

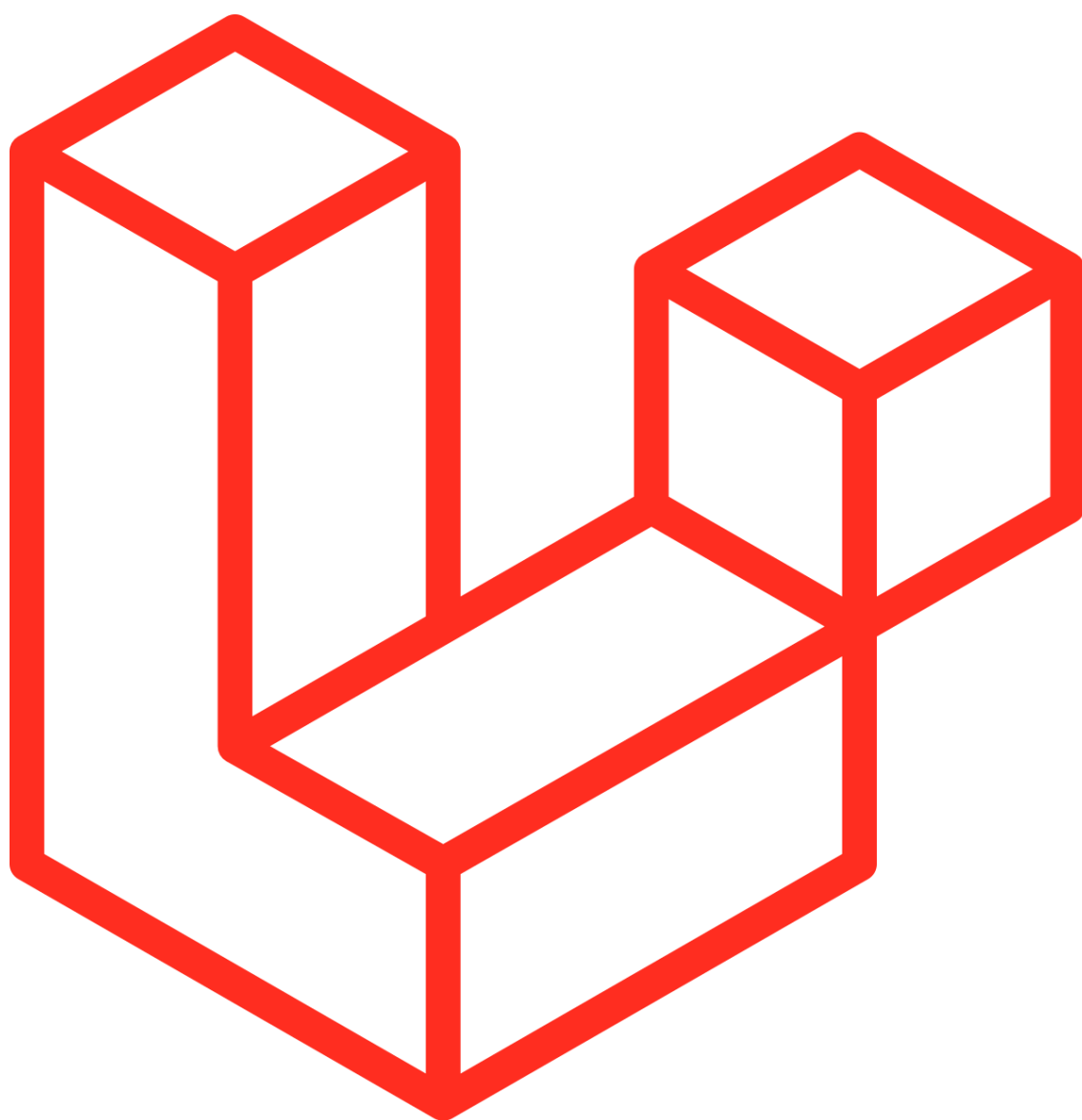
---

# LARAVEL

---

Miguel A. M. Rasteu

2021



## Contenido

---

1 – Introducción a Laravel.....	4
1.1 – Instalar Laravel .....	4
Crear proyecto .....	4
Crear un host virtual .....	4
Estructura de un proyecto de Laravel.....	6
2 – Routing.....	7
2.1 – Rutas básicas .....	7
2.2 – Parámetros en las rutas (routes).....	9
2.3 – Consola Artisan.....	10
3 – Plantillas y Vistas .....	11
3.1 – Vistas en Laravel .....	11
3.2 – Sistemas de plantillas Blade .....	11
Imprimir por pantalla.....	11
Condicionales.....	12
Bucles .....	12
Includes dentro de las views.....	13
3.3 – Plantillas base o layout.....	13
4 – Controladores.....	15
4.1 – Controladores básicos .....	15
***IMPORTANTE*** .....	16
4.2 – Controladores Resource .....	17
4.3 – Enlaces en Laravel .....	17
4.4 – Redirecciones.....	18
4.5 – Middleware.....	19
5 – Formularios .....	21
6 – Bases de datos.....	23
6.1 – Conectarse a una base de datos con Laravel .....	23
6.2 – Migraciones .....	23
*** IMPORTANTE *** .....	24

6.3 – Migraciones con SQL .....	26
6.4 – Seeders .....	26
6.5 – Listar datos .....	27
6.6 – Mostrar una fila .....	29
6.7 – Order by.....	30
6.8 – Insertar Registros.....	30
6.9 – Borrar Registros.....	32
6.10 – Actualizar Registros .....	33
Documentación.....	36

# 1 – Introducción a Laravel

---

## 1.1 – Instalar Laravel

En primer lugar nos dirigimos a la documentación de Laravel (ver [Documentación](#)) para comprobar si tenemos todos los requerimientos necesarios para la instalación de Laravel.

Podemos comprobar la versión de PHP que tenemos usando el siguiente comando en la consola de comandos:

```
php -v
```

Ahora deberemos instalar composer, que es el programa encargado de manejar las dependencias de Laravel, ya que Laravel es solo el framework.

También deberemos comprobar si están activadas todas las extensiones de PHP que se nos indica en la documentación de Laravel.

### Crear proyecto

Para crear un proyecto de Laravel con composer, deberemos dirigirnos a la carpeta donde se creará dicho proyecto, posteriormente introduciremos el siguiente comando en la consola.

```
$ composer create-project laravel/laravel nombre-proyecto
```

Composer se encargará de descargar las dependencias y de crear todas las carpetas del proyecto Laravel.

### Crear un host virtual

Un host virtual es una simulación de una url de un dominio real. De esta forma podemos trabajar de una forma mucho más orgánica y simple.

