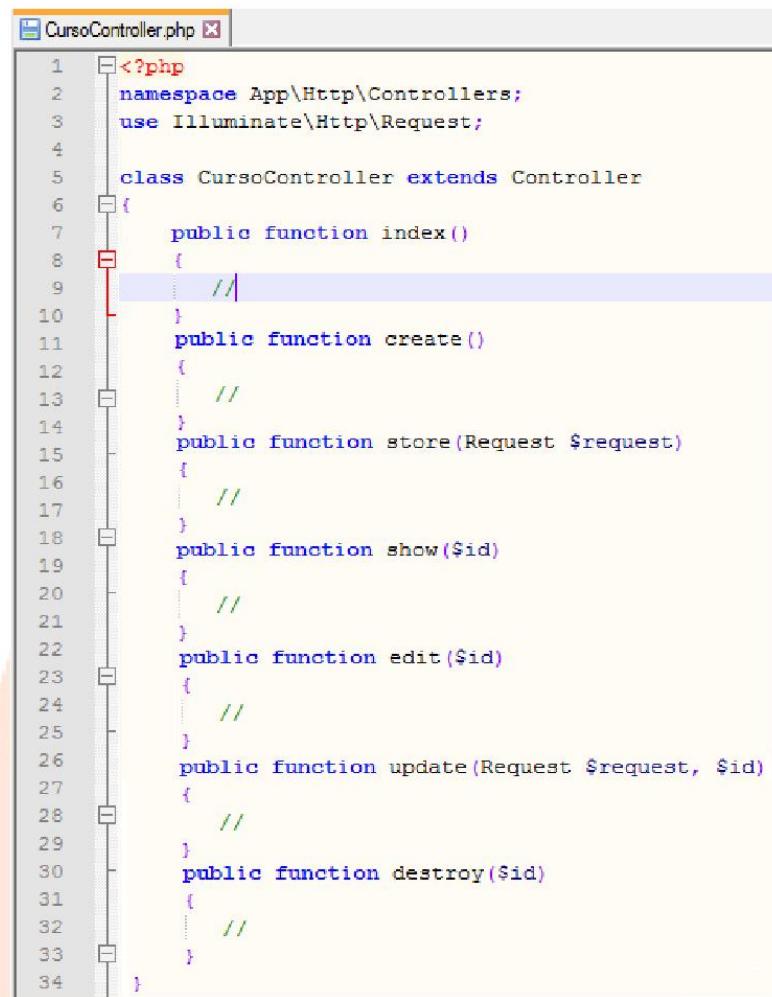
```
Route::resource("cursos", "CursoController");
```

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

Luego, se crean en el controlador los métodos específicos que respondan a cada una de las peticiones, lo cual se logra con el siguiente comando:

```
php artisan make:controller cursoController --resource
```

Esto creará un controlador con funciones ya establecidas y además rutas asociadas a cada uno de esos métodos. En la imagen podemos apreciar el contenido del controlador



```
1 <?php
2 namespace App\Http\Controllers;
3 use Illuminate\Http\Request;
4
5 class CursoController extends Controller
6 {
7     public function index()
8     {
9         //|
10    }
11    public function create()
12    {
13        //|
14    }
15    public function store(Request $request)
16    {
17        //|
18    }
19    public function show($id)
20    {
21        //|
22    }
23    public function edit($id)
24    {
25        //|
26    }
27    public function update(Request $request, $id)
28    {
29        //|
30    }
31    public function destroy($id)
32    {
33        //|
34    }
}
```

Se puede observar las nuevas rutas asociadas al controlador usando el siguiente comando:

```
php artisan route:list
```

En la imagen vemos el resultado de la ejecución del comando

Domain	Method	URI	Name	Action	Middleware
	GET HEAD	/		Closure	web
	GET HEAD	api/user		Closure	api,auth:api
	GET HEAD	cursos	cursos.index	App\Http\Controllers\CursoController@index	web
	POST	cursos	cursos.store	App\Http\Controllers\CursoController@store	web
	GET HEAD	cursos/create	cursos.create	App\Http\Controllers\CursoController@create	web
	GET HEAD	cursos/{curso}	cursos.show	App\Http\Controllers\CursoController@show	web
	PUT PATCH	cursos/{curso}	cursos.update	App\Http\Controllers\CursoController@update	web
	DELETE	cursos/{curso}	cursos.destroy	App\Http\Controllers\CursoController@destroy	web
	GET HEAD	cursos/{curso}/edit	cursos.edit	App\Http\Controllers\CursoController@edit	web



## Capítulo 6. VISTAS. PARTE 1

### 6.1.- Creado Vistas

Las vistas contienen el HTML servido por su aplicación y separa la lógica de su controlador de la de presentación. Las vistas se almacenan en el directorio resources / views. Una vista simple podría un código HTML como el siguiente:

```
<html>
  <body>
    <h1>Hola Mundo</h1>
  </body>
</html>
```

Nuestras vistas van a estar creadas en código HTML normal.

Para crear una vista no se necesita más que crear con cualquier editor de texto un archivo en el directorio resources/views , dicho archivo debe poseer un nombre como el que sigue:

nombreVista.blade.php

El archivo debe llevar en su nombre el .blade debido a que este es en motor de plantillas de Laravel que se explicará mas adelante

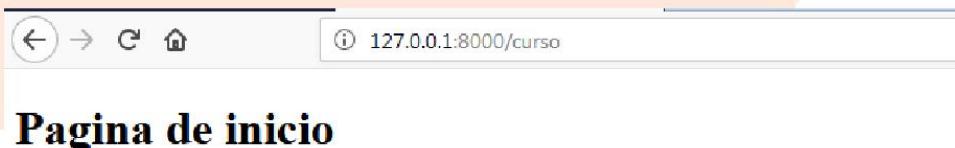


## 6.2.- Vista- Controlador

Las vistas han de ser llamadas dependiendo de la ruta a la que se esta accediendo, es por esto que debe ser el controlador que mediante sus métodos quien retorne las vistas necesarias. Vamos a llamar a la vista 'PrimeraVista' en el método "index" del controlador cursoController gracias a la función reservada "view".

