



**Laravel = PHP + Framework+ MVC. Nivel I**

Febrero, 2018



## 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 nuestra 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 2018. 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. El Patrón Mvc

- 2.1.- Concepto.
- 2.2.- El Patrón Mvc en Aplicaciones Web.
- 2.3.- El Patrón Mvc y Laravel.

### Capítulo 3. Enrutamiento

- 3.1.- Las Rutas.
- 3.2.- El Archivo Route.php.
- 3.3.- Estableciendo Rutas.

### Capítulo 4. Controladores

- 4.1.- Crear el Controlador.
- 4.2.- Enrutar un Método Del Controlador.
- 4.3.- Enrutar un Controlador Completo.

### Capítulo 5. Vistas. Parte 1

- 5.1.- Creado Vistas.
- 5.2.- Vista- Controlador.
- 5.3.- Pasando Datos a Las Vistas.

### Capítulo 6. Configuración de la bd

- 6.1.- Archivo de Configuración.
- 6.2.- Migraciones.
- 6.3.- Querybuilder.

## Capítulo 7. Query Builder

- 7.1.- Definición.
- 7.2.- Recuperando Tablas.
- 7.3.- Recuperando Datos Específicos.

## Capítulo 8. Eloquent

- 8.1.- Definición.
- 8.2.- Modelos.
- 8.3.- Tinker.

## Capítulo 9. Agregar Registros

- 9.1.- Introducción.
- 9.2.- Método Create.
- 9.3.- Formulario.

## Capítulo 10. Leer Registros

- 10.1.- Ruta Show.
- 10.2.- Método Show.

## Capítulo 11. Modificar Registros

- 11.1.- Introducción.
- 11.2.- Métodos Edit y Update.

## Capítulo 12. Eliminar Registros

- 12.1.- Introducción.
- 12.2.- Formulario de Eliminado.
- 12.3.- Método Destroy.

## Capítulo 13. Validaciones en el Modelo

- 13.1.- Método Validate().

## Capítulo 14. Vistas. Parte 2

14.1.- Agregando Hojas de Estilo.

14.2.- Heredando Vistas.



## 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 5.4 que se usó para este curso) hay algunos requisitos (además de tener conocimientos en PHP):

- Servidor web (cualquiera)
- PHP >= 5.6.4
- Habilitar la extensión OpenSSL PHP
- Habilitar la extensión PDO PHP
- Habilitar la extensión Mbstring PHP
- Habilitar la extensión Tokenizer PHP

## 1.2.- Instalación

El repositorio oficial del framework en GitHub 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.

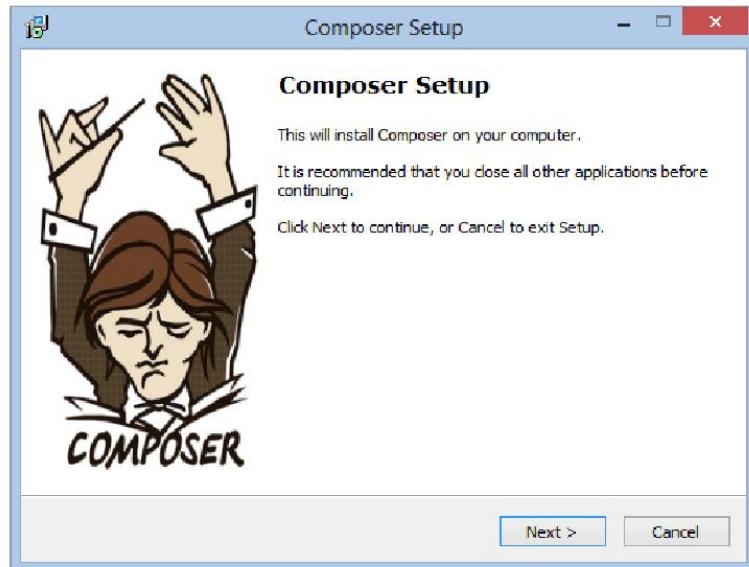
La forma recomendada de hacerlo en la documentación de la página oficial (<https://laravel.com/docs/5.2>) es usando Composer.

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.

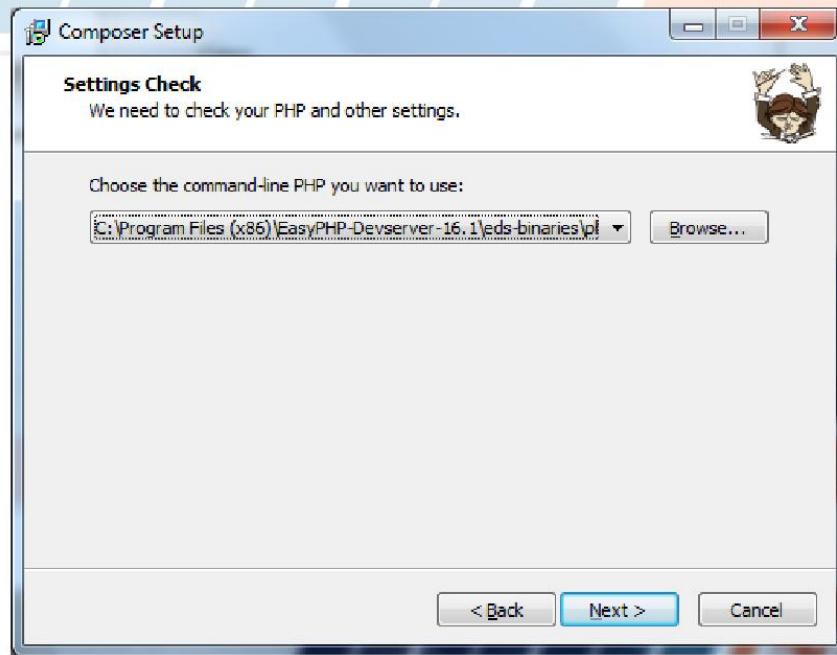
Composer corre en la línea de comandos e instala dependencias para una aplicación. También permite a los usuarios instalar aplicaciones que están disponibles en "Packagist" que es el repositorio principal de paquetes. Se puede visitar el sitio del repositorio en la url <https://packagist.org/>.



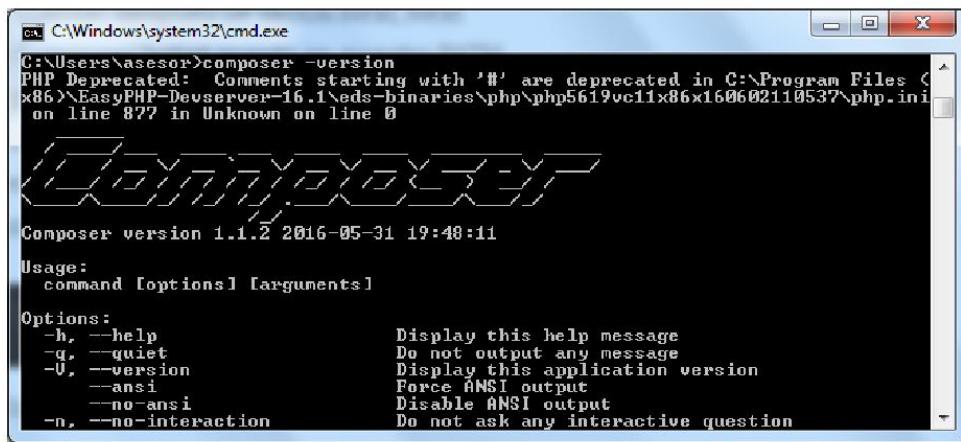
Para instalar Composer debe descargar el instalador desde la página oficial <https://getcomposer.org/>. Para Windows existe un instalador que facilita el trabajo. Antes de instalar Composer debe estar instalado Php (con cualquier paquete como EasyPhp o Wamp).