A continuación veremos cómo crear un host virtual usando wampp

**Paso 1:** Entrar al fichero **C:\wamp\bin\apache\apache2.4.9\conf\httpd.conf** y añadir o descomentar el include del fichero de los hosts virtuales:

```
# virtual hosts  
include conf/extra/httpd-vhosts.conf
```

**Paso 2:** Entrar al fichero `C:\wamp64\bin\apache\apache2.4.46\conf\extra\httpd-vhosts.conf` y añadir los virtualhosts.

```
# Virtual Hosts
#
<VirtualHost *:80>
    ServerName localhost
    ServerAlias localhost
    DocumentRoot "${INSTALL_DIR}/www"
    <Directory "${INSTALL_DIR}/www/">
        Options +Indexes +Includes +FollowSymLinks +MultiViews
        AllowOverride All
        Require local
    </Directory>
</VirtualHost>

# VHOST CURSO LARAVEL
<VirtualHost *:80>
    DocumentRoot "${INSTALL_DIR}/www/master-php/aprendiendo-
laravel/public"
    ServerName aprendiendo-laravel.com.devel
    ServerAlias www.aprendiendo-laravel.com.devel
    <Directory "${INSTALL_DIR}/www/master-php/aprendiendo-laravel/public">
        Options Indexes FollowSymLinks
        AllowOverride All
        Order Deny,Allow
        Allow from all
    </Directory>
</VirtualHost>
```

**Paso 3:** Añadir al fichero hosts de nuestro sistema, en el caso de Windows

`C:\Windows\System32\drivers\etc\hosts` (si estas en Windows 8 o 10 ejecuta el programa de edición de código como Administrador para poder guardar los cambios), y añadir las IP y las url.

```
127.0.0.1 localhost::1 localhost
127.0.0.1 localhost
127.0.0.1 aprendiendo-laravel.com.devel
```

Con estos tres pasos tendríamos creado y configurado nuestro propio host virtual.

## Estructura de un proyecto de Laravel

El directorio **app** es donde está el contenido principal de la aplicación y contiene el código principal tanto de Laravel como de la aplicación que nosotros vamos a construir. Dentro de este directorio tenemos el directorio **Http** donde tenemos los **Controllers** que iremos creando nosotros y los **Models**, entre otros recursos.

También tenemos el directorio **bootstrap** que contiene un archivo que inicia el framework y una caché.

El directorio **config** contiene los archivos de configuración de la aplicación.

El directorio **database** lo que incluye son las migraciones de la base de datos y los seeds, las migraciones nos permiten versionar nuestros cambios en la base de datos y los seeds nos permite rellenar nuestra base de datos de forma programática.

El directorio **public** es donde se contienen los archivos públicos, es decir, assets, el archivo index.php que es por el cual entra la aplicación, se carga de forma automática y es donde se muestra la página web de forma automática y las **views** de nuestra aplicación.

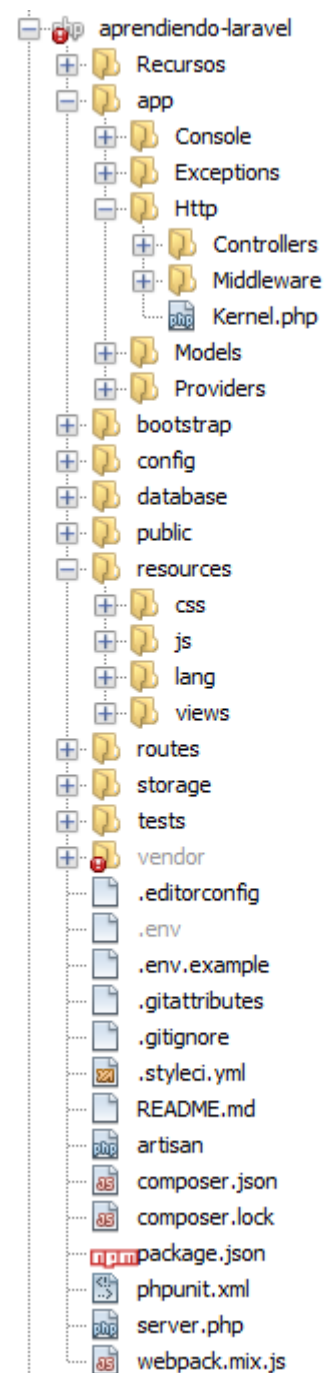
El directorio **routes** contiene todas las rutas de nuestra aplicación que podamos tener.

En el directorio **storage** se usará para almacenar archivos cuando hagamos subidas, etc.

El directorio **test** para hacer test unitarios

EL directorio **vendor** es donde se guardan todos los paquetes, librerías y dependencias que instalamos con composer.

El resto de archivos que se ven son archivos de configuración.



## 2 – Routing

### 2.1 – Rutas básicas

Las rutas dentro de un framework son uno de los pilares más importantes porque es uno de los puntos que más nos facilita a la hora de trabajar con un framework es la posibilidad de configurar nuestras rutas y de configurar nuestras url, pasarles parámetros, tener rutas limpias y amigables, etc.

Si abrimos la carpeta de **routes**, en Laravel, es donde podemos configurar nuestras **routes**. Lo más común es tener nuestras **routes** en el archivo **web.php** pero eso puede variar según la funcionalidad que le demos a la ruta.

La **route** que tenemos por defecto en Laravel lo único que hace es devolvernos una **view**, nos devuelve la **view** Welcome. Pero igual que nos muestra una **view** podemos ponerle un echo e imprimir por pantalla un h1.

Es decir que lo que hace una ruta es ejecutar un bloque de instrucciones.

Podemos añadir todas las rutas con los métodos http que queramos. Los principales métodos http (que veremos en mayor profundidad más adelante) son los siguientes:

GET: Conseguir datos

POST: Guardar datos

PUT: Actualizar datos

DELETE: Borrar datos

En base a esto crearemos nuestras rutas, además Laravel es un framework RESTFUL y permite crear rutas con el método http que queramos y testarlas con un cliente REST.

La sintaxis para definir una ruta es la siguiente:

```
Route::método-Http('/nombre-de-la-ruta', function(){  
    Bloque de código  
    a ejecutar  
});
```

A continuación, podemos ver un ejemplo práctico y su resultado.

```
31 Route::get('/mostrar-fecha', function() {  
32     echo "<h1>Fecha actual</h1>";  
33     echo date('d-m-Y');  
34     echo "<br/>";  
35     echo "<a href='/'>Inicio</a>";  
36 });
```

← → ↺ ↻ No es seguro aprendiendo-laravel.com.devel/mostrar-fecha

**Fecha actual**

26-08-2021

[Inicio](#)

Podemos crearnos una vista en **resources>view** y podemos organizar las vistas en carpeta.

Creamos un archivo que se llame `mostrar-fecha.blade.php` con el mismo contenido que tenemos en los `echo` dentro del bloque de código de la **route** que hemos creado anteriormente.

Es importante que las **views** tengan el `.blade.php` porque las **views** de Laravel se hacen con ese motor de plantillas que veremos en profundidad más adelante.

Una vez creada la **view** deberemos cargar en la **route** la **view** que hemos creado.

En las **routes** solo debemos tener **views** y lógica, nunca debemos imprimir datos en las **routes**.

El resultado lo podemos ver a continuación:

The diagram illustrates the process of creating and displaying a Blade view. It starts with a file explorer showing the `resources/views` directory containing `mostrar-fecha.blade.php` and `welcome.blade.php`. A large blue arrow points to a code editor showing the content of `mostrar-fecha.blade.php`:

```
<?php
2
3     echo "<h1>Fecha actual</h1>";
4     echo date('d-m-Y');
5     echo "<br/>";
6     echo "<a href='/'>Inicio</a>";
```

Another blue arrow points down to a code editor showing the corresponding route in `routes/web.php`:

```
39 Route::get('/mostrar-fecha', function() {
40     return view('mostrar-fecha');
41 });
```

A final blue arrow points left to a browser screenshot of `aprendiendo-laravel.com/devel/mostrar-fecha`, which displays:

**Fecha actual**  
26-08-2021  
[Inicio](#)

Incluso podríamos crearnos una variable en la **route**, por ejemplo `$titulo`, y pasarle esta variable como parámetro a la vista.

Para ello debemos pasarle un segundo parámetro en forma de `array()` y en una índice le pasamos el título con la variable `$titulo`.

