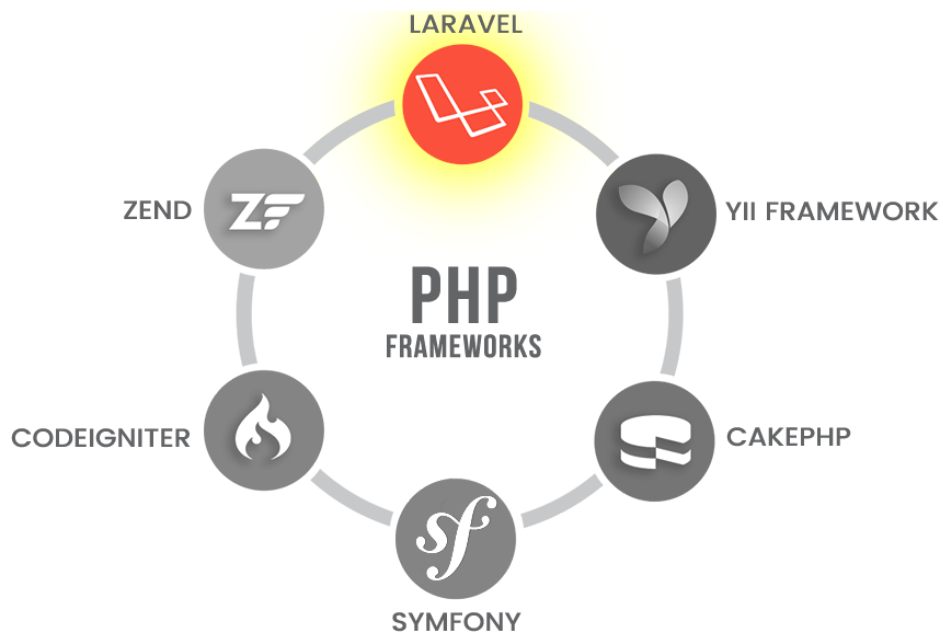


## unidad didáctica 7

# Framework Laravel



1. **duración y criterios de evaluación**
2. **consideraciones previas**
  2. 1. **MVC**
    2. 1. 1. **Modelo**
    2. 1. 2. **Vista**
    2. 1. 3. **Controlador**
    2. 1. 4. **Router**
  2. 2. **Artisan**
3. **instalar docker bitnami/Laravel**
  3. 1. **VSCode extensiones**
4. **carpetas en Laravel**
  4. 1. **app/Http/Controllers**
  4. 2. **Models**
  4. 3. **public**
  4. 4. **resources**
  4. 5. **routes**
  4. 6. **vendor**
  4. 7. **.env**
5. **rutas**
  5. 1. **alias**
  5. 2. **parámetros**
6. **plantillas o templates**
  6. 1. **directivas**
  6. 2. **separando código**
  6. 3. **estructuras de control**
7. **controladores**
  7. 1. **convenciones**
8. **tipos de Request**
  8. 1. **En HTTP y API's**
9. **anexo I - instalación de Tailwind CSS**
10. **anexo II - reinstalación de node**
11. **ejemplos**
  11. 1. **ejemplo 01. Hola Mundo**
  11. 2. **ejemplo 02. Otras vistas**
  11. 3. **ejemplo 03. Uso de directivas**
  11. 4. **ejemplo 04. Vista registrarse**
  11. 5. **ejemplo 05. controlador RegisterController y su formulario**
  11. 6. **ejemplo 06. petición post**
12. **referencias**

# 1. duración y criterios de evaluación

---

**Duración estimada:** ∞ sesiones.

---

**Resultado de aprendizaje y criterios de evaluación:**

---

## 2. consideraciones previas

---

### 2.1. MVC

---

**MVC** (Model View Controller o Modelo Vista Controlador) es un patrón de arquitectura de software que permite la separación de obligaciones de cada pieza de tu código.

Este paradigma de la programación enfatiza la separación de la lógica de programación con la presentación.

#### Ventajas:

- MVC no mejora el performance del código, tampoco da seguridad; pero **tu código tendrá un mejor orden y será fácil de mantener**.
- En un grupo de trabajo, el tener el código ordenado permite que más de una persona pueda entender que es lo que hace cada parte de él.
- Aprender MVC, te hará que otras tecnologías como *Nest, Rails, Django, Net Core, Spring Boot* te serán más sencillas de aprender.