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
  
class cursoController extends Controller  
{  
    public function index()  
    {  
        return view("primeraVista");  
    }  
}
```

Recordemos que para poder acceder al metodo "index" del controlador debemos llamar a la ruta 127..0.0.1:8000/curso, recordemos que es la ruta resource que establecimos en el archivo web.php para el controlador que se está usando que en este caso es cursoController. La salida en el navegador se muestra en la imagen:



### 6.3.- Pasando Datos a Las Vistas

Por tratarse Laravel de un framework de PHP, indiscutiblemente existe la forma de crear vistas que no solo muestren información estática como lo hemos visto hasta ahora, sino que podamos mostrar información dinámica que venga de algún método de un controlador posiblemente desde la base de datos. Es por esto que existen algunas maneras de pasar datos a nuestras vistas. Una manera es con ayuda del método "with" como sigue:

```
view()->with()
```

En la imagen vemos un ejemplo del uso de este método

```
/*
public function index()
{
    return view("primeraVista")->with("curso", "Laravel");
}
```

Luego de pasar el dato a la vista, tenemos la tarea de mostrarlo en la vista. La información que viene del controlador debe ser mostrar en la vista encerrada entre doble llaves

```
{{ $curso }}
```

Se utiliza el signo de dólar porque realmente es una variable lo que estamos intentando mostrar.

```
<html>
  <head>
    <meta charset="UTF-8">
    <title></title>
  </head>
  <body>

    <p>El nombre del curso es:{{$curso}}</p>

  </body>
</html>
```

← → ⌂ ⌂ ① 127.0.0.1:8000/curso

El nombre del curso es:Laravel

En algún momento necesitaremos pasar más de un dato a la vista, por lo que podemos usar un arreglo asociativo en el cual podamos pasar una gran cantidad de variables. El arreglo debe ser pasado como segundo parámetro en el view() .

```

^/
public function index() controlador
{
    $nombre1="Framework";
    $nombre2="Laravel";
    return view("primeraVista", ["dato1"=>$nombre1, "dato2"=>$nombre2]);
}

// **



---



| <body> | Vista                                                 |
|--------|-------------------------------------------------------|
|        | <p>El nombre del curso es:{{\$dato1}} {{\$dato2}}</p> |
|        | </body>                                               |


```

La función "compact()" nos da un mejor manejo del paso de datos a las vistas. Esta función también debe ser llamado como segundo parámetro del view().

`view("vista",compact("var1","var2","...","varN"));`

```

public function index()
{
    $nombre1="Framework";
    $nombre2="Laravel";
    $niveles=["Conociendo Laravel","Laravel avanzado"];
    return view("primeraVista", compact("nombre1", "nombre2", "niveles"));
}

```

Blade es el motor de creación de plantillas proporcionado por Laravel. Gracias a Blade, se pueden usar instrucciones PHP dentro de las vistas de una manera muy sencilla, lo cual va a facilitar el trabajo de impresión de datos dinámicos dentro del propio HTML. En la imagen podemos observar cómo se puede mostrar el arreglo "niveles" pasado como parámetro anteriormente. En el ejemplo se muestra la directiva @foreach()

Todas las directivas de Blade serán detalladas en los próximos capítulos.

```
<body>

    <p>El nombre del curso es:{{ $nombre1 }} {{ $nombre2 }}</p>
    <p>Los Nombres de los niveles del curso son:</p>

    <ul>
        @foreach($niveles as $nivel)
            <li>{{$nivel}}</li>
        @endforeach
    </ul>

</body>
```

← → ⌂ ⌂ ① 127.0.0.1:8000/curso

El nombre del curso es:Framework Laravel

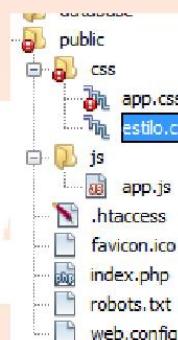
Los Nombres de los niveles del curso son:

## Capítulo 7. VISTAS. PARTE 2

### 7.1.- Agregando Hojas de Estilo

En Laravel, las hojas de estilo y archivos de Javascript tienen lugar en el directorio public en sus respectivas carpetas:

public/css  
public/js



Para hacer el llamado a una hoja de estilo en una vista, se debe usar el helper asset(). Este será la ruta base hacia la carpeta public. Una hoja de estilo llamada "estilo.css" debería llamarse de la siguiente manera:

```
<link rel="stylesheet" type="text/css" href="{{asset('css/estilo.css')}}">
```

### 7.2.- Heredando Vistas

Dos de los principales beneficios de usar Blade son la herencia de la plantilla y las secciones. Para comenzar, echemos un vistazo a un ejemplo simple. Primero, se examinará un diseño de página "maestro". Dado que la mayoría de las aplicaciones web mantienen el mismo diseño general en varias páginas, es conveniente definir este diseño como una sola vista de Blade. La siguiente directiva de blade es útil en la definición de una vista maestra:

`@yield()` : Mostrar el contenido de una sección. Define un bloque en la vista, en el cual se podrá colocar contenido en las vistas hijas.

En la imagen se aprecia una vista master.blade.php:

```
<!doctype html>
<html>
  <head>
    <title>@yield('titulo')</title>
    <link rel="stylesheet" type="text/css" href="{{ asset('css/estilo.css') }}">
  </head>
  <body>
    <div class="container">
      @yield('contenido')
    </div>
  </body>
</html>
```

`@extends` será la directiva de blade para que una vista pueda heredar de otra.

En las vistas hijas se deben crear `@section` que correspondan con los nombres de los `@yield` de la vista padre. En la imagen se muestra un ejemplo de una vista que hereda de la vista master y que en la sección "contenido" agrega un título principal

```
@extends('layout.master')
@section('titulo')
    Vistas heredadas
@endsection
@section('contenido')
    <h2>Laravel CADIF1</h2>
@endsection
```

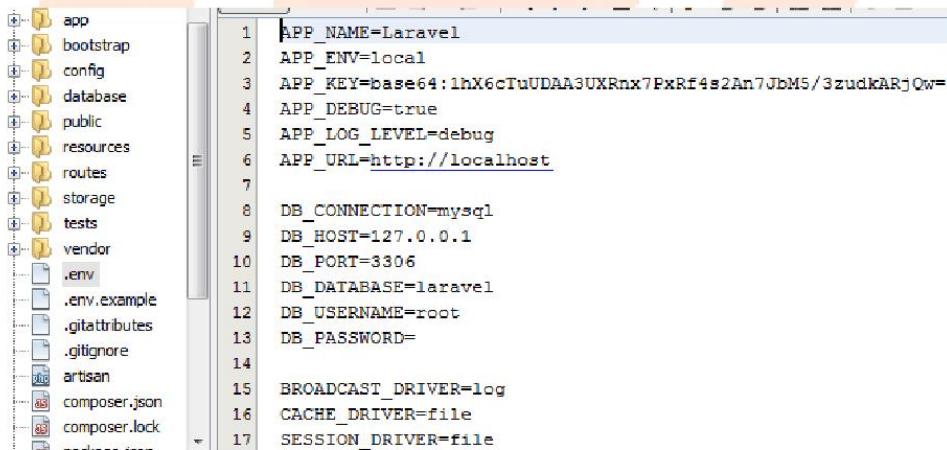
## Capítulo 8. CONFIGURACIÓN DE LA BD

### 8.1.- Archivo de Configuración

Laravel permite crear tablas en la base de datos mediante una interfaz orientada a objetos, de esta manera se estará creando un sistema compatible con las distintas bases de datos que soporta Laravel por defecto. Los motores de bases de datos que soporta Laravel son:

- MySQL
- Postgres
- SQLite
- SQL Server

Para establecer la conexión a una base de datos, Laravel contiene un archivo de configuración donde se debe colocar la información respectiva de la configuración de la base de datos. El archivo .env ubicado en la raíz de la aplicación es el archivo que se debe editar. En la imagen se muestra el archivo .env



The screenshot shows the file structure of a Laravel application. On the left, there's a tree view of files and folders: app, bootstrap, config, database, public, resources, routes, storage, tests, vendor, .env, .env.example, .gitattributes, .gitignore, artisan, composer.json, and composer.lock. The .env file is selected and shown in a code editor window on the right. The code in the .env file is as follows:

```
APP_NAME=Laravel
APP_ENV=local
APP_KEY=base64:1hX6cTuUDAA3UXRnx7PxRf4s2An7JbM5/3zudkARjQw=
APP_DEBUG=true
APP_LOG_LEVEL=debug
APP_URL=http://localhost

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=

BROADCAST_DRIVER=log
CACHE_DRIVER=file
SESSION_DRIVER=file
```

Los parámetros del archivo .env que se deben manipular para establecer una conexión con la base de datos son:

- DB\_CONNECTION (Manejador de BD)
- DB\_HOST (IP del servidor de BD)
- DB\_PORT (Puerto de conexión)
- DB\_DATABASE (Nombre de la base de datos)
- DB\_USERNAME (Usuario de la BD)
- DB\_PASSWORD (Contraseña de usuario de la BD)

Un ejemplo de los posibles valores que se pueden asignar a estos parámetros son:

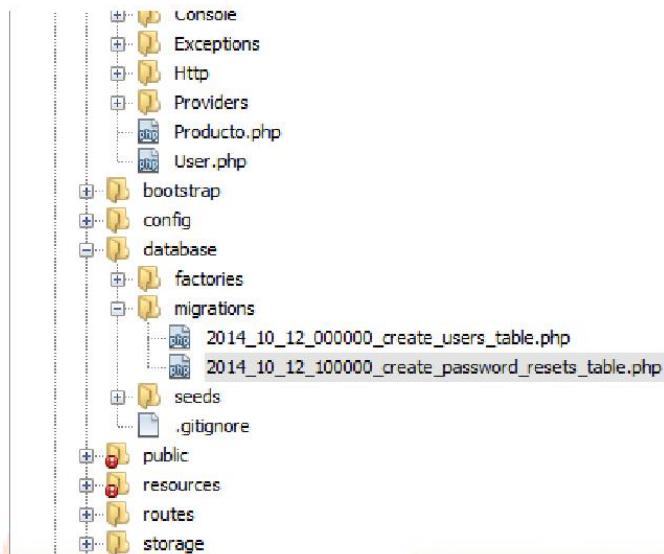
```
DB_CONNECTION=mysql  
DB_HOST=127.0.0.1  
DB_PORT=3306  
DB_DATABASE=laravel  
DB_USERNAME=root  
DB_PASSWORD=
```

## 8.2.- Migraciones

Las migraciones son el control de versiones para la base de datos, lo que le permite a su equipo modificar y compartir fácilmente el esquema de la base de datos de la aplicación. Las migraciones suelen combinarse con el generador de esquemas de Laravel para crear fácilmente el esquema de la base de datos de la aplicación.

Las migraciones se encuentran en el directorio database/migrations.

Laravel trae predefinidas 2 migraciones que se muestran en la siguiente imagen:



Las migraciones serán las maquetas para crear las tablas en la base de datos, en otras palabras, será en las migraciones donde se harán los cambios necesarios para cada tabla.

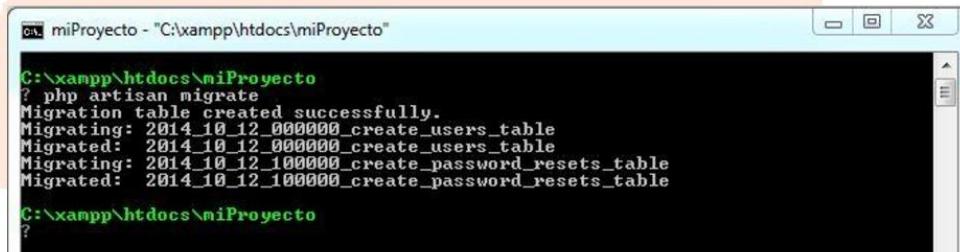
Ya configurada la conexión a la base de datos es suficiente para comenzar a pasar las migraciones e ir creando las tablas necesarias.

En la siguiente imagen se muestra un ejemplo de una migración:

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

class CreateUsersTable extends Migration
{
    public function up()
    {
        Schema::create('users', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('email')->unique();
            $table->string('password');
            $table->rememberToken();
            $table->timestamps();
        });
    }
    public function down()
    {
        Schema::dropIfExists('users');
    }
}
```

Con ayuda de artisan se puede usar el comando "migrate" para hacer la migración inicial como lo muestra la imagen:



The screenshot shows a terminal window titled "miProyecto - C:\xampp\htdocs\miProyecto". The command "php artisan migrate" is run, and the output shows the successful creation of three tables: users, password\_resets, and password\_resets\_table.