Como vemos en la figura.

```
39 Route::get('/mostrar-fecha', function() {
40     $titulo = "Estoy mostrando la fecha ";
41     return view('mostrar-fecha', array(
42         'titulo' => $titulo
43     ));
44 });
```

Y a su vez si en la **view** `mostrar-fecha.blade.php` cambio el contenido del primer `h1` por una variable llamada `$titulo` nos mostrará en dicha **view** el contenido del `array()` con los parámetros que le hemos indicado en la **route**, como vemos en la figura:

The diagram shows the updated view and its output. On the left, a code editor shows the updated content of `mostrar-fecha.blade.php`:

```
<?php
2
3     echo "<h1>$titulo</h1>";
4     echo date('d-m-Y');
5     echo "<br/>";
6     echo "<a href='/'>Inicio</a>";
7
```

On the right, a browser screenshot of `aprendiendo-laravel.com/devel/mostrar-fecha` displays:

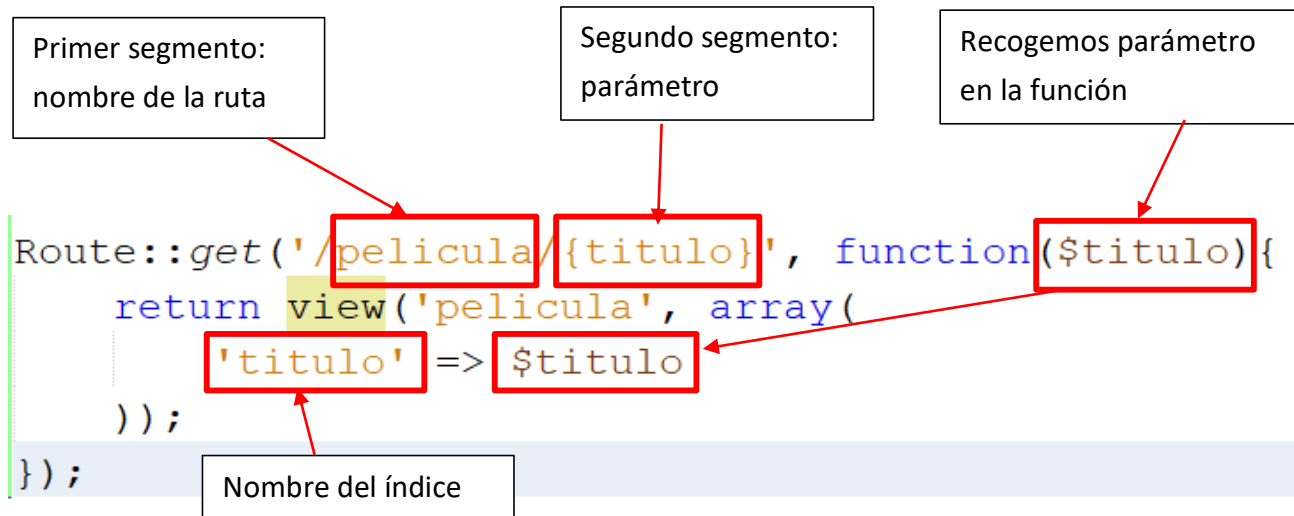
**Estoy mostrando la fecha**  
26-08-2021  
[Inicio](#)



## 2.2 – Parámetros en las rutas (routes)

Los parámetros se indican tras el nombre de la **route** y se encierra entre corchetes “{ }”. Si no indicamos nada en el parámetro es por defecto un parámetro obligatorio.

Posteriormente deberemos recoger ese parámetro en la función asociada a la **route**.



Si queremos que el parámetro sea opcional solo deberemos añadirle un símbolo “?” y en la función a la variable le asignaremos un valor por defecto.

```
Route::get('/pelicula/{titulo?}', function($titulo = "Empty"){
    return view('pelicula', array(
        'titulo' => $titulo
    ));
});
```

Otro punto interesante de las **routes** de Laravel es que me permiten hacer condiciones para permitir o denegar el acceso a la mismas.

Para ello usaremos un `where` con un array indicándolo en el valor del índice con expresiones regulares, como vemos en la siguiente imagen.

```
Route::get('/pelicula/{titulo}', function($titulo = "Empty"){
    return view('pelicula', array(
        'titulo' => $titulo
    ));
})->where(array(
    'titulo' => '[a-zA-z]+'
));
```

La condición dentro del `where` indica que el título debe contener únicamente texto en minúscula o en mayúsculas.

## 2.3 – Consola Artisan

Desde la consola de comandos tenemos acceso al siguiente comando:

```
$ php artisan
```

Este nos permite acceder a los comandos de Laravel.

Ahora listaremos todas las **routes** que tenemos en el proyecto donde nos encontramos con el siguiente comando:

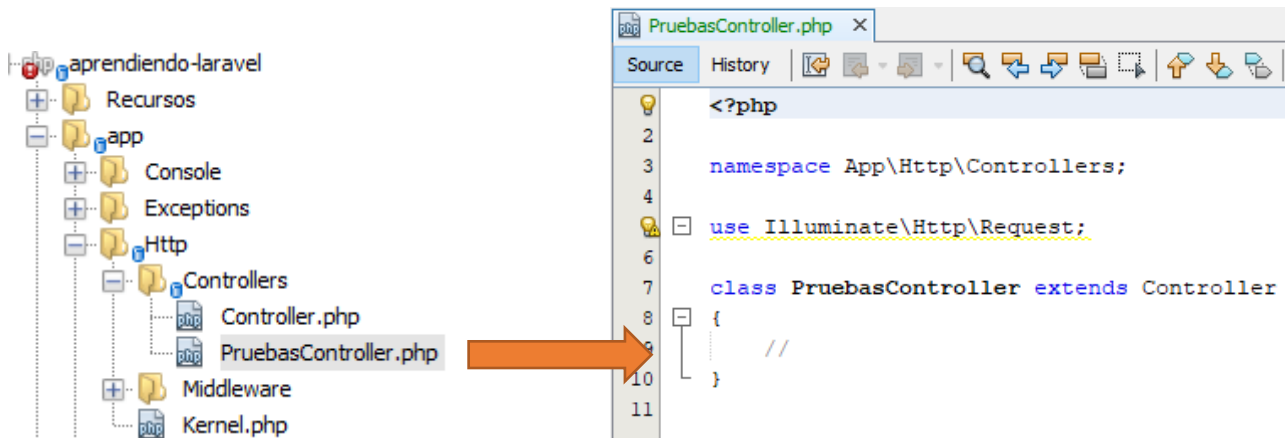
```
$ php artisan route:list
```

También podemos crear controladores o **Controllers**, que veremos en mayor profundidad más adelante, con el siguiente comando:

```
$ php artisan make:controller NombreController
```

Este comando nos genera un controlador en el proyecto con las clases correctas y los espacios de nombres correctos, como podemos ver en la siguiente imagen.

```
mmras@DESKTOP-UG00QBB /cygdrive/c/wamp64/www/master-php/aprendiendo-laravel
$ php artisan make:controller PruebasController
Controller created successfully.
```



Para ver los comandos más comunes podemos usar

```
$ php artisan help
```

O si queremos listar todos los comandos usaremos

```
$ php artisan list
```

## 3 – Plantillas y Vistas

---

### 3.1 – Vistas en Laravel

Vamos a crear una nueva **route** llamada listado-peliculas donde tendremos una nueva **view** llamada listado. Dentro del directorio **view** vamos a crear una nueva carpeta llamada **películas** y dentro de esta tendremos la **view** de listado.blade.php y todas las **views** relacionadas con películas.