#### 2.1.1. Modelo

Encargado de todas las interacciones en la base de datos (obtener datos, actualizarlos y eliminar).

El Modelo se encarga de consultar una base de datos, obtiene la información pero no la muestra (esa tarea es para las vistas).

El Modelo tampoco se encarga de actualizar la información directamente (esa tarea es del Controlador, que es quien decide cuándo llamarlo).

#### 2.1.2. Vista

Se encarga de todo lo que se ve en pantalla (HTML).

Laravel tiene un Template Engine llamado **Blade** para mostrar los datos.

Si utilizas *React, Vue, Angular, Svelte*, etc. estos serían tu vista.

El Modelo consulta la base de datos, pero es por medio del Controlador que se decide qué Vista hay que llamar y qué datos presentar.

#### 2.1.3. Controlador

Es el que comunica *Modelo* y *Vista*; antes de que el *Modelo* consulte la base de datos el *Controlador* es el encargado de llamar un *Modelo* en específico.

Una vez consultado el *Modelo*, el *Controlador* recibe esa información, manda llamar a la *Vista* y le pasa la información.

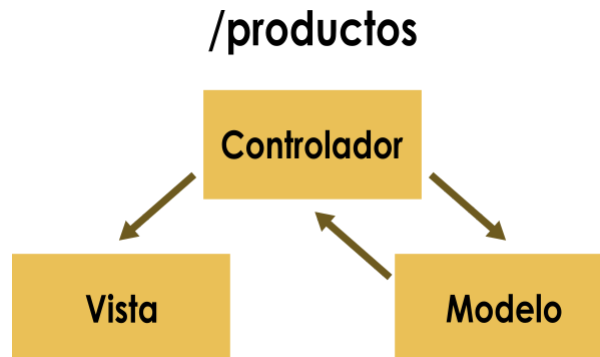
El *Controlador* es el que manda llamar la *Vista* y *Modelos*, que se requieren en cada parte de tu aplicación.

## 2.1.4. Router

Es el encargado de registrar todas las URL's o Endpoints que va a soportar nuestra aplicación.

Ejemplo:

Si el Usuario accede a `/productos` el router ya tiene registrada esa ruta y un controlador con una función que sabe que Modelo debe llamar y que vista mostrar cuando el usuario visita esa URL.



## 2.2. Artisan

---

Artisan es el CLI (Command Line Interface) incluido en Laravel.

Artisan es un script que existe en la base de tu proyecto de Laravel y cuenta con una gran cantidad de scripts disponibles.

Estos comandos te permiten crear *migraciones*, *controladores*, *modelos*, *policies* y mucho más.

Todos los comando podemos encontrarlos ejecutando:

```
1 # php artisan
2 # ó
3 sudo docker-compose exec myapp php artisan
```

Por ejemplo, si quisiéramos crear un controlador (como veremos más adelante) la orden sería:

```
1 # php artisan make:controller RegisterController
2 # ó
3 sudo docker-compose exec myapp php artisan make:controller RegisterController
```

### 3. instalar docker bitnami/Laravel

1. Lo primero de todo es crear una carpeta con el nombre del proyecto que vayamos a crear y nos metemos en ella.

Por ejemplo, creamos el proyecto myapp-laravel dentro de nuestra carpeta de proyectos del módulo:

```
1 $ mkdir ~/dwes/proyectos/myapp-laravel
```

2. Accedemos dentro de la carpeta de este nuevo proyecto.

3. En este punto tenemos dos opciones:

A. Utilizar la imagen de Bitnami ya preparada, así que lo que hacer ahora es [descargar el archivo docker-compose.yml](#) del repositorio de Github oficial.

```
1 curl -LO
https://raw.githubusercontent.com/bitnami/containers/main/bitnami/laravel/docker-compose.yml
```

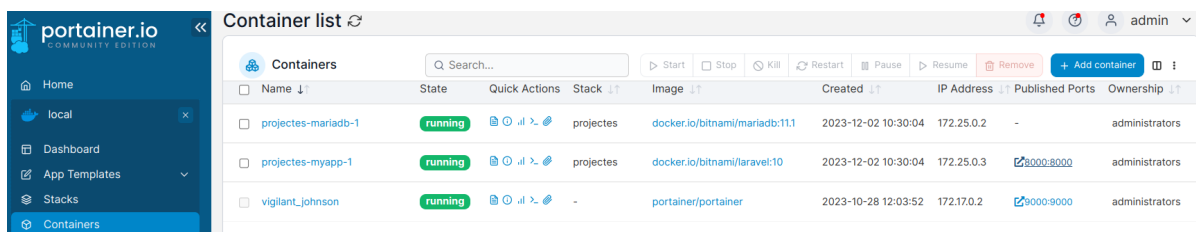
B. Utilizar el fichero `docker-compose.yml` que tenemos en nuestra carpeta del curso (en el que se añade el contenedor para *phpMyadmin*).