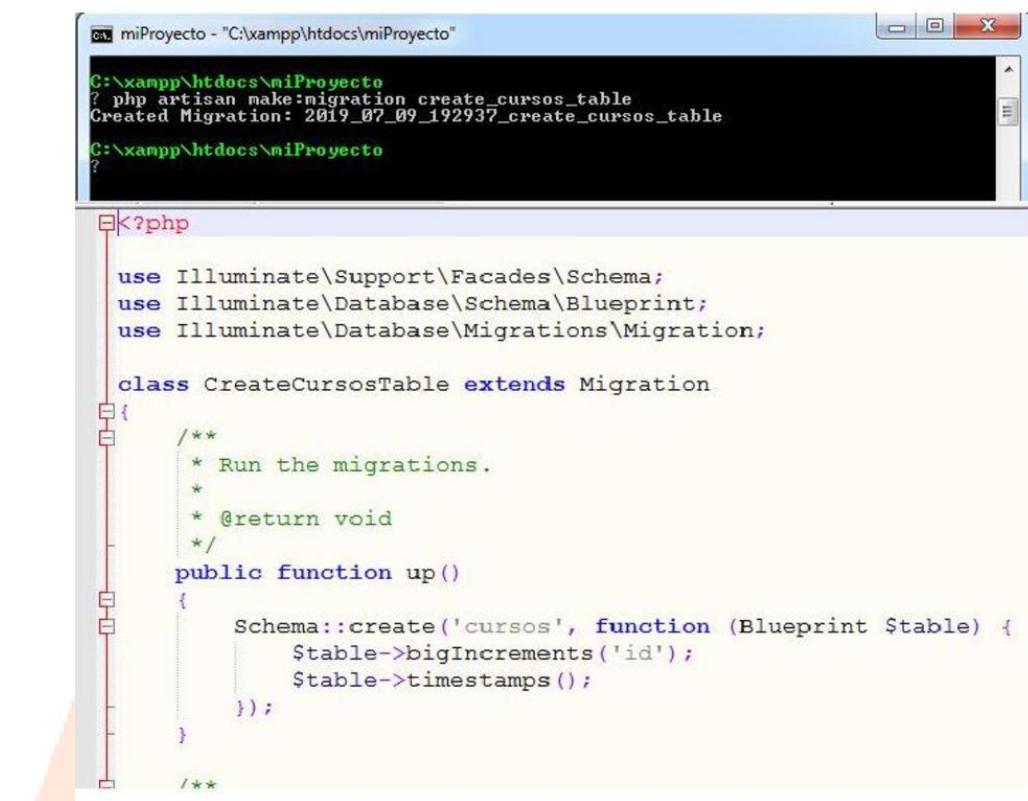
```
C:\xampp\htdocs\miProyecto
? php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table
C:\xampp\htdocs\miProyecto
?
```

Los nombres de las migraciones generalmente se escriben de la siguiente manera:

```
create_nombretabla_table
```

Para crear una nueva migración con el objetivo de ir creando nuevas tablas en la base de datos. Se utiliza el comando make:migration como sigue:

```
php artisan make:migration create_nombretabla_table --create=nombretabla
```



```
C:\xampp\htdocs\miProyecto
? php artisan make:migration create_cursos_table
Created Migration: 2019_07_09_192937_create_cursos_table
C:\xampp\htdocs\miProyecto
?

?php

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

class CreateCursosTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('cursos', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->timestamps();
        });
    }
    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::dropIfExists('cursos');
    }
}
```

Las migraciones tienen comandos asociados para ser gestionadas de una manera sencilla. Estos comandos son los siguientes:

- MIGRATE:FRESH. Para borrar todas las tablas de la base de datos y luego volverlas a crear.
- MIGRATE:RESET. Para eliminar todas las migraciones.
- MIGRATE:ROLLBACK. Permite deshacer el último grupo de migraciones ejecutadas.
- MIGRATE:STATUS. Para ver el estatus de cada migración.

Una vez creada la nueva migración, se debe ejecutar el comando "php artisan migrate:fresh" para actualizar la base de datos.



```
miProyecto - "C:\xampp\htdocs\miProyecto"
Created Migration: 2019_07_09_192937_create_cursos_table
C:\xampp\htdocs\miProyecto
? php artisan migrate:fresh
Dropped all tables successfully.
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table
Migrating: 2019_07_09_192937_create_cursos_table
Migrated: 2019_07_09_192937_create_cursos_table
C:\xampp\htdocs\miProyecto
?
```

## Capítulo 9. QUERY BUILDER

### 9.1.- Definición

El generador de consultas de base de datos de Laravel proporciona una interfaz conveniente y fluida para crear y ejecutar consultas de bases de datos. Se puede usar para realizar la mayoría de las operaciones de bases de datos en su aplicación y funciona en todos los sistemas de bases de datos compatibles.

El generador de consultas de Laravel utiliza el enlace de parámetros de PDO para proteger su aplicación contra los ataques de inyección de SQL. No es necesario limpiar las cadenas que se pasan como enlaces.

### 9.2.- Recuperando Tablas

Para recuperar todas las filas de una tabla puede usar el método "table" de la clase DB. El método "table" devuelve una instancia de generador de consultas con fluidez para la tabla dada, lo que le permite encadenar más restricciones a la consulta y finalmente obtener los resultados mediante el método get.

Debemos cargar datos iniciales en la tabla que vamos a consultar:

```
<?php Controlador
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class CursoController extends Controller
{
    /**
     * Display a listing of the resource.
     *
     * @return \Illuminate\Http\Response
     */
    public function index()
    {
        $cursos = DB::table('cursos')->get();

        return view('cursos.index', compact('cursos'));
    }
}

@extends('app') Vista
@section('contenido')



| Titulo                | Descripción                |
|-----------------------|----------------------------|
| {{ \$curso->titulo }} | {{ \$curso->descripcion }} |


@endsection
```

Para poder usar el facade DB no debemos olvidar la siguiente instrucción en nuestro controlador.

```
use Illuminate\Support\Facades\DB;
```

Los facades en Laravel proporcionan una interfaz “Estática” a las clases que están disponibles a través del contenedor de servicios de la aplicación. De esta manera podemos llamar a métodos sin tener que preocuparnos por instanciar la clase. En el ejemplo anterior, la vista esta mostrando el listado de cursos, como se muestra en la imagen:



A screenshot of a web browser window displaying a table of courses. The URL in the address bar is 'localhost/miProyecto/public/cursos'. The table has two columns: 'Titulo' and 'Descripción'. There are two rows of data: one for 'PHP' (curso de programación en php) and another for 'HTML5' (curso de maquetado web).

Titulo	Descripción
PHP	curso de programación en php
HTML5	curso de maquetado web

### 9.3.- Recuperando Datos Específicos

Existen métodos con los cuales se puede obtener información más detallada de las tablas usando el QueryBuilder. Métodos tales como:

- `first()` : trae el primer registro de la consulta
- `count()`: Cantidad de elementos de la consulta
- `max()`: Obtener el elemento con valor máximo en alguno de sus campos
- `where()`: Permite especificar el campo de y valor de búsqueda de la consulta.
- `pluck()`: Recuperar una Colección que contiene los valores de una sola columna
- Todos los métodos disponibles pueden ser revisados en la documentación oficial de Laravel (<https://laravel.com/docs/5.8/queries>)

Por ejemplo, si se quiere obtener de la tabla USERS el registro cuyo columna NAME contiene el valor JOHN, o si se quiere obtener un arreglo con todos los valores de la columna TITLE de la tabla ROLES, las instrucciones de cada uno de los casos se muestran en la imagen:

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

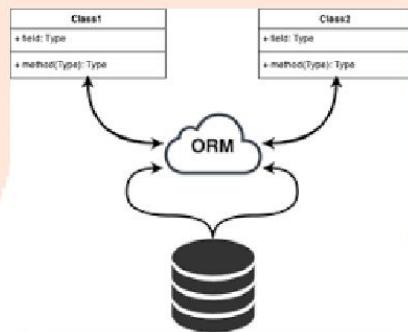
```
$titles = DB::table('roles')->pluck('title');
```



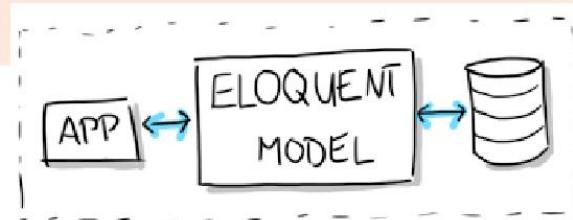
## Capítulo 10. ELOQUENT

### 10.1.- Definición

El mapeo objeto-relacional (más conocido por su nombre en inglés, Object-Relational mapping, o sus siglas O/RM, ORM, y O/R mapping) es una técnica de programación para convertir los registros de una base de datos relacional en objetos usados en la programación orientada a objetos.