Esta **view** al estar dentro de una carpeta deberemos indicarlo en la **route** de una forma especial, la sintaxis para ello sería la siguiente:

NombreCarpeta.NombreView

Si no queremos pasar los parámetros mediante un array disponemos de un método en la función **view** para indicarlo. El método es **with** que nos permite adjuntar variables nuevas, podemos llamarlo tantas veces como necesitemos.

```
Route::get('/listado-peliculas', function() {  
  
    $titulo = "Listado de peliculas";  
    $listado = array('Batman', 'Superman', 'Gran Torino');  
  
    return view('peliculas.listado')  
        ->with('titulo', $titulo)  
        ->with('listado', $listado);  
});
```

### 3.2 – Sistemas de plantillas Blade

Laravel utiliza Blade para trabajar con las **views** y las plantillas. Entre las ventajas de Blade es que nos permite la herencia de plantillas.

#### Imprimir por pantalla

Podemos mostrar el contenido de una variable con la **interpolación**. Para hacer esto usaremos las dobles llaves **{{ \$variable }}**

Como podemos ver a pesar de que Blade puede funcionar con PHP también tiene su propio lenguaje, que sería lo más correcto de usar en las **views**.

La intención de Blade es que las **views** tengan una sintaxis diferente a las **routes** o **controladores** para ayudar a la organización del proyecto.

Dentro de las dobles llaves también podemos poner funciones de php.

Para comentar en Blade usaremos la siguiente sintaxis

```
{{-- ESTO ES UN COMENTARIO --}}
```

En el caso de que queramos mostrar una variable en el caso de que exista, porque pudiera ser una variable opcional en el código podríamos usar la siguiente condición PHP para llevarlo a cabo.

```
<?php
isset($variable) ? $variable : 'No hay variable'
¿>
```

Esto es una condición ternaria que mostrara la variable si existe o imprimirá el texto 'No hay variable' en el caso de que no exista. Pero también podríamos hacerlo según la sintaxis de Blade de una forma mucho más rápida.

```
{{ $director ?? 'No hay director' }}
```

Obteniendo el mismo resultado.

## Condicionales

Blade tiene estructuras de control condicionales como otros lenguajes de programación ya que a fin de cuentas es solo una librería de php con su propia sintaxis para las vistas.

La sintaxis de los condicionales en Blade la podemos ver en el siguiente ejemplo:

```
@if($titulo)
    El título existe y es este: {{$titulo}}
@else
    El título no existe
@endif
```

## Bucles

Ahora veremos algunos ejemplos de bucles en Blade.

Ejemplo bucle for:

```
@for($i = 0; $i <= 20; $i++)
    El número es: {{$i}}
@endfor
```

Ejemplo bucle while

```
@while($contador < 50)
    @if($contador % 2 == 0)
        NUMERO PAR: {{$contador}} <br/>
    @endif

    <?php $contador++; ?>
@endwhile
```

Ejemplo bucle foreach

```
@foreach($listado as $pelicula)
    <p>{{$pelicula}}</p>
@endforeach
```

### Includes dentro de las views

En Blade, al igual que en PHP, podemos hacer un **include** de una plantilla dentro de otra. En PHP lo podemos hacer con cualquier fichero, pero en Laravel lo podemos hacer con la parte de las **views**. Para ejemplificar hemos creado la carpeta /include dentro de la carpeta /views y dentro de esta hemos creado dos views, header y footer.

Para incluir estas vistas dentro de otras plantillas deberemos usar la siguiente sintaxis

```
@include('carpeta-donde-están-los-includes.nombre-del-fichero')
```

En nuestro ejemplo sería así:

```
@include('includes.header')
```

En el punto del documento donde pongamos el include es donde se imprimirá el otro archivo del cual hacemos Includes, por ello si nos vamos a final y hacemos un include del footer este aparecerá al final del documento.

## 3.3 – Plantillas base o layout

Ahora vamos a aprender a utilizar las plantillas maestras, plantillas bases o layout.

Las plantillas en Laravel se definen en el directorio resources/views y dentro del directorio es recomendable tener otro directorio llamado layouts. Porque ahí van a estar nuestras plantillas y archivos comunes de nuestras vistas. Con Blade podemos definir una serie de bloques dentro de estas plantillas que luego serán usadas por las vistas normales para rellenar esas plantillas.

Podemos crear variables sustituibles en estas plantillas que creamos con **@yield('nombre-variable')**

Las secciones las definimos con **@section('nombre-sección')** y la sección la cerramos con la palabra **@show**

La diferencia entre @yield y @section es que con @yield no tenemos un contenido por defecto, pero con @section si podemos tener un contenido un predefinido.

A continuación, podemos ver un ejemplo de plantilla llamada master.

```
1 <!DOCTYPE html>
2 <html lang="es">
3   <head>
4     <title>Título - @yield('titulo')</title>
5     <meta charset="utf-8">
6     <meta name="title" content="@yield('titulo')">
7     <meta name="description" content="Descripción de la WEB">
8     <meta name="author" content="Miguel A. M. Rastau">
9   </head>
10  <body>
11    @section('header')
12      CABECERA DE LA WEB (master)
13    @show
14
15    <div class="container">
16      @yield('content')
17    </div>
18
19    @section('footer')
20      PIE DE PÁGINA DE LA WEB (master)
21    @show
22  </body>
23 </html>
```

Para que una **view** herede esta plantilla maestra deberemos usar al inicio de la misma la palabra **@extends** y a continuación deberemos sustituir el contenido de la plantilla por el contenido que nos interesa mostrar abriendo una sección con **@section** y deteniéndola con **@stop**, o si es solo texto como en el caso del título se podría usar **@section('titulo', 'Nuevo título')**.

Si en el lugar de sustituir el contenido de la plantilla maestra queremos agregar contenido a lo que ya existe en dicha plantilla deberemos usar la palabra **@parent()** dentro de la sección.

A continuación, un ejemplo de lo especificado.

```
1 @extends('layouts.master')
2
3 @section('titulo', 'Curso Laravel')
4
5 @section('header')
6   @parent()
7   <h2>Hola Mundo</h2>
8 @stop
9
10 @section('content')
11   <h1>Contenido de la página genérica</h1>
12 @stop
```

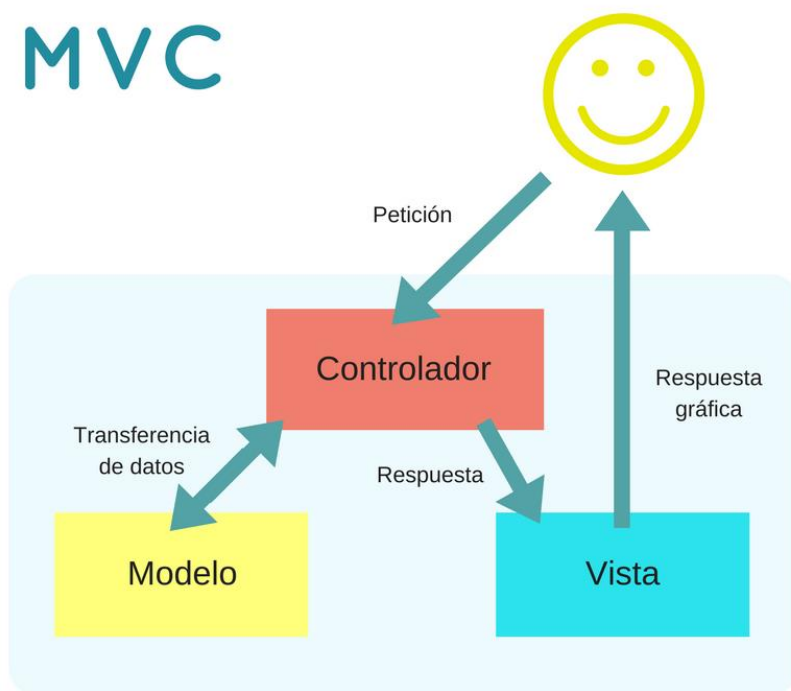
CABECERA DE LA WEB (master)

**Hola Mundo**

**Contenido de la página genérica**

PIE DE PÁGINA DE LA WEB (master)

## 4 – Controladores



Un Controlador es la parte del MVC (Modelo-Vista-Controlador) que se encarga de recibir datos de las Vistas, devolver datos a una Vista, hacer consultas al Modelo y hacer cierta lógica en la aplicación.