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



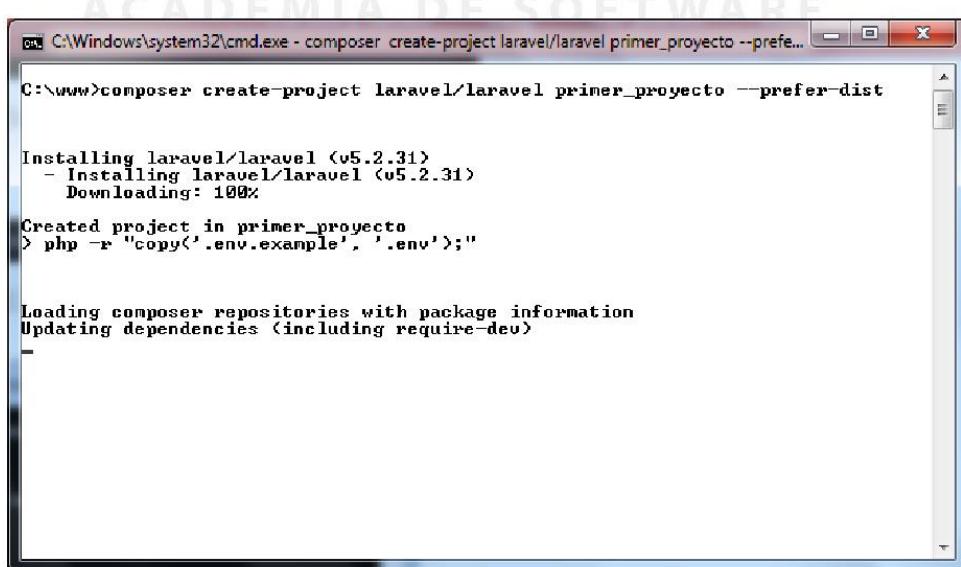
Al finalizar el proceso de instalación se puede comprobar que Composer se haya instalado correctamente abriendo una ventana de comandos y escribiendo el comando: composer -version, lo que mostrará un resultado como el siguiente:



```
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\php5619vc11x86\160602110537\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');"

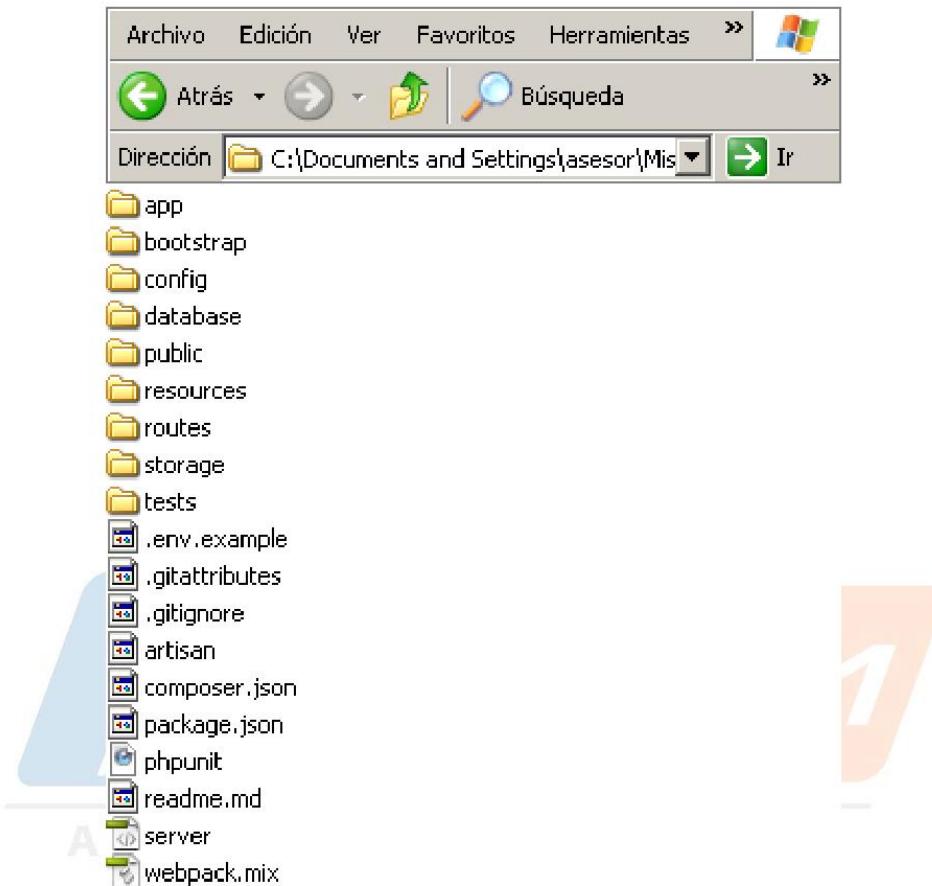
Loading composer repositories with package information
Updating dependencies (including require-dev)
-
```

### 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:



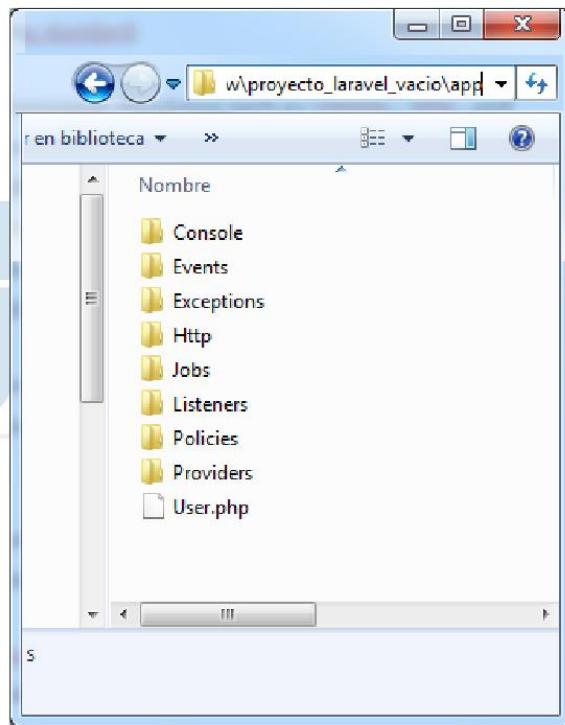
Los directorios que se encuentran en la raíz del proyecto son:

- app: contiene el código de la aplicación.
- bootstrap: contiene los archivos que bootstrap utiliza.
- config: archivos de configuración de la aplicación.
- database: contiene los archivos de migración de la base de datos y los seeds.
- public: contiene los controladores y los recursos (images, JavaScript, CSS, etc.).
- resources: contiene las vistas.
- storage: contiene las plantillas Blade, caches y otros archivos generados por el framework. Esta carpeta está dividida en app

(almacena cualquier archivo que necesite la aplicación), framework (para los archivos caché) y log (los archivos log).

- tests: almacena los archivos para hacer pruebas unitarias con PHPUnit.
- vendor: contiene las dependencias de 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. EL PATRÓN MVC

### 2.1.- Concepto

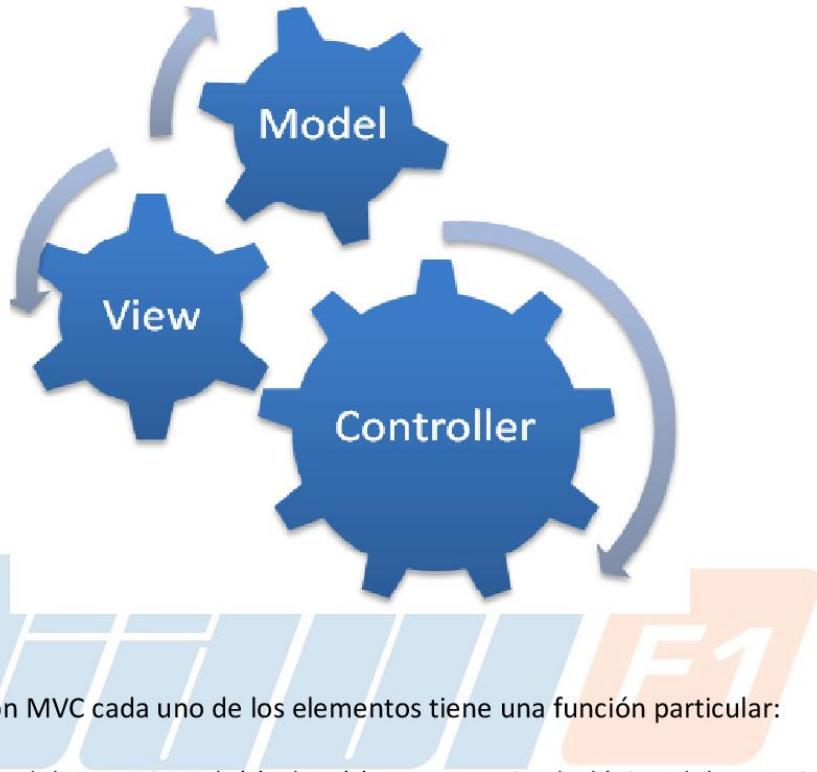
Durante el desarrollo de software es posible encontrarse con situaciones en las que el programador nota que está resolviendo un problema similar a otro(s) que ya ha resuelto, aunque el problema actual tenga ciertas particularidades que no tenían los anteriores. Lo ideal sería resolver el nuevo problema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