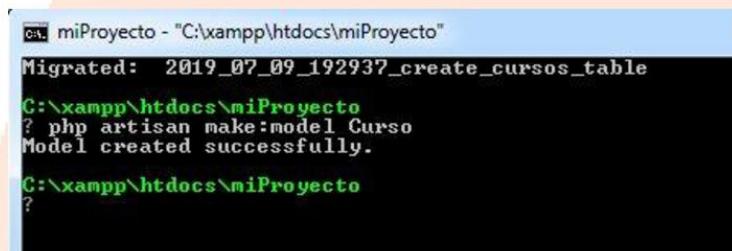
El ORM Eloquent incluido en Laravel proporciona una implementación de ActiveRecord. Cada tabla de la base de datos tiene un "Modelo" correspondiente, que se utiliza para interactuar con esa tabla. Los modelos permiten consultar datos en las tablas, así como insertar nuevos registros, modificarlos y eliminarlos.



## 10.2.- Modelos

Para comenzar, se crea un modelo Eloquent. Los modelos generalmente viven en el directorio de la aplicación, pero puede colocarlos en cualquier lugar que pueda cargarse automáticamente de acuerdo con su archivo composer.json. Todos los modelos Eloquent heredan de la clase Illuminate Database Eloquent Model.

El nombre de los modelos debe comenzar con mayúscula y estar en singular, a diferencia de la tabla de la base de datos a la cual el va a representar la cual debe estar escrita en minúsculas y en plural (Convención Laravel). En la imagen se muestra la creación de un modelo:



A screenshot of a terminal window titled "miProyecto - "C:\xampp\htdocs\miProyecto"". The window shows the output of a command: "Migrated: 2019\_07\_09\_192937\_create\_cursos\_table". Below this, the command "php artisan make:model Curso" is run, followed by the message "Model created successfully.". The prompt "C:\xampp\htdocs\miProyecto>" is visible at the bottom.

Los modelos van a permitir el mapeo de la base de datos gracias a Eloquent, por lo que vamos a necesitar tanto una migración como un controlador de cada modelo. Para crear la migración y el controlador de recursos del modelo de forma sencilla, se puede correr el siguiente comando:

```
Php artisan make:model NombreModelo --mc --resource
```

```
miProyecto - "C:\xampp\htdocs\miProyecto"  
C:\xampp\htdocs\miProyecto  
? php artisan make:model Curso -mc --resource  
Model created successfully.  
Created Migration: 2019_07_09_195443_create_cursos_table  
Controller created successfully.  
C:\xampp\htdocs\miProyecto  
?  
miProyecto  
├── app  
│   ├── Console  
│   ├── Exceptions  
│   ├── Http  
│   │   └── Controllers  
│   │       ├── Auth  
│   │       │   ├── Controller.php  
│   │       │   └── CursoController.php  
│   │       └── Middleware  
│   │           └── Kernel.php  
│   ├── Providers  
│   │   ├── Curso.php  
│   │   └── User.php  
│   ├── bootstrap  
│   ├── config  
│   └── database  
│       ├── factories  
│       └── migrations  
│           ├── 2014_10_12_000000_create_users_table.php  
│           ├── 2014_10_12_100000_create_password_resets_table.php  
│           └── 2019_07_09_195443_create_cursos_table.php  
└── seeds
```

Debido a que Eloquent esta haciendo un mapeo de la de base de datos para asociarla a los modelos, toda instancia de un modelo tendrá asociado métodos que permitirán tanto obtener datos como modificarlos. Los métodos son:

- `save()`: Permite guardar un objeto dentro de una tabla.
- `all()`: Permite recuperar todos los registros de la tabla.
- `find()`: Permite recuperar un registro mediante su identificador único.

- pluck(): Solicitar valores de alguna columna en específico.

### 10.3.- Tinker

Para interactuar por primera vez con los modelos asociados a las tablas de la base de datos, puede hacer uso de la consola de Laravel "Tinker". Tinker es la herramienta de Laravel que permite escribir código PHP por consola. Para invocar la consola solo debe ejecutar el comando:

```
php artisan tinker
```

Esta consola se usa para crear nuevos registros en las tablas asociadas a los modelos, aunque no será esta la manera habitual de hacerlo.



```
C:\xampp\htdocs\miProyecto
? php artisan tinker
Psy Shell v0.9.9 (PHP 7.2.11 Ôçö cli) by Justin Hileman
>>>
```

Para crear un nuevo registro, debe instanciar un objeto de una de las clases definidas (modelos) en la aplicación, cargar el objeto y finalmente usar el método save() de Eloquent para guardar en la base de datos. La imagen muestra la creación de un nuevo registro en la tabla "cursos":



```
C:\ miProyecto - "C:\xampp\htdocs\miProyecto" - php artisan tinker
C:\xampp\htdocs\miProyecto
php artisan tinker
Psy Shell v0.9.9 (PHP 7.2.11 Ôçö cli) by Justin Hileman
>>> $curso =new App\Curso()
=> App\Curso {#2972}
>>> $curso->titulo = "Laravel 5.8"
=> "Laravel 5.8"
>>> $curso->descripcion = "Este es un master en Laravel"
=> "Este es un master en Laravel"
>>> $curso->save()
=> true
>>>
```

El método all() permitirá retornar todos los registros de la tabla:

```
>>> $cursos = App\Curso::all()
=> Illuminate\Database\Eloquent\Collection {#2978
    all: [
        App\Curso {#2979
            id: 1,
            titulo: "PHP",
            descripcion: "curso de programaci\u00f3n en php",
            created_at: "2019-07-09 00:00:00",
            updated_at: "2019-07-09 00:00:00",
        },
        App\Curso {#2980
            id: 2,
            titulo: "HTML5",
            descripcion: "curso de maquetado web",
            created_at: "2019-07-09 00:00:00",
            updated_at: "2019-07-09 00:00:00",
        },
        App\Curso {#2981
            id: 3,
            titulo: "Laravel 5.8",
            descripcion: "Este es un master en Laravel",
            created_at: "2019-07-09 20:02:59",
            updated_at: "2019-07-09 20:02:59",
        },
    ],
}>>>
```