Podríamos crear nuestros controladores de Laravel de forma manual accediendo a `app/Http/Controllers` y creando un archivo en dicha ruta con el contenido que necesitamos o podríamos usar la consola para crearlo de forma automática.

### 4.1 – Controladores básicos

Para crear un controlador de forma correcta nos dirigiremos a la consola de comandos y con `artisan` crearemos el controlador usando el siguiente comando:

```
$ php artisan make:controller NombreController
```

Como hemos visto anteriormente.

En primera instancia este controlador no será más que una clase vacía, entonces ¿Cómo podemos usarlo?

Un controlador va a tener acciones, una acción no es nada más que un método de la clase.

Debemos crear una carpeta en **views** por cada controlador.

En el fichero web para no crear una **route** por cada **view** es más óptimo crear una **route** por cada acción del controlador, para ello lo haremos como se muestra en el ejemplo para vincular la **route** indicada con la acción (o método) del controlador indicado.

**Desde Laravel 8 vamos a necesitar:**

**Importar el namespace completo del controller que deseamos usar**

**Deberemos (para este caso) seguir la sintaxis de: [AlgoController::class, 'metodo']**

Quedando, nuestro ejemplo, de esta forma entonces:

```
use App\Http\Controllers\PeliculaController;  
Route::get('/peliculas', [PeliculaController::class, 'index']);
```

### \*\*\*IMPORTANTE\*\*\*

(

Actualización del manual:

Una vez creada la ruta como se ve en el ejemplo anterior debemos definir un nombre como podemos ver a continuación:

```
Route::get('/detalle', [PeliculaController::class, 'detalle'])->name('pelicula.detalles');
```

Debido a las actualizaciones de Laravel 8 ahora es obligatorio indicar el nombre para poder usar el nombre de la ruta, si no nos dará un error de que dicho nombre no existe.

De esta forma podemos utilizar el nombre con normalidad para rutas y acciones

)

El contenido del controlador es el siguiente:

```
<?php
```

```
namespace App\Http\Controllers;
```

```
use Illuminate\Http\Request;
```

```
class PeliculaController extends Controller
```

```
{
```

```
    public function index() {
```

```
        $titulo = 'Listado de mis películas';
```

```
        return view('pelicula.index', [
```

```
            'titulo' => $titulo
```

```
        ]);
```

```
    }
```

```
}
```



## 4.2 – Controladores Resource

Podríamos crear una ruta por cada uno de los métodos de la clase en el caso de que necesitemos un nivel de personalización muy grande, pero puede ser un poco engorroso. Para esto, Laravel nos da la solución que son los controladores de tipo de resource. Esto lo que hace es generar un controlador que de manera automática tiene una serie de rutas y métodos ya establecidos. Para crear un controlador de tipo resource debemos dirigirnos a la consola y utilizar el siguiente comando de Artisan.

```
$ php artisan make:controller NombreController --resource
```

El siguiente paso para poder tener todos los métodos creados automáticamente en una ruta es crear una ruta de tipo resource, en el archivo de las rutas, con la siguiente sintaxis:

```
Route::resource('nombre-ruta', NombreControllerResource::class);
```


Veamos un ejemplo:

```
use App\Http\Controllers\UsuarioController;  
Route::resource('usuario', UsuarioController::class);
```

## 4.3 – Enlaces en Laravel

Para generar una URL a una acción, es decir a otra ruta debemos indicar el namespace completo en el método **action**, como se muestra en el ejemplo:

```
<a href="{{action('App\Http\Controllers\PeliculaController@index')}}">  
    Ir al listado  
</a>
```



```
class PeliculaController extends Controller  
{  
    public function index($pagina = 1){  
        $titulo = 'Listado de mis peliculas';  
  
        return view('pelicula.index', [  
            'titulo' => $titulo,  
            'pagina' => $pagina  
        ]);  
    }  
}
```

También podemos usar **route** indicando el nombre completo de la ruta a la que queremos ir o bien si le hemos definido un nombre a la ruta

```
<a href="{{route('pelicula.detalle')}}">
    Ir al detalle
</a>
```

Si la URL permite argumentos se deben pasar como se muestra a continuación:

```
<a href="{{route('pelicula.detalle', ['id' => 12])}}">
    Ir al detalle 12
</a>
```

## 4.4 – Redirecciones

También podemos redireccionar desde una acción del controlador a otra fácilmente, podemos redireccionar desde una URL a otra de manera programática, es decir que el usuario no debería darle a un enlace ni nada, para ello crearemos una función como podemos ver a continuación:

```
public function redirigir(){
    return redirect()->action('App\Http\Controllers\PeliculaController@detalle');
}
```

\*También podemos usar **redirect** indicando la ruta a la que queremos ir directamente o usar el método **route**, de esta manera:

```
redirect(/ruta-a-redirigir);
redirect()->route(/ruta-a-redirigir)
```

Posteriormente crearemos la **route** para redirigir:

```
Route::get('/redirigir', [PeliculaController::class, 'redirigir'])
    ->name('pelicula.redirigir');
```

## 4.5 – Middleware

Ahora vamos a aprender a trabajar con los middlewares o filtros. Estos son componentes que tiene Laravel que nos permite filtrar las peticiones que hacemos mediante HTTP. De forma que un middleware al final es una **clase php** que nos permite comprobar o hacer cierta lógica antes de mostrar una página o antes de que se ejecute la acción de un controlador, de forma que podemos evaluar ciertas cosas, hacer cierta lógica antes de mostrar la página.

Por ejemplo, podemos comprobar si un parámetro nos llega o si tiene cierto valor para dejar pasar o no al usuario a la página, de forma que el middleware aplica un filtro y decide si la petición se realiza, si se permite la ejecución de una acción, si da un error, si redirecciona a otra página o cualquier cosa que haga.

Ahora vemos como se crea con un ejemplo:

**Primero:** Indicamos en la ruta de detalle que existe un parámetro opcional llamado **year**.

```
Route::get('/detalle/{year?}', [PelículaController::class, 'detalle'])
    ->name('película.detalle');
```

**Segundo:** Indicamos en el controlador PelículaController que este valor por defecto vale **null**

```
public function detalle($year = null) {
    return view('película.detalle');
}
```

**Tercero:** Vamos a dejar pasar al usuario si el parámetro vale 2019 y si no le redirigimos a otra, para esto vamos a crear un middleware. Para crearlo lo haremos desde la consola artisan con el siguiente comando

```
$ php artisan make:middleware nombreMiddleware
```

Esto nos creará un middleware que se guardará en la carpeta Middleware dentro de http.