En caso ejecutaremos la opción B.

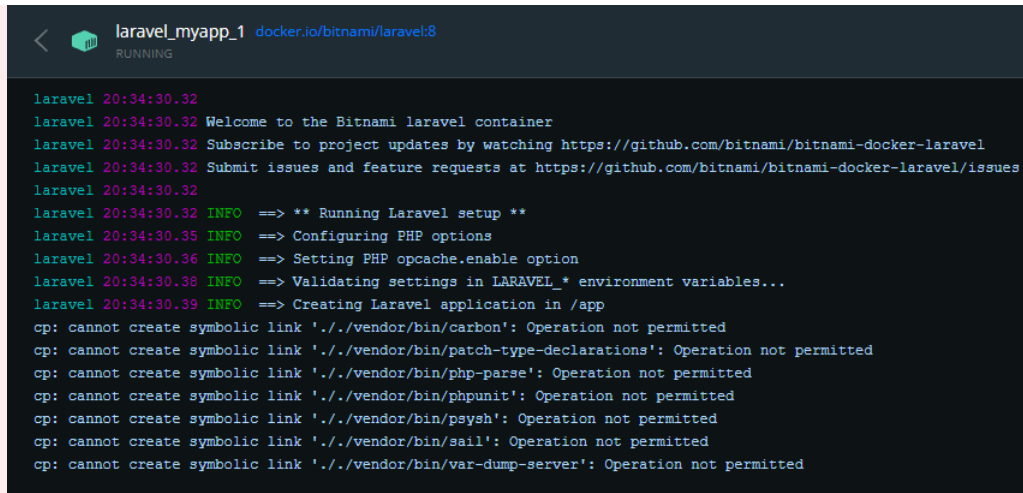
4. Una vez descargado el archivo en nuestra carpeta que acabamos de crear con el nombre del proyecto, lanzamos el siguiente comando por consola para instalar todas las dependencias y crear las imágenes de Docker correspondientes.

```
1 sudo docker-compose up -d
```

5. Si utilizamos el contenedor `Portainer` para la gestión de nuestros contenedores, podremos observar que estarán en marcha nuestros dos contenedores (pertenecientes al servidor web y servidor de bases de datos):



Si por alguna extraña razón estás en Windows y no te funciona una de las 2 imágenes, puede ser debido a la instalación de composer dentro de la imagen de Laravel.



```

laravel_myapp_1 docker.io/bitnami/laravel:8
RUNNING

laravel 20:34:30.32
laravel 20:34:30.32 Welcome to the Bitnami laravel container
laravel 20:34:30.32 Subscribe to project updates by watching https://github.com/bitnami/bitnami-docker-laravel
laravel 20:34:30.32 Submit issues and feature requests at https://github.com/bitnami/bitnami-docker-laravel/issues
laravel 20:34:30.32
laravel 20:34:30.32 INFO ==> ** Running Laravel setup **
laravel 20:34:30.35 INFO ==> Configuring PHP options
laravel 20:34:30.36 INFO ==> Setting PHP opcache.enable option
laravel 20:34:30.38 INFO ==> Validating settings in LARAVEL_* environment variables...
laravel 20:34:30.39 INFO ==> Creating Laravel application in /app
cp: cannot create symbolic link './vendor/bin/carbon': Operation not permitted
cp: cannot create symbolic link './vendor/bin/patch-type-declarations': Operation not permitted
cp: cannot create symbolic link './vendor/bin/php-parse': Operation not permitted
cp: cannot create symbolic link './vendor/bin/phpunit': Operation not permitted
cp: cannot create symbolic link './vendor/bin/psysh': Operation not permitted
cp: cannot create symbolic link './vendor/bin/sail': Operation not permitted
cp: cannot create symbolic link './vendor/bin/var-dump-server': Operation not permitted

```

Para solucionarlo, nos vamos a la carpeta del proyecto que se te habrá creado por defecto al hacer docker-compose; en este caso, y si no has modificado el archivo .yml, la carpeta del proyecto sera `my-proyect` y dentro de ella eliminamos la carpeta vendor.