## Capítulo 11. LEER REGISTROS

### 11.1.- Index

En la imagen se muestra un código html que genera una tabla de cursos, cada curso es un enlace donde su atributo href contiene la ruta para invocar el método show del controlador de cursos:

The screenshot shows a code editor with two tabs open:

- CursoController.php** (Controlador): This file contains PHP code for a controller. It includes imports for the Illuminate\Http\Request and App\Curso classes, and defines an index() method that retrieves all courses from the database and returns a view named 'cursos.index'.
- index.blade.php** (Vista): This file is a Blade template. It extends the 'layouts.app' layout and defines a section named 'contenido'. Inside this section, it displays a table with course titles and descriptions. For each course, it generates a table row containing three columns: the course title, the course description, and a link labeled 'Ver detalle' that points to the 'show' method of the 'CursosController' via the route('cursos.show', \$curso) dynamic segment.

```

CursoController.php
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use Illuminate\Http\Request;
6 use App\Curso;
7
8 class CursosController extends Controller
9 {
10
11     /**
12      * Display a listing of the resource.
13      *
14      * @return \Illuminate\Http\Response
15     */
16     public function index()
17     {
18         $cursos = Cursos::get();
19
20         return view('cursos.index', compact('cursos'));
21     }
22 }

index.blade.php
1 @extends('layouts.app')
2
3 @section('contenido')
4     <h1>Lista de Cursos</h1>
5     <table border="1">
6         <tr>
7             <th>Título</th>
8             <th>Descripción</th>
9             <th></th>
10        </tr>
11        @foreach($cursos as $curso)
12            <tr>
13                <td>{{ $curso->titulo }}</td>
14                <td>{{ $curso->descripcion }}</td>
15                <td><a href="{{ route('cursos.show', $curso) }}>Ver detalle</a></td>
16            </tr>
17        @endforeach
18    </table>
19 @endsection

```

## 11.2.- Ruta Show

Para recuperar un registro específico y mostrar algún detalle, se debe tener un enlace que apunte a la ruta que activa el método show del controlador, el cual recibe como parámetro el id del registro que se desea buscar. Siguiendo con el ejemplo anterior, la ruta es la siguiente:

cursos/{curso}

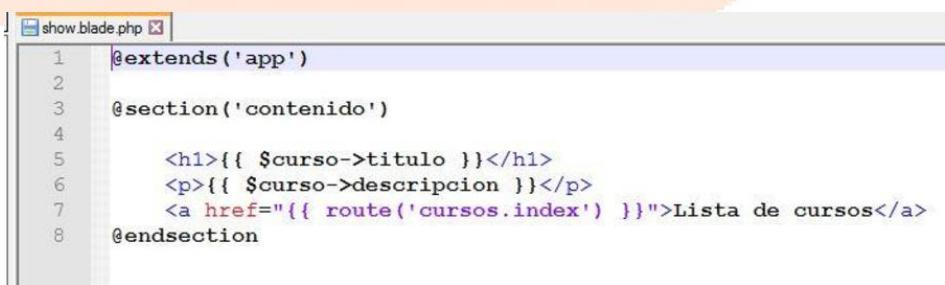
Se puede hacer uso del nombre establecido para esa ruta el cual es productos.show  
Se utiliza el helper "route" de Laravel para llamar a las rutas por su nombre.

## 11.3.- Método Show

El método show del controlador debe retornar la vista donde se van mostrar los datos del registro que se está buscando. Gracias al Route Model Binding, la vinculación del modelo de ruta proporciona una forma útil de injectar automáticamente al método una instancia del modelo en los cierres de ruta:

```
public function show(Curso $curso)
{
    return view('cursos.show', compact('curso'));
}
```

La vista encargada para mostrar el registro específico se muestra a continuación



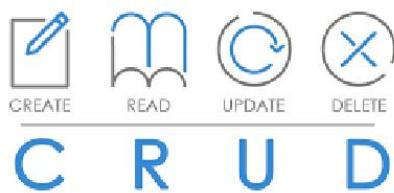
```
show.blade.php
1 @extends('app')
2
3 @section('contenido')
4
5     <h1>{{ $curso->titulo }}</h1>
6     <p>{{ $curso->descripcion }}</p>
7     <a href="{{ route('cursos.index') }}">Lista de cursos</a>
8 @endsection
```

## Capítulo 12. AGREGAR REGISTROS

### 12.1.- Introducción

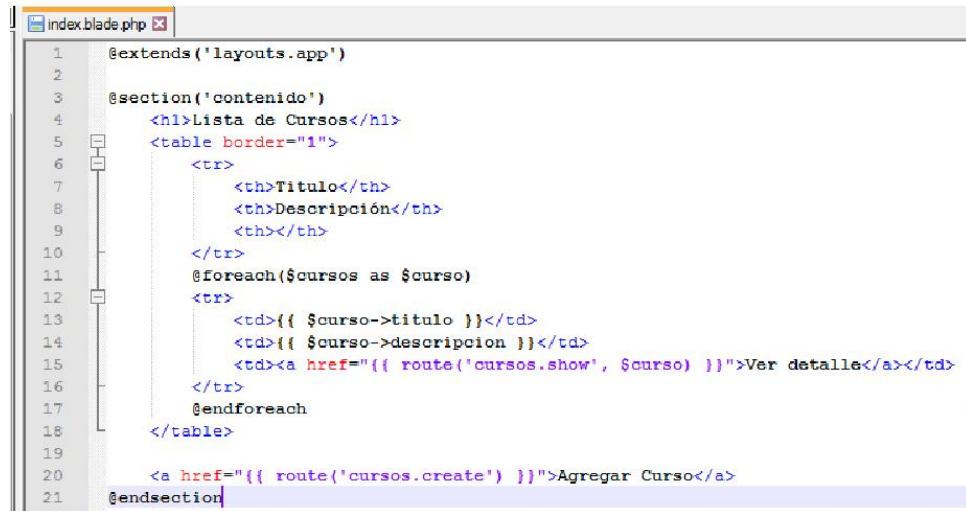
En informática, CRUD es el acrónimo de "Crear, Leer, Actualizar y Borrar" (del original en inglés: Create, Read, Update and Delete), es usado para referirse a las funciones básicas en bases de datos o la capa de persistencia en un software.

Para crear nuevos registros en tablas de la base de datos haciendo uso de los modelos y controladores, debe tener un método en el controlador que se encargue de recibir datos desde un formulario hecho para crear nuevos registros, de manera que reciba los datos y con ayuda del modelo y Eloquent usar el método `save()` para registrar la información pertinente.