Dentro de esta clase que hemos creado comprobamos la lógica propuesta:

```
public function handle(Request $request, Closure $next)
{
    $year = $request->route('year');

    if(is_null($year) || $year != 2019 ){
        return redirect('/películas');
    }

    return $next($request);
}
```

**Cuarto:** Nos dirigimos al archivo kernel en la carpeta http e indicamos en el routeMiddleware el nombre con el que usaremos el middleware.

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'password.confirm' => \Illuminate\Auth\Middleware\RequirePassword::class,
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
    'testyear' => \App\Http\Middleware\TestYear::class,
];
```

**Quinto:** Indicamos en la ruta el middleware que se usará con el método middleware.

```
Route::get('/detalle/{year?}', [PelículaController::class, 'detalle'])
    ->name('película.detalle')
    ->middleware('testyear');
```

## 5 – Formularios

---

En primer lugar crearemos una acción que nos lleve a una vista con un formulario:

```
public function formulario() {  
    return view('pelicula.formulario');  
}
```

La construcción del formulario es, como podemos apreciar a continuación, igual que en html:

```
<h1>Formulario en Laravel</h1>  
<form action="" method="POST" >  
    <label for="nombre">Nombre</label>  
    <input type="text" name="nombre" />  
  
    <label for="email">Correo</label>  
    <input type="email" name="email" />  
  
    <input type="submit" value="Enviar" />  
</form>
```

A continuación, crearemos la **route**:

```
Route::get('/formulario', [PeliculaController::class, 'formulario'])  
    ->name('pelicula.formulario');
```

Ahora debemos asegurarnos de recibir los datos del formulario en una acción del controlador, creamos la función recibir:

```
public function recibir(Request $request) {  
    $nombre = $request->input('nombre');  
  
    var_dump($nombre);  
    die();  
}
```

---

Para poder ver los datos que recogemos del formulario con la función recibir debemos crear una nueva ruta de tipo post donde se mostraran estos datos:

```
Route::post('/recibir', [PelículaController::class, 'recibir'])  
    ->name('película.recibir');
```

Y en la acción del formulario la acción correspondiente como podemos ver:

```
<h1>Formulario en Laravel</h1>  
<form action="{{ action('App\Http\Controllers\PelículaController@recibir') }}" method="POST" >
```

**\*\* IMPORTANTE \*\***

Para que los formularios funcionen en Laravel debemos incluir protección contra ataques csrf, entonces debemos utilizar en nuestros formularios la siguiente funcionalidad incluida en Laravel.

```
<h1>Formulario en Laravel</h1>  
<form action="{{ action('App\Http\Controllers\PelículaController@recibir') }}" method="POST" >  
    {{ csrf_field() }}  
    <label for="nombre">Nombre</label>  
    <input type="text" name="nombre" value="{{ old('nombre') }}" />
```

Con todo esto el formulario ya sería funcional.

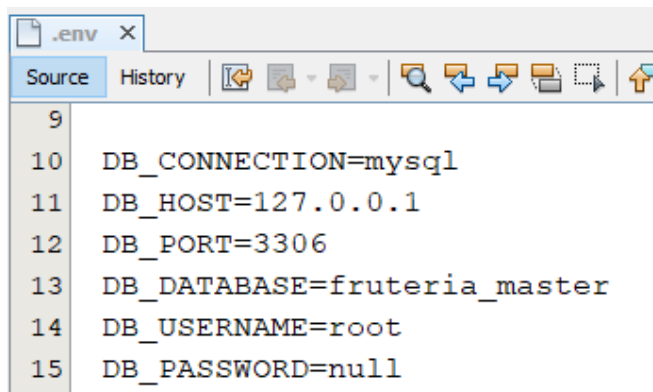
## 6 – Bases de datos

### 6.1 – Conectarse a una base de datos con Laravel

En primer lugar, vamos a crear una base de datos con PhpMyAdmin.

En archivos como `.env` podemos modificar varios aspectos del proyecto, como su URL, si se van a mostrar los errores, el tipo de conexión a la base de datos, etc.

Nos dirigimos a este archivo y modificamos el nombre de la base de datos de este proyecto, que hemos creado anteriormente y que se llama **frutería\_master**, además del nombre del usuario y la contraseña.



Como en este caso no tenemos contraseña debemos poner **null**.

### 6.2 – Migraciones

Las migraciones nos permiten versionar nuestra base de datos de forma que si nosotros estamos trabajando en nuestro proyecto y se nos ocurren hacer ciertas modificaciones nos permite trabajar de manera programática, de forma que si le pasamos el proyecto a un compañero podrá, al ejecutar el proyecto, levantar esos cambios que hemos hecho en la base de datos de manera más sencilla.

Para crearla ejecutamos el siguiente comando desde la consola:

```
$ php artisan make:migration nombre_migracion --table=nombre_tabla
```

Esta migración la podemos encontrar en la carpeta

**database/migrations**

Ahora dentro de este archivo deberemos definir,

**obligatoriamente**, los campos de la columna de la tabla que se va a crear dentro de la función **up()**

```
class CreateUsuariosTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('usuarios', function (Blueprint $table) {
            $table->increments('id');
            $table->string('nombre', 255);
            $table->string('email', 255);
            $table->string('password', 255);
            $table->integer('edad');
            $table->timestamps();
        });
    }
}
```

```

public function down()
{
    Schema::drop('usuarios');
}

```

El evento up crea la tabla y el evento down borra la tabla, por lo que en el método down() pondremos el siguiente método schema para borrar la tabla.

En el caso de que queramos modificar las tablas cambiaremos el método **create** por el método **table**

```

class CreateUsuariosTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('usuarios', function (Blueprint $table) {
            $table->increments('id');
            $table->string('nombre', 255);
            $table->string('email', 255);
            $table->string('password', 255);
            $table->integer('edad');
            $table->timestamps();
        });
    }
}

```

Ahora veremos cómo lanzar una migración. En el momento en el que lanzamos una migración se van a ejecutar cambios en la base de datos, para ello desde la consola ejecutaremos el siguiente comando:

```
$ php artisan migrate
```

Este comando ejecutará todas las migraciones que tengamos en la carpeta **migrations**.

\*\*\*\*\*

**\*\*\* IMPORTANTE \*\*\***

**Por algún motivo la configuración por defecto de Laravel me ha devuelto el siguiente error:**

SQLSTATE[42000]: Syntax error or access violation: 1071 Specified key was too long; max key length is 1000 bytes (SQL: alter table `users` add unique `users\_email\_unique`(`email`))



Existen tres posible soluciones:

- 1- Actualizar la versión de MySQL
- 2- Modificar el archivo config/database
- 3- Modificar el archivo app/Providers/AppServiceProvider.php

#### Solución 1

Si utilizas MariaDB o versiones anteriores de MySQL v5.7.7 simplemente actualiza a la versión estable más reciente y no volverás a tener este problema.

#### Solución 2

Si no puedes o no quieres actualizar MySQL, intenta modificando el engine de mysql en el archivo config/database.php

```
'mysql' => [  
    'engine' => 'InnoDB',  
]
```

#### Solución 3

Si por algún motivo no te funciona la solución 2, agrega la siguiente línea al archivo app/Providers/AppServiceProvider.php

```
use Illuminate\Support\Facades\Schema;  
public function boot()  
{  
    Schema::defaultStringLength(191);  
}
```

¿Por qué ocurre este error?

Desde la versión 5.4 de Laravel se realizó un cambio en el conjunto de caracteres predeterminado de la base de datos, y ahora es utf8mb4, que incluye soporte para almacenar emojis.

Esto solo afecta a las aplicaciones que utilicen versiones anteriores a MySQL v5.7.7

#### LA SOLUCIÓN 2 ME HA FUNCIONADO

\*\*\*\*\*

Para revertir una migración usaremos el siguiente comando en la consola:

```
$ php artisan migrate:rollback
```

En el caso de que no solo queramos revertirlo para volver a cargarlo si no que queremos refrescarlo porque hemos modificado la migración usaremos el siguiente comando:

```
$ php artisan migrate:refresh
```

Este comando borra las migraciones y las vuelve a ejecutar.

## 6.3 – Migraciones con SQL

Ya hemos visto cómo hacer migraciones con el sistema schema builder, pero también podemos hacerlo con SQL, para ello usaremos el objeto DB con el método statement como podemos ver a continuación:

```
DB::statement("CREATE TABLE usuarios(  
    id            int(255) auto_increment not null,  
    nombre        varchar(255),  
    email         varchar(255),  
    password      varchar(255),  
    PRIMARY KEY (id)  
);");
```

Y esto nos funcionará exactamente igual que lo que hemos hecho anteriormente.

## 6.4 – Seeders

Son un mecanismo para rellenar de datos nuestras tablas.

Para crear un seed o seeder ejecutamos el siguiente comando en la consola:

```
$ php artisan make:seed nombre_seed
```

Esto nos creará un archivo en la carpeta database>seeds

Dentro de este archivo tenemos el método run, dentro de este método crearemos el código que creará los nuevos registros, en este caso vamos a crear un bucle para que inserte una línea nueva a la tabla por vuelta. Y posteriormente mostraremos un mensaje por la consola de comando indicando que se ha finalizado correctamente la inserción.

Con `DB::table('tabla')` accedemos a la tabla indicada. Luego usamos el método `insert` y podemos insertar en la tabla los datos que indiquemos en el array.

```
public function run()
{
    for($i = 1; $i <= 20; $i++){
        DB::table('fruta')->insert(array(
            'nombre'=>'Cereza '.$i,
            'descripcion'=>'Descripción de la fruta '.$i
        ));
    }

    $this->command->info('La tabla de frutas ha sido rellena');
}
```

Con esta línea hacemos que ese mensaje salga por la consola de comandos.

\*\*\*\*\* IMPORTANTE \*\*\*\*\*

Hay que añadir la siguiente clase en el archivo.

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

\*\*\*\*\*

Para ejecutar el seed que hemos creado debemos introducir el siguiente comando en la consola:

```
$ php artisan db:seed --class=nombre_seed
```

## 6.5 – Listar datos

Vamos a realizar un CRUD con nuestra base de datos para ver todos los aspectos del manejo de bases de datos con Laravel.

(El concepto CRUD está estrechamente vinculado a la gestión de datos digitales. CRUD hace referencia a un acrónimo en el que se reúnen las primeras letras de las cuatro operaciones fundamentales de aplicaciones persistentes en sistemas de bases de datos:

- Create (Crear registros)
- Read. Retrieve (Leer registros)
- Update (Actualizar registros)
- Delete. Destroy (Borrar registros)

)

Para trabajar con la base de datos usando querybuilder en Laravel vamos a dirigirnos a la carpeta Controllers, en `app>Http>Controllers`

En esta carpeta podemos ver que tenemos los controladores que hemos creado anteriormente, pero ahora nos interesa realizar un CRUD de “frutas” y para ello vamos a crear un nuevo controlador que se llamara FrutaController.

Una vez creado nos dirigimos al controlador y comenzamos a crear los métodos que necesitamos para el CRUD

El primer método que vamos a crear será **index** y este método va a listar las frutas.

```
class FrutaController extends Controller
{
    public function index() {
        $frutas = DB::table('frutas')->get();

        return view('fruta.index', [
            'frutas' => $frutas
        ]);
    }
}
```

Usaremos el objeto **DB** con el método **table** para seleccionar la tabla y posteriormente el método **get** para sacar todos los registros de la base de datos. Estos métodos son el equivalente en SQL a hacer un “SELECT \* FROM frutas”. Estos métodos nos devuelven un array de objetos, por lo que vamos a

crearnos una **view** llamada **fruta.index** y dentro de esta vista vamos a listar el contenido, por lo que deberemos pasarle el array de objetos desde el **controlador** a la **view** usando los corchetes como vemos en la imagen.

En la vista que hemos creado usaremos un bucle **foreach** para recorrer el array de objetos y mostrar el nombre de cada fruta.

También deberemos crear la ruta a dicha vista en el fichero de rutas.

```
<h1>Listado de frutas</h1>
<ul>
    @foreach($frutas as $fruta)
        <li>{{ $fruta->nombre }}</li>
    @endforeach
</ul>
```

Para ello vamos a crear un grupo de rutas donde indicaremos el prefijo de cada una de ellas, en este caso el prefijo de todas las rutas será “fruta” por lo que lo crearemos como podemos ver en la imagen.

```
//Rutas frutas
use App\Http\Controllers\FrutaController;
Route::group(['prefix'=>'fruta'], function() {
    Route::get('index', [FrutaController::class, 'index'])
        ->name('fruta.index');
});
```

## 6.6 – Mostrar una fila

Vamos a mostrar el detalle de una fruta en concreto, para ello vamos a crear un nuevo método en el controlador donde necesitamos hacer un `where` para indicar que fruta queremos listar concretamente para mostrar su detalle en base a un `id` que nos va a llegar por la URL.

Para ello usaremos el método `where` para indicar la condición que queremos, este método se compone de 3 parámetros: Primero el campo que queremos evaluar, segundo el operador que queremos utilizar y por último por donde le vamos a pasar el dato que en este caso será una variable llegada desde la URL.

```
public function detail($id) {  
    $fruta = DB::table('frutas')->where('id', '=', $id)->first();  
  
    return view('fruta.detail', [  
        'fruta' => $fruta  
    ]);  
}
```

En este caso la condición que tenemos es que nos muestre todas las frutas cuyo `id` coincida con el parámetro dado por la URL. Y por último devolvemos toda esta información a una **view**.

\*\*\*\*\***IMPORTANTE**\*\*\*\*\*

Cuando usamos el método `where` debemos finalizarlo siempre con el método `get()` o el método `first()` si queremos que nos devuelva un objeto limpio, de lo contrario Laravel nos devolverá un error de sintaxis.

\*\*\*\*\*

Finalmente creamos una ruta en el grupo de rutas que tenemos preparado para frutas, indicando que la url debe tener un parámetro obligatorio para el `id`.

```
Route::get('detail/{id}', [FrutaController::class, 'detail'])  
    ->name('fruta.detail');
```

Ahora vamos a realizar una serie de enlaces en `frutas.index` para que nos lleve al detail de la fruta en concreto. Para ello en la vista de `frutas.index` haremos lo siguiente:

```
<h1>Listado de frutas</h1>

<ul>
    @foreach($frutas as $fruta)
        <li>
            <a href="{{ action('App\Http\Controllers\FrutaController@detail', ['id' =>$fruta->id] ) }}">
                {{ $fruta->nombre }}
            </a>
        </li>
    @endforeach
</ul>
```

## 6.7 – Order by

Supongamos, a modo de ejercicio, que queremos invertir el orden del listado del index. Para ello usaremos la cláusula **order by**, o en este caso el método `orderBy()` de la siguiente forma:

```
public function index() {
    $frutas = DB::table('frutas')
        ->orderBy('id', 'desc')
        ->get();
}
```

El método `orderBy()` tiene dos parámetros, el primero es el campo de referencia para la ordenación, el segundo el tipo de orden que queremos si ascendente (`asc`) o descendente (`desc`)

Podemos encontrar todos los métodos que son cláusulas en SQL en la documentación de Laravel.

## 6.8 – Insertar Registros

Ahora vamos a crear una página donde tengamos un formulario que añada registros a la base de datos. Esta página será **fruta.create** que será un formulario para crear frutas en la base de datos.