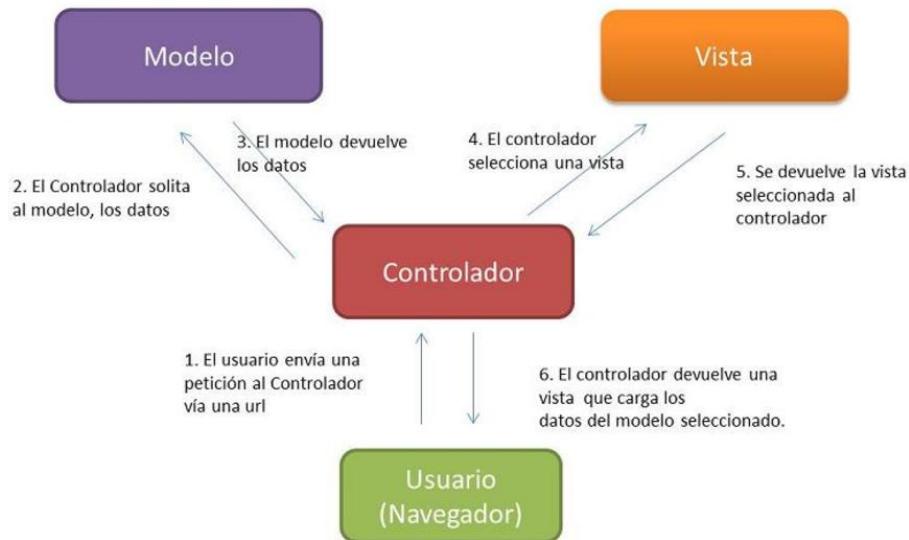


En el patrón MVC cada uno de los elementos tiene una función particular:

- 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.

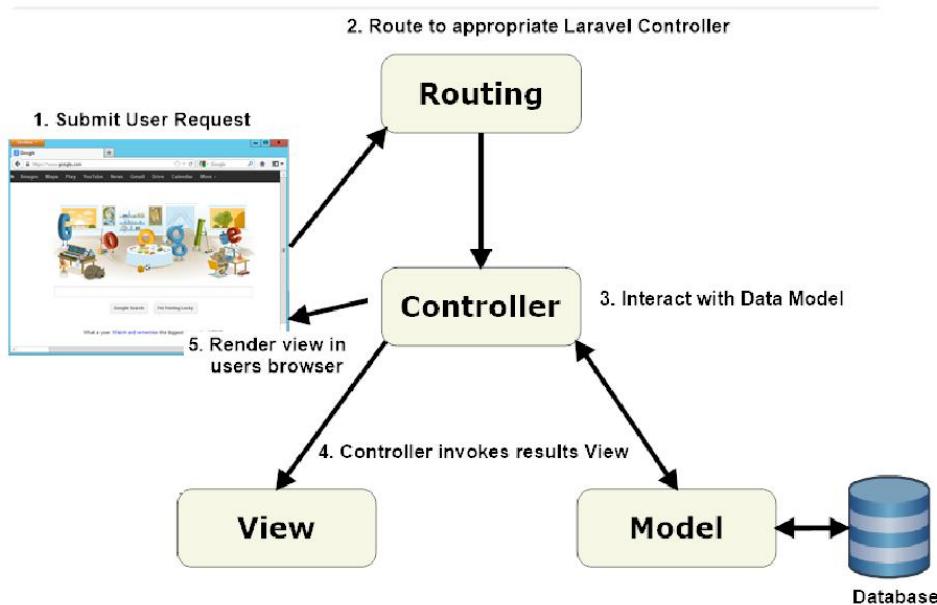
## 2.2.- El Patrón Mvc en Aplicaciones Web

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



### 2.3.- El Patrón Mvc y Laravel

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



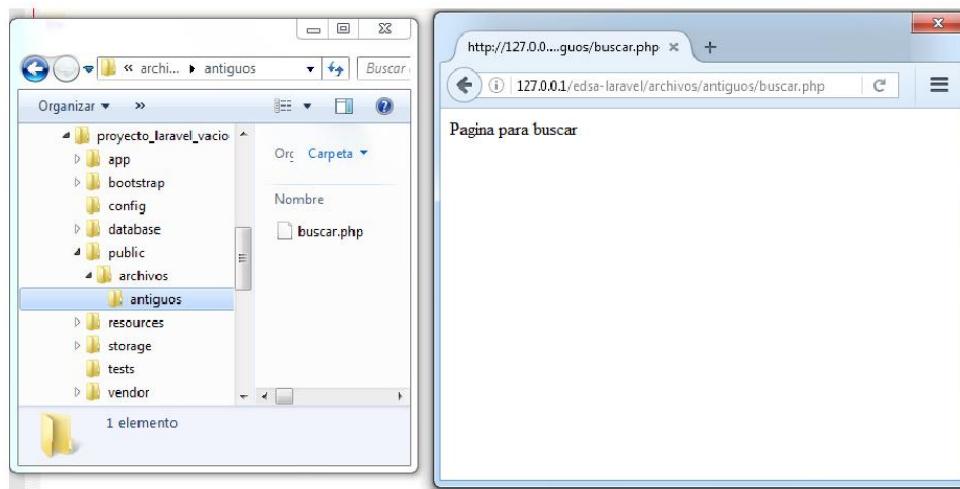
En Laravel específicamente, el funcionamiento del patrón y la dinámica de ejecución de la aplicación es muy diferente a la forma de trabajar aplicaciones php sin Framework. En forma general, la dinámica es la siguiente:

- 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 3. ENRUTAMIENTO

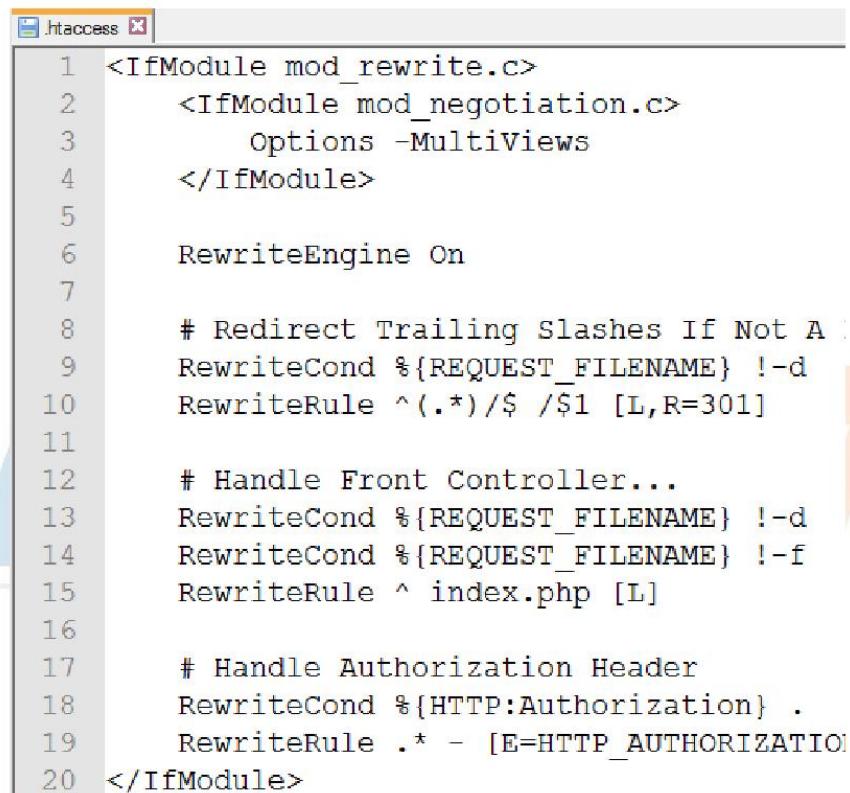
### 3.1.- Las Rutas

En las aplicaciones web tradicionales, la URL con la cual se accede a los recursos tiene que ver con la estructura de carpetas física 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.