Acto seguido instalar Composer de manera global en nuestro sistema Windows (bájate el instalador [desde este enlace](#)).

Una vez lo instales ya serás capaz de lanzar el comando composer desde cualquier consola de Windows.

## 3.1. VSCode extensiones

Recomendable instalar los siguientes plugins para Visual Studio Code.

Referentes a PHP:

- *PHP Intelephense*
- PHP IntelliSense
- *PHP Namespace Resolver*

Referentes a Laravel:

- Laravel Blade Snippets
- *Laravel Snippets*
- Laravel goto view
- *Laravel Extra Intellisense*

Referentes a CSS:

- *Tailwind CSS IntelliSense*

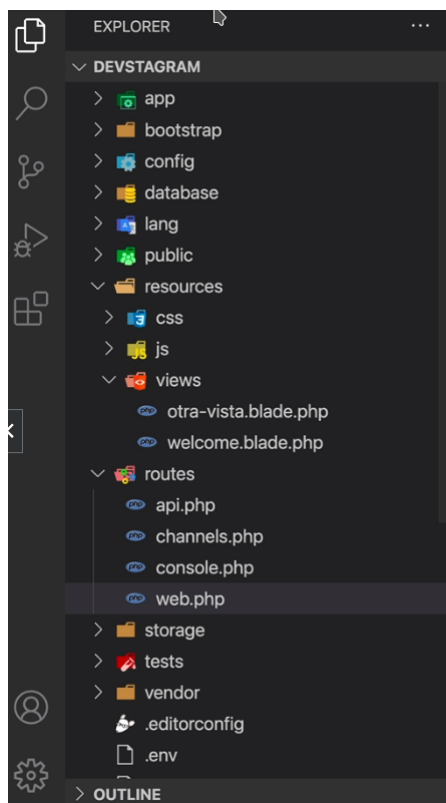
### Aporte

Un aporte, o instalación, a tener en cuenta, podría ser la de instalar `Tailwind CSS`. Este software nos va a proporcionar, de manera sencilla y cómoda, una opción de utilizar CSS.

Para ello, seguir las instrucciones del [anexo I - instalación de Tailwind CSS](#).

## 4. carpetas en Laravel

Al crear un nuevo proyecto con este framework, Laravel crea una serie de carpetas por defecto. Esta estructura de carpetas es la recomendada para utilizar Laravel.



### 4.1. app/Http/Controllers

En esta carpeta es donde se van a introducir nuestros controladores (cuando se cree un controlador, es aquí donde va a ubicarse).

### 4.2. Models

Para seguir el paradigma de **MVC**, los modelos van a introducirse en esta carpeta.

### 4.3. public

Esta es la carpeta más importante ya que es donde se ponen todos los archivos que el cliente va a mostrar al usuario cuando introduzcamos la URL de nuestro sitio web (por ejemplo, dentro de esta carpeta crear otra carpeta `img`). Normalmente se carga el archivo `index.php` por defecto.

### 4.4. resources

Esta es nuestra carpeta de recursos donde guardaremos los siguientes archivos, que también, están separados por sus carpetas... como cada nombre indica:

- `css` Archivos CSS (archivos originales que son procesados y se colocan las versiones compiladas en `public` mediante al archivo `webpack.mix.js`).
- `js` Archivos JS o JavaScript (archivos originales que son procesados y se colocan las versiones compiladas en `public` mediante al archivo `webpack.mix.js`).
- `lang` Archivos relacionados con el idioma del sitio (variables & strings).



- `views` Archivos de nuestras vistas, lo que las rutas cargan (comúnmente: lo que se ve en pantalla).

Podemos observar todo esto en el [ejemplo 01](#).

## 4.5. routes

---

Otra de las carpetas que más vamos a usar a lo largo de esta unidad dedicada a Laravel es `routes`. En ella se albergan todas las **rutas** (redirecciones web) de nuestro proyecto, pero más concretamente en el archivo `web.php`:

```
1 | Dada una ruta → se cargará una vista
```

## 4.6. vendor

---

En esta carpeta se colocan todas las dependencias de `Composer` que son necesarias tanto para que Laravel funcione o si quieres agregar alguna dependencia extra (por ejemplo: agregar pagos con *Paypal* . Laravel tiene un paquete llamado `cashier` que permite pagos en línea y también podemos instalar una dependencia extra mediante Composer).