```
<h1>Crear fruta</h1>
<form action="{{ action('App\Http\Controllers\FrutaController@save') }}" method="POST">
    {{ csrf_field() }}

    <label for="nombre">Nombre</label>
    <input type="text" name="nombre" />
    <br/>

    <label for="precio">Precio</label>
    <input type="text" name="precio" />
    <br/>

    <label for="descripcion">Descripcion</label>
    <input type="text" name="descripcion" />
    <br/>

    <input type="submit" value='Guardar' />
</form>
```

Ahora debemos crear las **routes** de los métodos para el formulario (que veremos a continuación)

```
Route::get('create', [FrutaController::class, 'create'])
    ->name('fruta.create');
Route::post('save', [FrutaController::class, 'save'])
    ->name('fruta.save');
```

En FrutasController crearemos el método create para mostrar la **view** anterior y otro método para la acción de guardar que guardará los registros recibidos por el formulario como podemos ver en la imagen:

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

public function save(Request $request) {
    //guardar el registro
    $fruta = DB::table('frutas')->insert(array(
        'nombre' => $request->input('nombre'),
        'descripcion' => $request->input('descripcion'),
        'precio' => $request->input('precio'),
        'fecha' => date('Y-m-d')
    ));

    //redirigimos tras guardar la fruta
    return redirect()->action('App\Http\Controllers\FrutaController@index');
}
```

## 6.9 – Borrar Registros

Desde la página de detalle vamos a añadir la posibilidad de borrar el registro que estamos viendo. En dicha página vamos a añadir a un enlace que nos llevará a un método del controlador que borrará el registro, este método lo crearemos más adelante.

```
<a href="{{ action('App\Http\Controllers\FrutaController@delete', ['id' => $fruta->id]) }}">
    Eliminar
</a>
```

Crearemos en FrutasController una acción que nos permita borrar un registro según el id indicado, también añadiremos una redirección al index con un with que es una sesión flash (es decir de un solo uso) que nos indique que el registro ha sido borrado exitosamente. Como añadido haremos lo mismo en la acción de crear registros que hicimos anteriormente.

```
public function delete($id) {
    $fruta = DB::table('frutas')
        ->where('id', '=', $id)
        ->delete();

    return redirect()->action('App\Http\Controllers\FrutaController@index')
        ->with('status', 'Fruta borrada correctamente');
}
```

Y por último crearemos la **route** necesaria en el grupo de rutas.

```
Route::get('delete/{id}', [FrutaController::class, 'delete'])
    ->name('fruta.delete');
```

En el index debemos asegurarnos de mostrar la sesión flash status que hemos creado, si existe, con un if como podemos ver a continuación:

```
@if(session('status'))
    <p style="background: green; ">
        {{session('status')}}
    </p>
@endif
```



## 6.10 – Actualizar Registros

Ahora vamos a actualizar los registros desde un enlace que incluiremos en la vista de detalle.

```
<a href="{{ action('App\Http\Controllers\FrutaController@edit', ['id' => $fruta->id]) }}">
    Actualizar
</a>
```

Este enlace nos llevara a una acción del controlador FrutaController que primero sacará el registro de la base de datos y luego nos mostrará una vista con un formulario relleno con los datos que hemos sacado para que podamos modificarlos y posteriormente mandarlos a la BD para que se editen.

```
public function edit($id) {
    //sacar el registro de la base de datos
    $fruta = DB::table('frutas')
        ->where('id', '=', $id)
        ->first();
    //pasarle a la vista el objeto y rellenar el formulario
    return view('fruta.create', [
        'fruta' => $fruta
    ]);
}
```

No sin antes crear la ruta en el grupo de rutas:

```
Route::get('edit/{id}', [FrutaController::class, 'edit'])
    ->name('fruta.edit');
```

En la acción de edit vamos a reutilizar la vista de create, por lo que deberemos modificarla para hacer unas comprobaciones. Como desde la acción de edit vamos a mandar un objeto a la vista de create vamos a comprobar si se está recibiendo un objeto en la vista create y de ser así modificar el título de Crear fruta por Modificar fruta, además de rellenar los datos del formulario con los datos del objeto si este existe y dejarlo en blanco en caso contrario. También deberemos modificar la acción en función de si existe el objeto fruta o no, para saber si el formulario añadirá un registro nuevo o editará uno existente con los datos del formulario. También pasaremos el id por un campo hidden en caso de que \$fruta exista. Porque luego este dato será recogido por la acción update y podemos pasarle todos los datos por cabeceras.

```
@if(isset($fruta) && is_object($fruta))
    <h1>Editar fruta</h1>
@else
    <h1>Crear fruta</h1>
@endif

<form action="{{ isset($fruta) ? action('App\Http\Controllers\FrutaController@update')
: action('App\Http\Controllers\FrutaController@save') }}"
    method="POST">
    {{csrf_field()}}

@if(isset($fruta) && is_object($fruta))
    <input type="hidden" name="id" value="{{ $fruta->id }}" />
@endif

    <label for="nombre">Nombre</label>
    <input type="text" name="nombre" value="{{ $fruta->nombre ?? '' }}" />
    <br/>

    <label for="precio">Precio</label>
    <input type="text" name="precio" value="{{ $fruta->precio ?? 0 }}" />
    <br/>

    <label for="descripcion">Descripcion</label>
    <input type="text" name="descripcion" value="{{ $fruta->descripcion ?? '' }}" />
    <br/>

    <input type="submit" value='Guardar' />
</form>
```

Como podemos ver hemos utilizado una condición ternaria en el action de tal manera que si existe \$fruta llamará a la acción **update**, que veremos a continuación, y en caso de no existir dicho elemento irá a la acción **save**.

Ahora vamos a crear la ruta a la acción update:

```
Route::post('update', [FrutaController::class, 'update'])
    ->name('fruta.update');
```

Como podemos ver es una ruta de tipo post porque recogerá los datos del formulario, como podemos ver a continuación:

```
public function update(Request $request) {
    $id = $request->input('id');

    $fruta = DB::table('frutas')
        ->where('id', '=', $id)
        ->update(array(
            'nombre' => $request->input('nombre'),
            'descripcion' => $request->input('descripcion'),
            'precio' => $request->input('precio')
        ));

    return redirect()->action('App\Http\Controllers\FrutaController@index')
        ->with('status', 'Fruta actualizada correctamente');
}
```

## Documentación

---

Curso Udemy

<https://www.udemy.com/course/master-en-php-sql-poo-mvc-laravel-symfony-4-wordpress>

Documentación instalación de Laravel

<https://laravel.com/docs/8.x/installation>

Crear un Host Virtual

<https://victorroblesweb.es/2016/03/26/crear-varios-hosts-virtuales-en-wampserver/>

Documentación Blade

<https://styde.net/laravel-6-doc-plantillas-blade/>

Documentación migraciones en Laravel

<https://laravel.com/docs/8.x/migrations>

Solucionar posible error con las migraciones

<https://aprendible.com/blog/como-solucionar-el-error-specified-key-was-too-long-error-en-laravel>