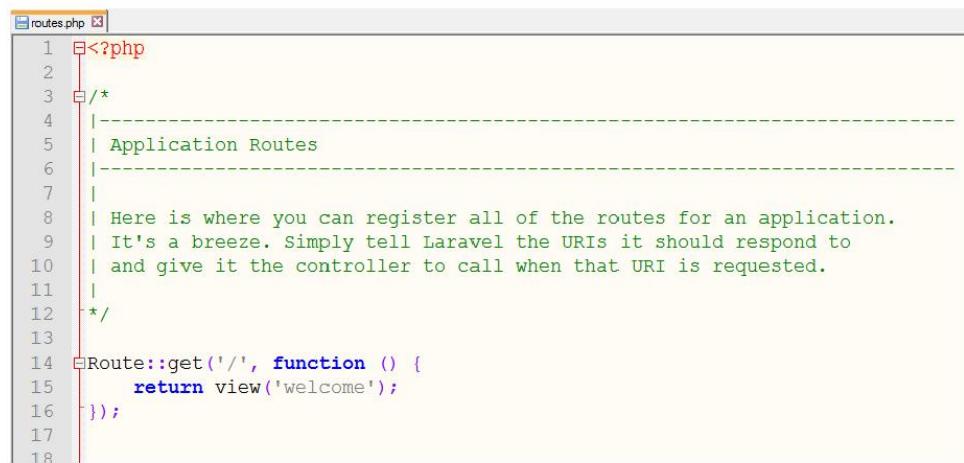
La aplicación del "Front Controller" se logra gracias al uso de archivos .htaccess de apache. En la carpeta "public" se encuentra este archivo y es quien le dice a apache que todas las solicitudes recibidas deben ser manejadas por el archivo index.php:



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

### 3.2.- El Archivo Route.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:



```

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 ()
{
    codigo de respuesta al llamado de la ruta
})

```

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

### 3.3.- 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:

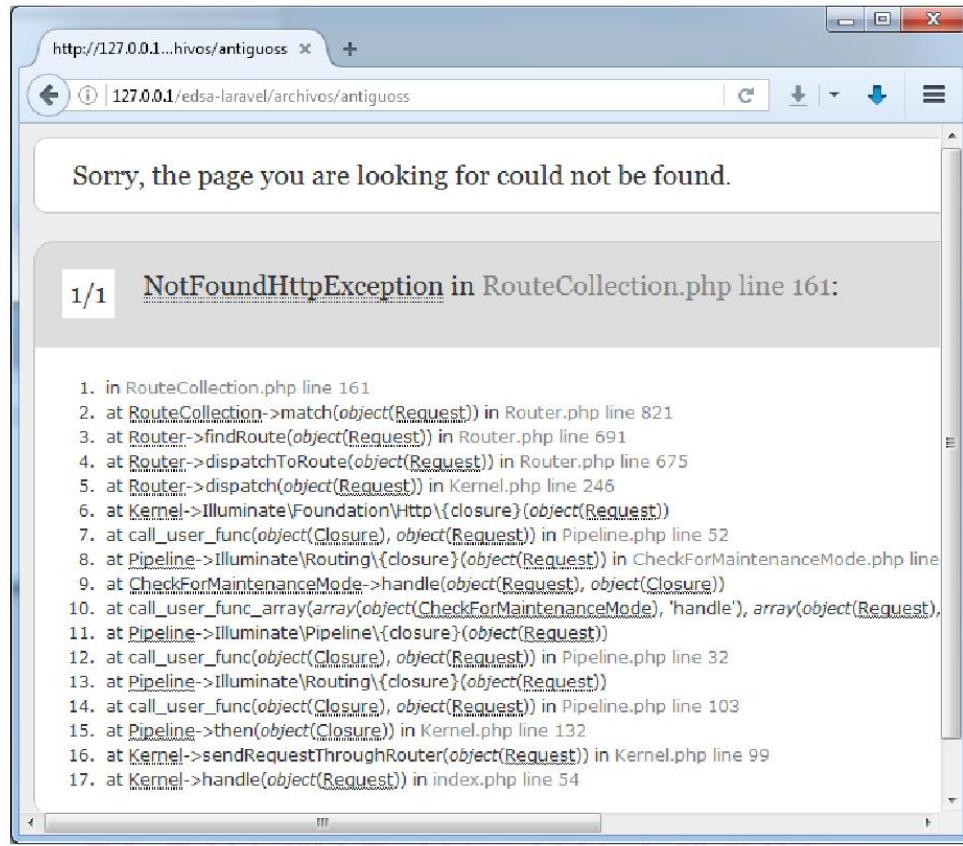
The screenshot shows a code editor window titled "routes.php" containing the following PHP code:

```
1 <?php
2
3 Route::get('/', function () {
4     return view('welcome');
5 });
6
7 Route::get('/archivos/antiguos', function () {
8     return "Pagina para buscar";
9 });
```

Below the code editor is a web browser window with the URL "http://127.0.0.1...chivos/antiguos". The browser displays the text "Pagina para buscar".

Es posible que al hacer una solicitud de una URL se necesiten enviar datos (usando el método get de HTML)

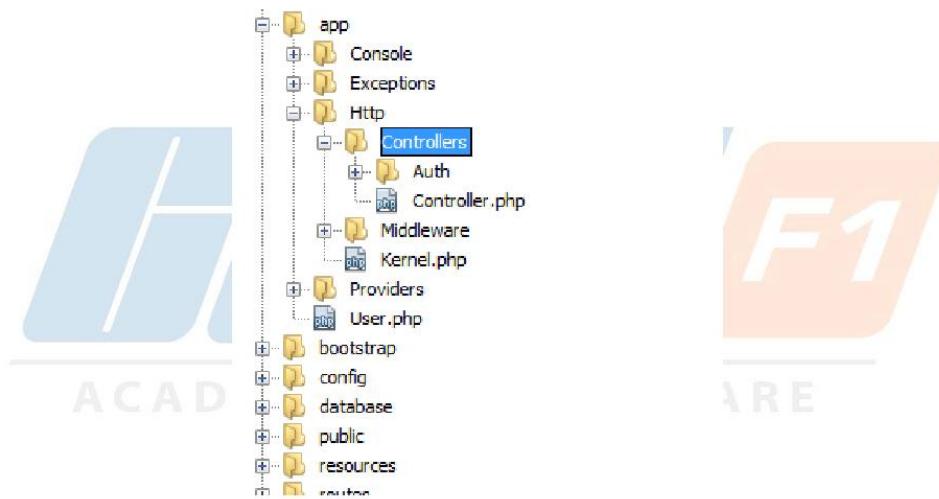
Si se hace referencia a una ruta que no existe, se mostrará un error de ejecución (una excepción) que indica que no se encuentra el manejador correspondiente a la ruta escrita en la URL. Esto sería un equivalente a un error 404 (página no encontrada). En el ejemplo se está haciendo la petición de la ruta "/archivos/antiguoss" que no está definida en el route.php:



## Capítulo 4. CONTROLADORES

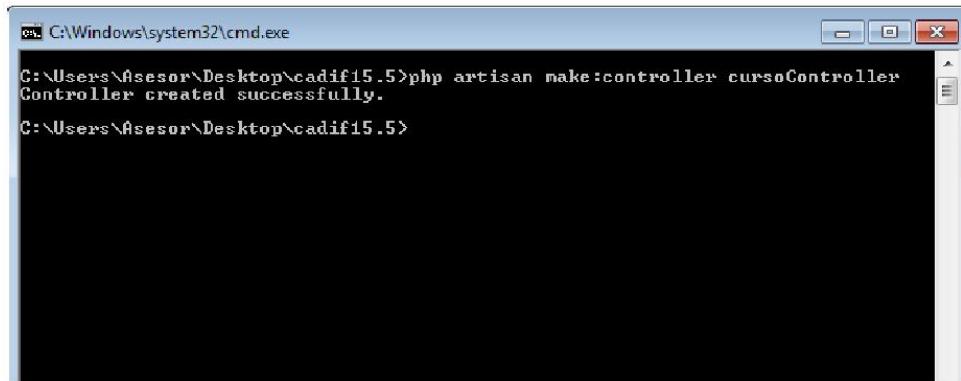
### 4.1.- Crear el Controlador

En lugar de definir toda su lógica de gestión de solicitudes como cierres en archivos de ruta, 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.