### 12.2.- Index

Ahora en la vista index que lista los cursos, agregamos un enlace cuyo atributo href es la ruta hacia el método `create` del controlador de cursos.



```

1  @extends('layouts.app')
2
3  @section('contenido')
4      <h1>Lista de Cursos</h1>
5      <table border="1">
6          <thead>
7              <tr>
8                  <th>Título</th>
9                  <th>Descripción</th>
10                 <th></th>
11             </tr>
12         <tbody>
13             @foreach($cursos as $curso)
14                 <tr>
15                     <td>{{ $curso->título }}</td>
16                     <td>{{ $curso->descripción }}</td>
17                     <td><a href="{{ route('cursos.show', $curso) }}">Ver detalle</a></td>
18                 </tr>
19             @endforeach
20         </tbody>
21     </table>
22
23     <a href="{{ route('cursos.create') }}">Agregar Curso</a>
24
25     @endsection

```

### 12.3.- Método Create

Para crear debemos usar el método create y store de nuestro controlador. El create se encargará simplemente de mostrar la vista con un formulario para captar los datos y el store será el método que invocaremos para que haga el guardado en la base de datos.

```

public function create()
{
    return view('cursos.create');
}

```

### 12.4.- Formulario

La vista de captura de datos debe estar compuesta por un formulario, el cual debe tener en el atributo action una referencia a la ruta de que invoca el método store. Los formularios deben enviar un token de seguridad en cada envío, esto como parte de la seguridad de la aplicación. Laravel nos provee de un helper y una directiva blade para colocar este token de seguridad dentro del formulario, por lo que podemos utilizar uno de los dos.

---

`{{ csrf_field() }}` o `@csrf`

Este token nos va a proteger de ataques mal intencionados que puedan existir hacia nuestra aplicación

The screenshot shows a code editor window titled 'create.blade.php' containing the following PHP Blade template code:

```
1 @extends('layouts.app')
2
3 @section('contenido')
4     <h1>Nuevo Curso</h1>
5
6     <form action="{{ route('cursos.store') }}" method="POST">
7
8         @csrf
9
10        <label>Titulo</label><br/>
11        <input type="text" name="titulo"><br/>
12
13        <label>Descripción</label><br/>
14        <textarea name="descripcion" rows="5" cols="50"></textarea><br/>
15
16        <button type="submit">Guardar</button>
17
18    </form>
19 @endsection
```

Below the code editor is a browser address bar showing the URL: `localhost/miProyecto/public/cursos/create`.

## Nuevo Curso

The screenshot shows a web form titled 'Nuevo Curso'. It contains two input fields: 'Titulo' (Title) and 'Descripción' (Description). Below the fields is a 'Guardar' (Save) button.

La función `redirect()` permite hacer una redirección a cualquier ruta de la aplicación. Es por esto que al finalizar un método del controlador, podría ser necesario redirigir al usuario a alguna de las vistas.

En la imagen se muestra el ejemplo del código del método `store` dentro del controlador de cursos:

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

    $curso->titulo = $request->titulo;
    $curso->descripcion = $request->descripcion;
    $curso->save();

    return redirect()->route('cursos.index');
}
```

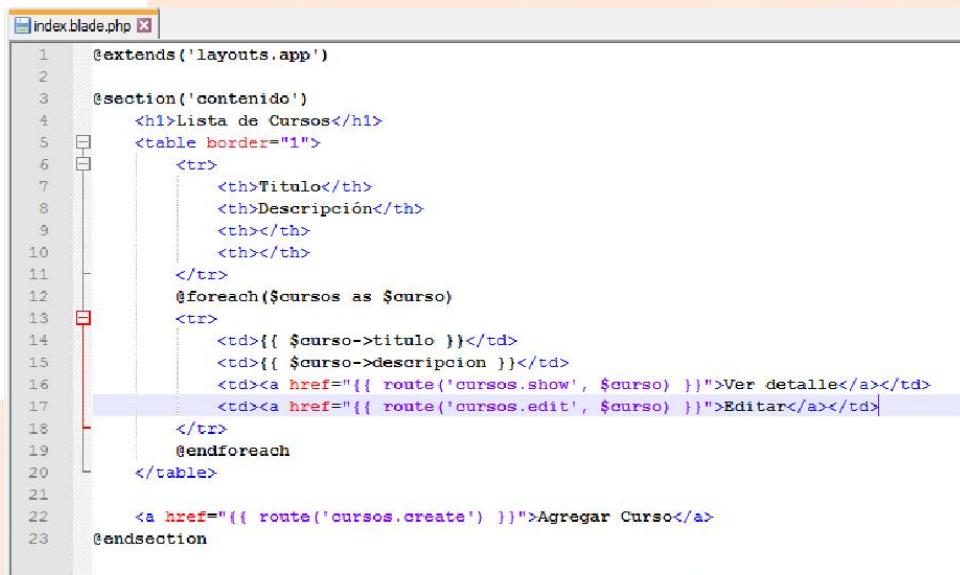


## Capítulo 13. MODIFICAR REGISTROS

### 13.1.- Introducción

Actualizar registros en Laravel hace uso de los métodos “edit” y “update” del controlador. El método edit será el encargado de mostrar el formulario para la captura de datos modificados, y será el método update del controlador el que se encargue de recibir los datos desde el formulario de edición para proceder a guardar los cambios.

Se debe tener un enlace que apunte a la ruta definida para la edición, esta ruta ya fue creada al momento de crear una ruta de recursos.



```

1 @extends('layouts.app')
2
3 @section('contenido')
4     <h1>Lista de Cursos</h1>
5     <table border="1">
6         <tr>
7             <th>Título</th>
8             <th>Descripción</th>
9             <th></th>
10            <th></th>
11        </tr>
12        @foreach($cursos as $curso)
13            <tr>
14                <td>{{ $curso->titulo }}</td>
15                <td>{{ $curso->descripcion }}</td>
16                <td><a href="{{ route('cursos.show', $curso) }}>Ver detalle</a></td>
17                <td><a href="{{ route('cursos.edit', $curso) }}>Editar</a></td>
18            </tr>
19        @endforeach
20    </table>
21
22    <a href="{{ route('cursos.create') }}>Agregar Curso</a>
23 @endsection

```

### 13.2.- Métodos Edit y Update

El método edit del controlador va a ser el encargado de mostrar la vista para la edición. Laravel necesita el token de seguridad en todos sus formularios, así que acá no es la excepción. También debemos saber que debemos pasar como campo oculto el tipo de petición que se hará al servidor, en este caso es el método “PUT”.

Se usan los siguientes helpers:

```
 {{csrf_field()}}
```

```
 {{method_field("PUT")}}
```

también

```
@csrf  
@method("PUT")
```



```
controlador
public function edit(Curso $curso)
{
    return view('cursos.edit', compact('curso'));
}

edit.blade.php vista
1 <extends('layouts.app')>
2
3 @section('contenido')
4     <h1>Modificar Curso</h1>
5
6     <form action="{{ route('cursos.update', $curso) }}" method="POST">
7
8         @csrf
9         @method('PUT')
10
11         <label>Titulo</label><br/>
12         <input type="text" name="titulo" value="{{ $curso->titulo }}><br/>
13
14         <label>Descripción</label><br/>
15         <textarea name="descripcion" rows="5" cols="50">{{ $curso->descripcion }}</textarea><br/>
16
17         <button type="submit">Guardar</button>
18
19     </form>
20 @endsection
```

El método update del controlador recibe los datos, actualiza el objeto y almacena los cambios en la base de datos.