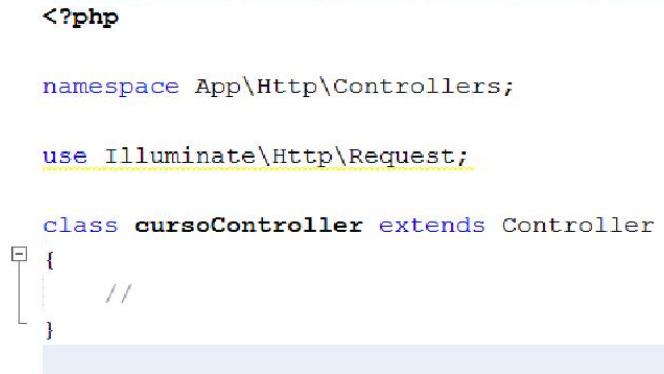
Para crear un controlador se debe usar la interfaz de línea de comandos (CLI por sus siglas en inglés de Command-line interface), la cual es un medio para la interacción con la aplicación donde los usuarios (en este caso los desarrolladores) dan instrucciones en forma de línea de texto simple o línea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

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



```
C:\Windows\system32\cmd.exe
C:\Users\Asesor\Desktop\cadif15.5>php artisan make:controller cursoController
Controller created successfully.
C:\Users\Asesor\Desktop\cadif15.5>
```

Los controladores que se generen en la aplicación se van almacenando en el directorio `app\Http\Controllers`, los cuales de manera predeterminada contendrán su definición como se muestra en la imagen



```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class cursoController extends Controller
{
    //
}
```

#### 4.2.- Enrutar un Método Del Controlador

En Laravel son los controladores los que van a responder a las peticiones de rutas hechas por el usuario o por la propia aplicación, de esta manera mantendremos la lógica de programación dentro de los controladores y no directamente en la definición de las rutas en el archivo web.php. Es por esto que debemos crear métodos que sean llamados en las definiciones de las rutas de la siguiente manera:

```

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

Route::get('index', 'cursoController@index');



---


<?php
Controlador
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class cursoController extends Controller
{
    function index(){
        return "Pagina de inicio";
    }
}

```

#### 4.3.- Enrutar un Controlador Completo

Un controller usualmente trabaja con las peticiones:

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

PATCH.

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.

Los controladores nos ayudan a agrupar estas peticiones en una clase que se liga a las rutas, en el archivo routes/routes.php, para esto usamos un tipo de ruta llamada resource:

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

Es hora de crear un controlador con métodos específicos que respondan a cada una de las peticiones habladas anteriormente, 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

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class cursoController extends Controller
{
    public function index() {
        //
    }
    public function create() {
        //
    }
    public function store(Request $request) {
        //
    }
    public function edit($id)
    {
        //
    }
    public function update(Request $request, $id)
    {
        //
    }
    public function destroy($id)
    {
        //
    }
}
```



## ACADEMIA DE SOFTWARE

Podemos 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

```
C:\Users\Asesor\Desktop\cadif154>php artisan route:list
+-----+-----+-----+-----+
| Domain | Method | URI          | Name      | Action
+-----+-----+-----+-----+
|        | GET    | /             | Closure   |
| web    | GET    | api/user     | Closure   |
|        | POST   | curso        | curso.store | App\Http\Controllers\cursoController@
store   | web    |               |           |
|        | GET    | curso        | curso.index | App\Http\Controllers\cursoController@
index   | web    |               |           |
|        | GET    | curso/create | curso.create | App\Http\Controllers\cursoController@
create  | web    |               |           |
|        | DELETE | curso/{curso} | curso.destroy | App\Http\Controllers\cursoController@
destroy | web    |               |           |
|        | PUT/PATCH | curso/{curso} | curso.update | App\Http\Controllers\cursoController@
update  | web    |               |           |
|        | GET    | curso/{curso} | curso.show  | App\Http\Controllers\cursoController@
show    | web    |               |           |
|        | GET    | curso/{curso}/edit | curso.edit | App\Http\Controllers\cursoController@
edit    | web    |               |           |
+-----+-----+-----+-----+
```



## Capítulo 5. VISTAS. PARTE 1

### 5.1.- 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:

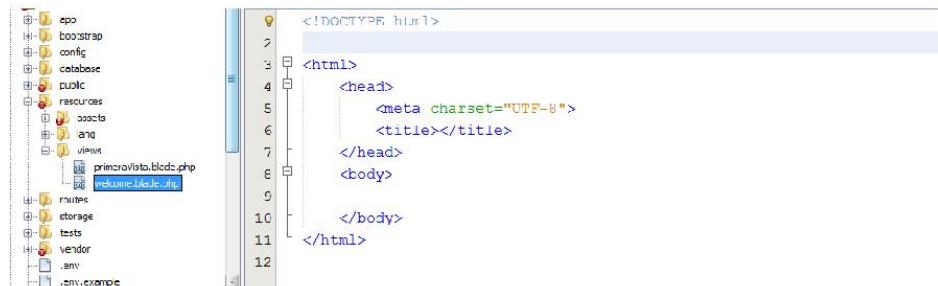
```
<html>
  <body>
    <h1>Hola Mundo</h1>
  </body>
</html>
```

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 un motor de plantillas de Laravel que se explicará más adelante



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

## 5.2.- Vista- Controlador

Las vistas han de ser llamadas dependiendo de la ruta a la que se está 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 metodo "index" del controlador cursoController gracias a la palabra 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 método "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:



## Página de inicio

### 5.3.- 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>
```



The screenshot shows a browser window with the URL 127.0.0.1:8000/curso. The page content is: El nombre del curso es:Laravel

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]);
}

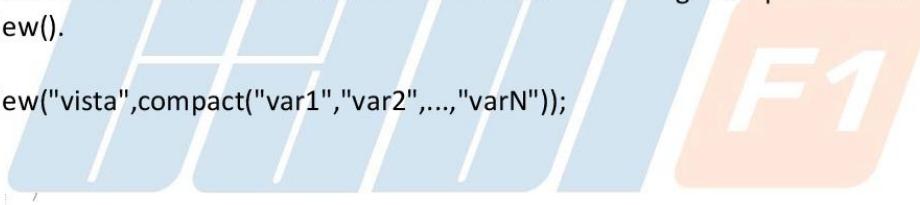
***
```

---

<body>	<b>Vista</b>
<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"));
}
```

Gracias a Blade, podemos usar directivas que nos permitan usar instrucciones PHP de una manera muy sencilla. 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 6. CONFIGURACIÓN DE LA BD

### 6.1.- Archivo de Configuración

Laravel nos 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. Recuerda que los motores de datos que soporta Laravel son:

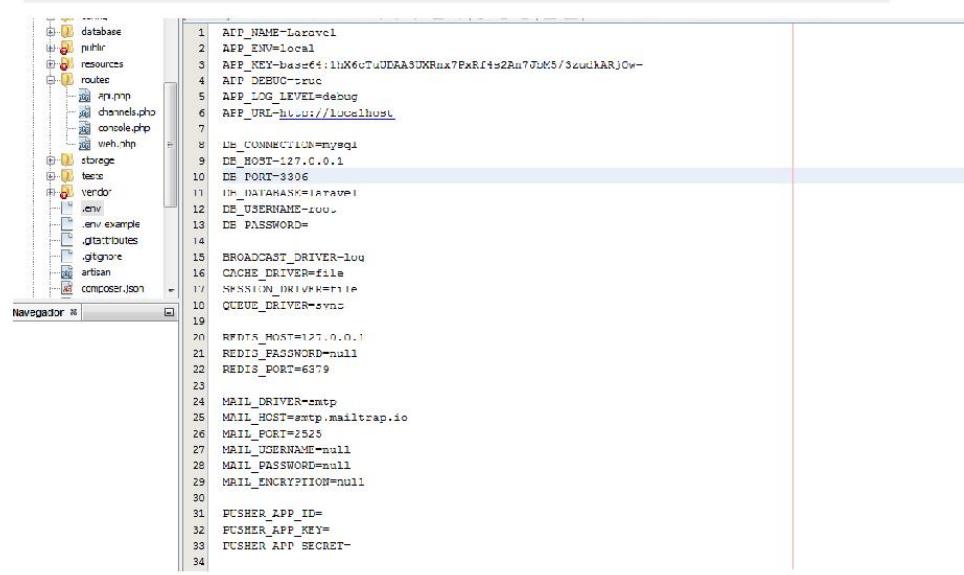
MySQL

Postgres

SQLite

SQL Server

Para establecer la conexión a una base de datos, Laravel contiene un archivo de configuración donde debemos colocar la información respectiva de la configuración de nuestra base de datos. El archivo .env ubicado en la raíz de la aplicación es el archivo que debemos editar. Demos un vistazo a este archivo:



```

database
  migrations
    channels.php
    console.php
    routes.php
    web.php
storage
tests
vendor
.env
.env.example
.gitattributes
.gitignore
artisan
composer.json
Navegador
1 APP_NAME=Laravel
2 APP_ENV=local
3 APP_KEY=bases64:1iX6cTuJDAASUXRnx?PxRf4s2Rn7JnK5/3zuuhARjOw-
4 APP_DEBUG=true
5 APP_LOG_LEVEL=debug
6 APP_URL=http://lucuelhusu
7
8 DB_CONNECTION=mysql
9 DB_HOST=127.0.0.1
10 DB_PORT=3306
11 DB_DATABASE=laravel
12 DB_USERNAME=root
13 DB_PASSWORD=
14
15 BROADCAST_DRIVER=log
16 CACHE_DRIVER=file
17 SESSION_DRIVER=file
18 QUEUE_DRIVER=sqs
19
20 RNDIS_HOST=127.0.0.1
21 REDIS_PASSWORD=null
22 REDIS_PORT=6379
23
24 MAIL_DRIVER=smtp
25 MAIL_HOST=smtp.mailtrap.io
26 MAIL_PORT=2525
27 MAIL_USERNAME=null
28 MAIL_PASSWORD=null
29 MAIL_ENCRYPTION=null
30
31 PUSHER_APP_ID=
32 PUSHER_APP_KEY=
33 PUSHER_APP_SECRET=
34

```

Los parámetros del archivo .env que debemos manipular para poder establecer una conexión con nuestra 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 la edición de este archivo se muestra en la imagen:

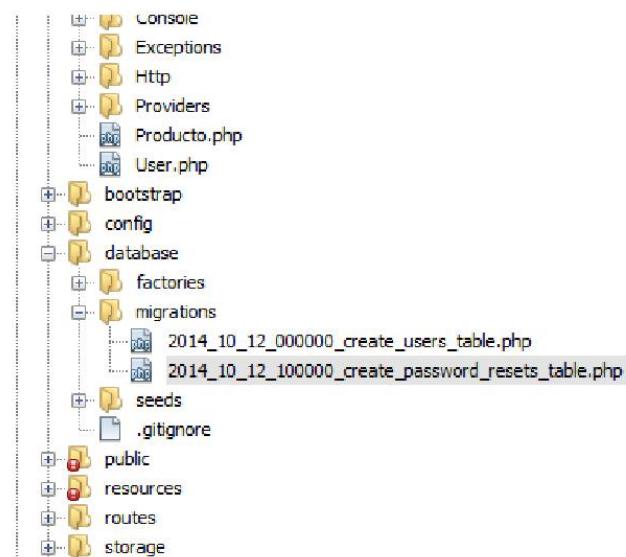
```
DB_CONNECTION=mysql  
DB_HOST=127.0.0.1  
DB_PORT=3306  
DB_DATABASE=laravel  
DB_USERNAME=root  
DB_PASSWORD=
```

## 6.2.- Migraciones

Las migraciones son como el control de versiones para su 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 su aplicación.

Las migraciones se encuentran en el directorio App/database/migrations.

Laravel trae predefinidas 2 migraciones que se muestran en la imagen



Un archivo de migración es una clase que contiene la definición de la tabla que se desea crear en la Base de datos. Está compuesta por el método UP y el método DOWN.

El método UP es el encargado de correr la migración. Se ejecuta al momento de hacer el migrate.

El método Down se encarga de deshacer la migración en cualquier momento.

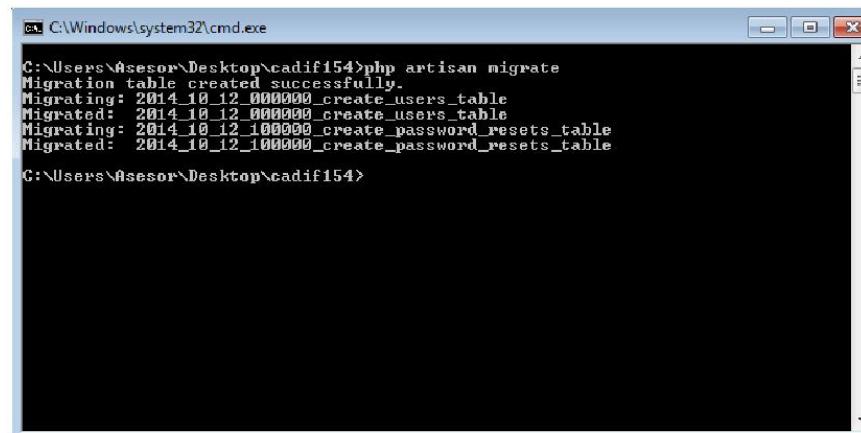
```
use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

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');
    }
}
```

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

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

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



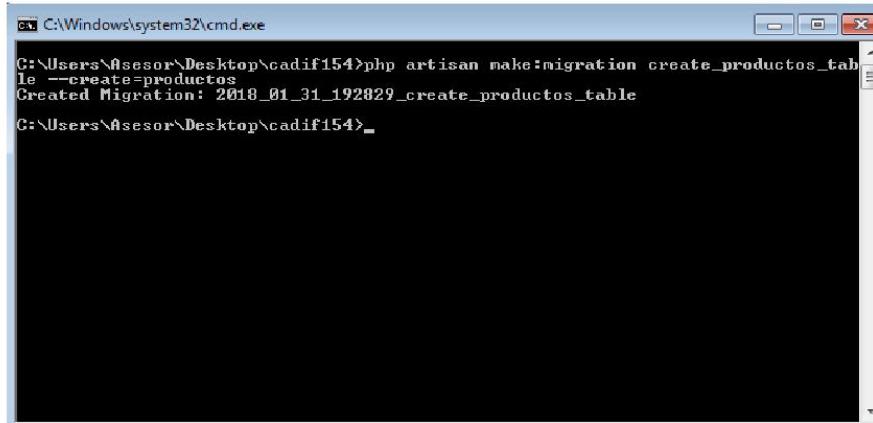
```
C:\Users\Asesor\Desktop\cadif154>php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table
C:\Users\Asesor\Desktop\cadif154>
```

El nombre de las migraciones generalmente son de la siguiente manera:

create\_nombretabla\_table

Para crear una nueva migración con el objetivo de ir creando nuevas tablas en nuestra base de datos, utilizamos el comando `make:migration` como sigue:

php artisan make:migration create\_nombretabla\_table --create=nombretabla

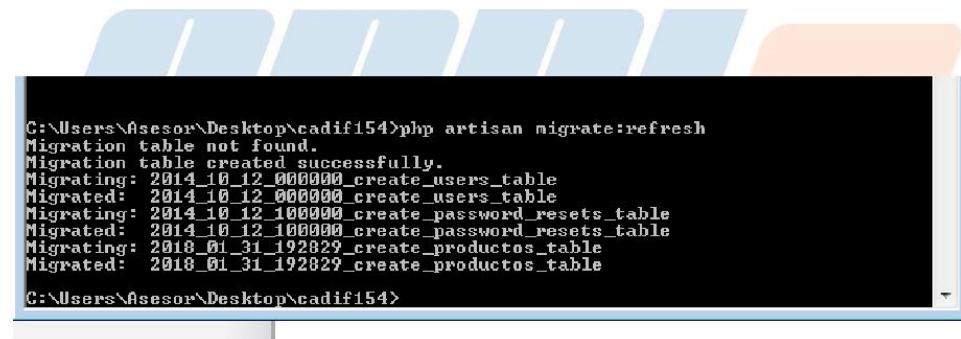


```
C:\Windows\system32\cmd.exe
C:\Users\Asesor\Desktop\cadif154>php artisan make:migration create_productos_table --create=productos
Created Migration: 2018_01_31_192829_create_productos_table
C:\Users\Asesor\Desktop\cadif154>
```

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

- MIGRATE:REFRESH. 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, debemos ejecutar el comando `php artisan migrate:refresh` para actualizar nuestra base de datos



```
C:\Users\Asesor\Desktop\cadif154>php artisan migrate:refresh
Migration table not found.
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: 2018_01_31_192829_create_productos_table
Migrated: 2018_01_31_192829_create_productos_table
C:\Users\Asesor\Desktop\cadif154>
```

## Capítulo 7. QUERY BUILDER

### 7.1.- Definicion

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.