```
public function update(Request $request, Curso $curso)
{
    $curso->titulo = $request->titulo;
    $curso->descripcion = $request->descripcion;
    $curso->save();

    return redirect()->route('cursos.index');
}
```



## Capítulo 14. ELIMINAR REGISTROS

### 14.1.- Introducción

La letra D del CRUD significa "DELETE". En laravel podemos eliminar o destruir nuestros elementos de modelos usando el método delete, el cual puede ser usado de la siguiente manera:

```
$objeto->delete()
```

### 14.2.- Formulario de Eliminado

Para proceder a eliminar registros de nuestras tablas, se ha de contar con un formulario que haga una petición a la ruta que invoque el método destroy de nuestro controlador, sin olvidar pasar un parámetro que será el id del elemento.

El token es obligatorio en el uso de todos los envíos de datos en nuestra aplicación, además en este caso se debe pasar como un campo oculto el método usado para eliminación en Laravel como lo es el método "DELETE"

```

1  @extends('layouts.app')
2
3  @section('contenido')
4      <h1>Lista de Cursos</h1>
5      <table border="1">
6          <thead>
7              <tr>
8                  <th>Título</th>
9                  <th>Descripción</th>
10                 <th></th>
11                 <th></th>
12                 <th></th>
13             </tr>
14         @foreach($cursos as $curso)
15             <tr>
16                 <td>{{ $curso->titulo }}</td>
17                 <td>{{ $curso->descripcion }}</td>
18                 <td><a href="{{ route('cursos.show', $curso) }}>Ver detalle</a></td>
19                 <td><a href="{{ route('cursos.edit', $curso) }}>Editar</a></td>
20                 <td>
21                     <form action="{{ route('cursos.destroy', $curso) }}" method="POST">
22                         @csrf
23                         @method('DELETE')
24                         <button type="submit">Eliminar</button>
25                     </form>
26                 </td>
27             </tr>
28         @endforeach
29     </table>
30
31     <a href="{{ route('cursos.create') }}>Agregar Curso</a>
@endsection

```

### 14.3.- Método Destroy

Para proceder a eliminar registros de nuestras tablas, se ha de contar con un formulario que haga una petición a la ruta que invoque el método destroy de nuestro controlador, sin olvidar pasar un parámetro que será el id del elemento.

```

public function destroy(Curso $curso)
{
    $curso->delete();

    return redirect()->route('cursos.index');
}

```

## Capítulo 15. VALIDACIONES

### 15.1.- Método Validate()

Laravel proporciona varios enfoques diferentes para validar los datos entrantes de su aplicación. Por defecto, la clase de controlador base de Laravel usa un rasgo(trait) ValidatesRequests que proporciona un método conveniente para validar la solicitud HTTP entrante con una variedad de poderosas reglas de validación.

El método de validar acepta una solicitud HTTP entrante y un conjunto de reglas de validación. Si se aprueban las reglas de validación, su código seguirá ejecutándose normalmente; sin embargo, si falla la validación, se lanzará una excepción y la respuesta de error correcta se enviará automáticamente al usuario.

Por ejemplo, se puede hacer uso del validate() para asegurarnos de que un formulario para crear nuevos registros no contenga campos vacíos. La regla "required" comprueba si el campo esta vacío.

```
public function store(Request $request)
{
    $this->validate($request, ['titulo' => 'required',
                           'descripcion' => 'required']);

    $curso = new Curso();

    $curso->titulo = $request->titulo;
    $curso->descripcion = $request->descripcion;
    $curso->save();

    return redirect()->route('cursos.index');
}
```

Si algunas de las reglas establecidas en el validate falla, se enviará mensajes de error en un arreglo llamado \$errors, al cual tendremos acceso en la vista que está enviando el request.

Para asegurarnos de mostrar errores de validación en el caso de que existan, debemos asegurarnos imprimiendo el contenido del arreglo \$errors en nuestra vista de la siguiente manera:

```
1 @extends('layouts.app')
2
3 @section('contenido')
4     <h1>Nuevo Curso</h1>
5
6     <ul>
7         @foreach($errors->all() as $error)
8             <li>{{ $error }}</li>
9         @endforeach
10    </ul>
11
12    <form action="{{ route('cursos.store') }}" method="POST">
13
14        @csrf
15
16        <label>Titulo</label><br/>
17        <input type="text" name="titulo"><br/>
18        <label>Descripción</label><br/>
19        <textarea name="descripcion" rows="5" cols="50"></textarea><br/>
20
21        <button type="submit">Guardar</button>
22
23    </form>
24 @endsection
```

## 15.2.- Form Requests

Para escenarios de validación más complejos, es posible que desee crear un Form Request. Los Form Request son clases de solicitud personalizadas que contienen lógica de validación. Las validaciones escritas en estos Form Request pueden ser reutilizadas en toda la aplicación.

Para crear una clase Request, use el comando make:request de Artisan.

```
php artisan make:request Nombre
```

Donde el parámetro nombre es un identificador arbitrario que se le da a la clase. Los Request creados serán almacenados en el directorio app/Http/Request

La clase Request debe contener un método llamado “rules”, en el cual se deben especificar las validaciones igual que en el método validate() del tema anterior.

En la imagen se muestra un ejemplo del método rules()

```
public function rules()
{
    return [
        'title' => 'required|unique:posts|max:255',
        'body' => 'required',
    ];
}
```

Entonces, ¿cómo se evalúan las reglas de validación? Todo lo que necesita es usar el Request en su método de controlador. El request entrante se valida antes de llamar al método del controlador, lo que significa que no necesita saturar su controlador con ninguna lógica de validación:

En el caso de que existan errores de validación, el código del método del controlador no se ejecutará, simplemente se enviarán los errores a la vista desde la cual se están enviando los datos de algún formulario. En la imagen se muestra como el método store del formulario, valida la llegada de los datos mediante un Request llamado "StoreBlogPost"

```
public function store(StoreBlogPost $request)
{
    // The incoming request is valid...
}
```