### 7.2.- 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:

The screenshot shows a code editor with two sections. The top section, labeled 'Controlador' (Controller), contains PHP code for a 'cursoController'. It includes imports for App\Http\Controllers, Illuminate\Support\Facades\DB, and Illuminate\Http\Request. The 'index()' method retrieves all products from the database and returns a view named 'vistaPrincipal' with the products compacted. The bottom section, labeled 'Vista' (View), contains Blade template code for a list of products. It starts with <body>, followed by <h2>Listado de productos</h2>, and then an <ul> tag with a @foreach loop that prints the product names.

```
<?php  
namespace App\Http\Controllers;  
use Illuminate\Support\Facades\DB;  
use Illuminate\Http\Request;  
  
class cursoController extends Controller  
{  
    /** Display a listing of the resource. */  
    public function index()  
    {  
        $productos=DB::table('productos')->get();  
        return view('vistaPrincipal',compact('productos'));  
    }  
}  
  
<body>  
    <h2>Listado de productos</h2>  
    <ul>  
        @foreach($productos as $prod)  
            <li>{{$prod->nombre}}</li>  
        @endforeach  
    </ul>
```

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

Nuestra vista está mostrando entonces el listado de productos como se muestra en la imagen:



## Productos

- Laptop
- smarthphone

### 7.3.- 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/5.4/queries>). Por ejemplo, si se quiere obtener de la tabla USERS el registro cuya 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:

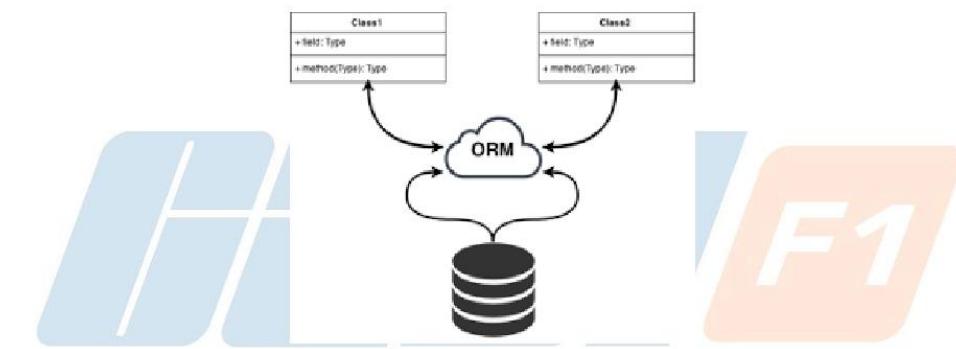
```
$user = DB::table('users')->where('name', 'John')->first();
```

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

## Capítulo 8. ELOQUENT

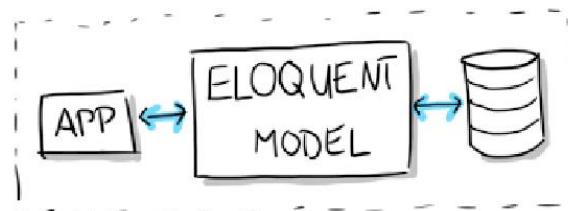
### 8.1.- Definicion

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 datos entre el sistema de tipos utilizado en un lenguaje de programación orientado a objetos y la utilización de una base de datos relacional como motor de persistencia.



### ACADEMIA DE SOFTWARE

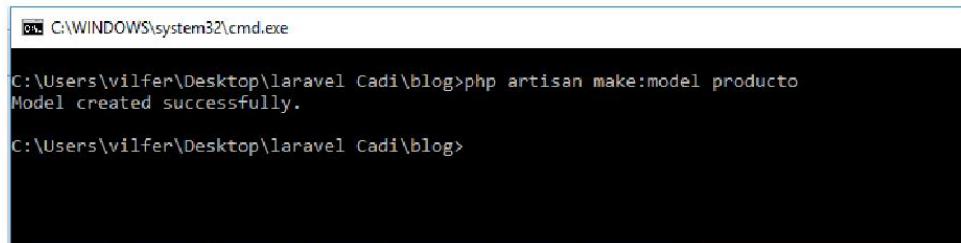
El ORM Eloquent incluido con Laravel proporciona una implementación de ActiveRecord bella y sencilla para trabajar con su base de datos. Cada tabla de base de datos tiene un "Modelo" correspondiente que se utiliza para interactuar con esa tabla. Los modelos le permiten consultar datos en sus tablas,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



## 8.2.- Modelos

Para comenzar, creamos 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 amplían 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:

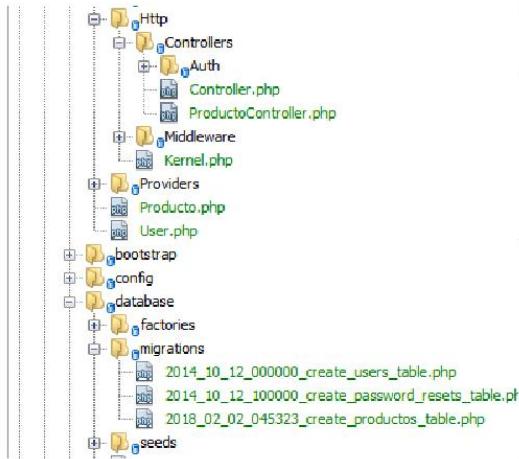


```
C:\WINDOWS\system32\cmd.exe
C:\Users\vilfer\Desktop\laravel Cadi\blog>php artisan make:model producto
Model created successfully.
C:\Users\vilfer\Desktop\laravel Cadi\blog>
```

Los modelos van a permitir el mapeo de la base de datos gracias 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, podemos correr el siguiente comando:

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

```
C:\Users\wilfer\Desktop\laravel Cadi\blog>php artisan make:model Producto -mc --resource
Model created successfully.
Created Migration: 2018_02_02_045323_create_productos_table
Controller created successfully.
```



Debido a que Eloquent está haciendo un mapeo de nuestra 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:

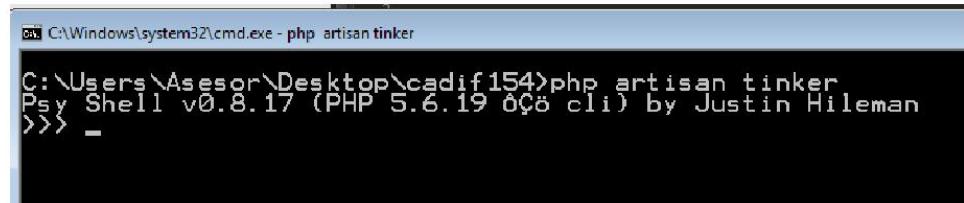
- `save()`: Permite guardar un objeto dentro de una tabla.
- `all()`: Permite recuperar todos los registros de la tabla
- `find()`: Permite un registr mediante su identificador único.
- `pluck()`: Solicitar valores de alguna columna en específico

### 8.3.- 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:

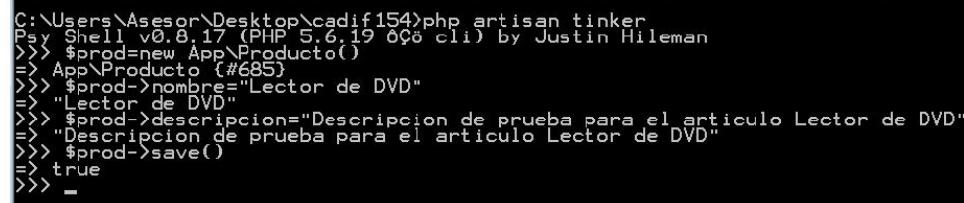
```
php artisan tinker
```

Usaremos esta consola para crear nuevos registros en nuestras tablas asociadas a los modelos, aunque no será esta la manera habitual de hacerlo.



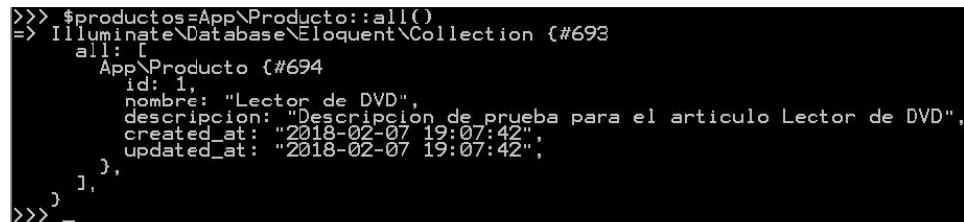
```
C:\Windows\system32\cmd.exe - php artisan tinker
C:\Users\Asesor\Desktop\cadif154>php artisan tinker
Psy Shell v0.8.17 (PHP 5.6.19 cli) by Justin Hileman
>>> _
```

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 "Productos":



```
C:\Users\Asesor\Desktop\cadif154>php artisan tinker
Psy Shell v0.8.17 (PHP 5.6.19 cli) by Justin Hileman
>>> $prod=new App\Producto()
=> App\Producto {#685}
>>> $prod->nombre="Lector de DVD"
=> "Lector de DVD"
>>> $prod->descripcion="Descripcion de prueba para el articulo Lector de DVD"
=> "Descripcion de prueba para el articulo Lector de DVD"
>>> $prod->save()
=> true
>>> _
```

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



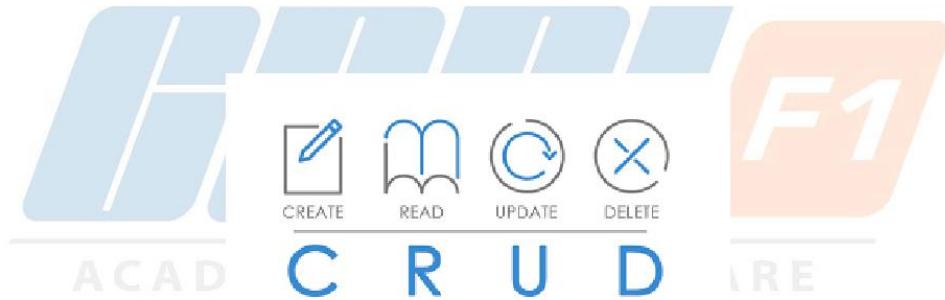
```
>>> $productos=App\Producto::all()
=> Illuminate\Database\Eloquent\Collection {#693
    all: [
        App\Producto {#694
            id: 1,
            nombre: "Lector de DVD",
            descripcion: "Descripcion de prueba para el articulo Lector de DVD",
            created_at: "2018-02-07 19:07:42",
            updated_at: "2018-02-07 19:07:42",
        },
    ],
}>>> _
```

## Capítulo 9. AGREGAR REGISTROS

### 9.1.- Introducción

En informática, CRUD es el acrónimo de "Crear, Leer, Actualizar y Borrar" (del original en inglés: Create, Read, Update and Delete), que se usa 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.



### 9.2.- 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('productos.create');
    }
```

Foto: Unsplash

### 9.3.- 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 para colocar este token de seguridad dentro del formulario.

`{{ csrf_field() }}`

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



```
<h1>Nuevo producto</h1>

<form action="{{ route('productos.store') }}" method="post">

    {{csrf_field()}}
    <label>Nombre</label><br/>
    <input type="text" name="nombre"><br/>
    <label>Descripcion</label><br/>
    <textarea name="descripcion" row="100" col="150"></textarea><br/>
    <input type="submit" name="enviar" value="Guardar"></input>

</form>
```

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

```
public function store(Request $request)
{
    $pro=new Producto();
    $pro->nombre=$request->nombre;
    $pro->descripcion=$request->descripcion;
    $pro->save();
    return redirect()->route('productos.index');
}
```



## Capítulo 10. LEER REGISTROS

### 10.1.- Ruta Show

Para recuperar un registro en específico y mostrar algún detalle, debemos tener un enlace a 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, recordemos que la ruta es la siguiente:

productos/{id}

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

```
<h1>Listado de productos</h1>
<table border=1>
    @foreach($productos as $pro)
        <tr>
            <td>{{$pro->nombre}}</td>
            <td>{{$pro->descripcion}}</td>
            <td><a href="{{route('productos.show',$pro->id)}}>Ver detalle</a></td>
            <td><a href="{{route('productos.edit',$pro->id)}}>Editar</a></td>
            <td><form action="{{route('productos.destroy',$pro->id)}}" method="post">
                {{csrf_field()}}
                <input type="submit" name="eliminar" value="Eliminar">
                <input type="hidden" name="_method" value="DELETE"></input>
            </form>
            </td>
        </tr>
    @endforeach
</table>
```

## 10.2.- 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 el método de una instancia de modelo en los cierres de ruta o las acciones del controlador.

```
    public function show(Producto $producto)
    {
        return view("productos.show",compact('producto'));
    }
```

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

```
<!DOCTYPE html>
<html>
<head>
| |     <title></title>
</head>
<body>
<h1>{{ $producto->nombre }}</h1>
<p>{{ $producto->descripcion }}</p>
<a href="{{ route('productos.index') }}">Listado de productos</a>
</body>
</html>
```

## Capítulo 11. MODIFICAR REGISTROS

### 11.1.- 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 un controlador con sus recursos.

```
<h1>Listado de productos</h1>
<table border=1>
    @foreach($productos as $pro)
        <tr>
            <td>{{$pro->nombre}}</td>
            <td>{{$pro->descripcion}}</td>
            <td><a href="{{route('productos.show',$pro->id)}}>Ver detalle</a></td>
            <td><a href="{{route('productos.edit',$pro->id)}}>Editar</a></td>
            <td><form action="{{route('productos.destroy',$pro->id)}}" method="post">
                {{csrf_field()}}
                <input type="submit" name="eliminar" value="Eliminar">
                <input type="hidden" name="_method" value="DELETE"></input>
            </form>
            </td>
        </tr>
    @endforeach
</table>
```

### 11.2.- 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”. Usamos los siguientes helpers

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

```
<h1>Editar Registro</h1>  
<form action="{{route('productos.update', $producto->id)}}" method="post">  
    {{csrf_field()}}  
    {{method_field('PUT')}}<!--<input type="hidden" name="_method" value="PUT"></input>-->  
    <label>Nombre</label><br/>  
    <input type="text" name="nombre" value="{{ $producto->nombre }}"><br/>  
    <label>Descripción</label><br/>  
    <textarea name="descripcion" row="100" col="150">{{ $producto->descripcion }}</textarea><br/>  
    <input type="submit" name="enviar" value="Guardar"></input>  
</form>
```

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, Producto $producto)  
{  
    $producto->nombre=$request->nombre;  
    $producto->descripcion=$request->descripcion;  
    $producto->save();  
    return redirect()->route('productos.index');  
}
```

## Capítulo 12. ELIMINAR REGISTROS

### 12.1.- Introducción

La letra D del DRUD 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()
```

### 12.2.- Formulario de Eliminado

Para proceder a eliminar registros de nuestras tablas, se 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 csrf\_field() 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"

```
<h1>Listado de productos</h1>
<table border=1>
    @foreach($productos as $pro)
        <tr>
            <td>{{$pro->nombre}}</td>
            <td>{{$pro->descripcion}}</td>
            <td><a href="{{route('productos.show',$pro->id)}}>Ver detalle</a></td>
            <td><a href="{{route('productos.edit',$pro->id)}}>Editar</a></td>
            <td><form action="{{route('productos.destroy',$pro->id)}}" method="post">
                {{csrf_field()}}
                <input type="submit" name="eliminar" value="Eliminar">
                <input type="hidden" name="_method" value="DELETE"></input>
            </form>
            </td>
        </tr>
    @endforeach
</table>
```

### 12.3.- Método Destroy

Para proceder a eliminar registros de nuestras tablas, se 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(Producto $producto)
{
    $producto->delete();
    return redirect()->route('productos.index');
}
```



## Capítulo 13. VALIDACIONES EN EL MODELO

### 13.1.- 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 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 está vacío.

ACADEMIA DE SOFTWARE

```
public function store(Request $request)
{
    $this->validate($request, ['nombre'=>'required']);
    $pro=new Producto();
    $pro->nombre=$request->nombre;
    $pro->descripcion=$request->descripcion;
    $pro->save();
    return redirect()->route('productos.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 el arreglo \$errors en nuestra vista de la siguiente manera:

```
<form action="{{route('productos.store')}}" method="post">
    {{csrf_field()}}
    <label>Nombre</label><br/>
    <input type="text" name="nombre"><br/>
    <label>Descripcion</label><br/>
    <textarea name="descripcion" row="100" col="150"></textarea><br/>
    <input type="submit" name="enviar" value="Guardar"></input>
</form>
<ul>
    @foreach($errors->all() as $error)
        <li>{{$error}}</li>
    @endforeach
</ul>
```

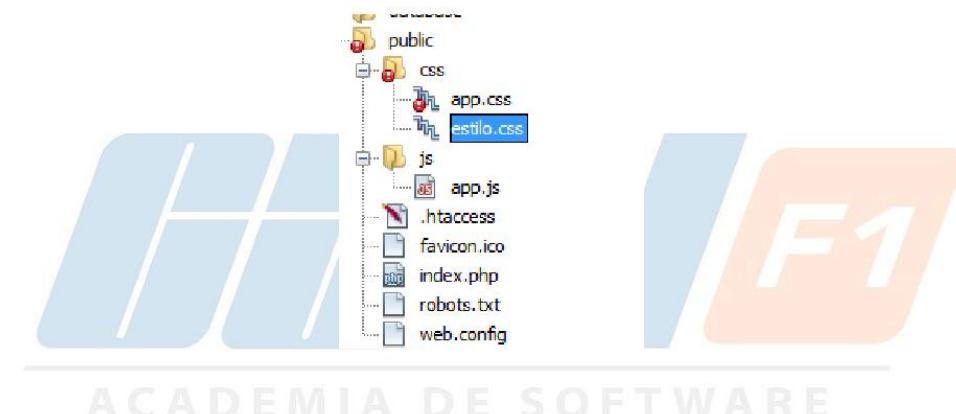


## Capítulo 14. VISTAS. PARTE 2

### 14.1.- Agregando Hojas de Estilo

En Laravel, las hojas de estilo y archivos de Javascript tienen lugar en el directorio public.

```
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')}}">
```

### 14.2.- 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, examinaremos 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:

@section() : Define una sección de contenido

@yield() : Mostrar el contenido de una sección

```
<!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.

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