Podemos observar en el archivo `composer.json` las dependencias instaladas.

## 4.7. .env

---

Aunque `.env` no es una carpeta, sino un archivo, también merece especial atención por ser un fichero de configuración de nuestro proyecto. Por ejemplo la conexión a base de datos:

```
1 | # ...
2 | DB_CONNECTION=mysql
3 | DB_HOST=127.0.0.1
4 | DB_PORT=3306
5 | DB_DATABASE=dwes
6 | DB_USERNAME=dwes
7 | DB_PASSWORD=dwes
8 | # ...
```

## 5. rutas

Las rutas en Laravel (y en casi cualquier Framework) sirven para redireccionar al cliente (o navegador) a las vistas que nosotros queramos.

Estas rutas se configuran en el archivo `routes/web.php` donde se define la ruta que el usuario pone en la URL después del dominio y se retorna la vista que se quiere cargar al introducir dicha dirección en el navegador.

En este ejemplo podemos observar la clase `Route` y un método estático `get`, este método estático toma la URL ('/' en este caso) y también un *closure* o *callback* `function () ...`

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

En el ejemplo de arriba vamos a cargar la vista llamada *welcome* que hace referencia a la vista `resources/views/welcome.blade.php`.

### closure VS controlador

Se puede definir en el segundo parámetro un closure o un controlador (veremos más adelante este caso).

Por ejemplo:

```
1 Route::get('/register', [RegisterController::class, 'index']) ->
  name('register');
```

### 5.1. alias

Es interesante darle un alias o un nombre a nuestras rutas para poder utilizar dichos alias en nuestras plantillas de Laravel que veremos más adelante.

Para ello, basta con utilizar la palabra `name` al final de la estructura de la ruta y darle un nombre que queramos; normalmente descriptivo y asociado a la vista que tiene que cargar el enrutador de Laravel.

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

Después veremos que es muy útil ya que a la hora de refactorizar o hacer un cambio, si tenemos enlaces o menús de navegación que apuntan a esta ruta, sólo tendríamos que cambiar el parámetro dentro del `get()` y no tener que ir archivo por archivo.

Laravel nos proporciona una manera más cómoda a la hora de cargar una vista si no queremos parámetros ni condiciones. Tan sólo definiremos la siguiente línea que hace referencia la ruta datos en la URL y va a cargar el archivo `usuarios.php` de nuestra carpeta views como le hemos indicado en el segundo parámetro.

```
1 <?php
2     /* http://localhost/datos/ */
3     Route::view('datos', 'usuarios');
```

Pero no sólo podemos retornar una vista, sino, desde un simple string, a módulos propios de Laravel.

Podemos observar todo esto en el [ejemplo 02](#).

## 5.2. parámetros

Ya hemos visto que con PHP podemos pasar parámetros a través de la URL, como si fueran variables, que las recuperábamos a través del método GET o POST.

Con Laravel también podemos introducir parámetros pero de una forma más vistosa y ordenada, de tal manera que sea visualmente más cómodo de recordar y de indexar por los motores de búsqueda como Google.

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

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

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

¿Qué pasa si no introducimos un id y sólo navegamos hasta cliente/ ? ... Nos va a devolver un `404 | NOT FOUND`.

Para resolver esto, podemos definir una ruta por defecto en caso de que el id (o parámetro) no sea pasado. Para ello usaremos el símbolo `?` en nuestro nombre de ruta e inicializaremos la variable con el valor que queramos.

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

Ahora tenemos otro problema, porque estamos filtrando por id del cliente que, normalmente es un número; pero si metemos un parámetro que no sea un número, vamos a obtener un resultado no deseado.

Para resolver este caso haremos uso de la cláusula `where` junto con una expresión regular numérica.

```

1 <?php
2     Route::get('cliente/{id?}', function($id = 1) {
3         return ('Cliente con el id: ' . $id);
4     }) -> where('id', '[0-9]+');

```

Además, podemos pasarle variables a nuestra URL para luego utilizarlas en nuestros archivos de plantillas o en archivos .php haciendo uso de un array asociativo. Veamos un ejemplo con la forma reducida para ahorrarnos código.

```

1 <?php
2     Route::view('datos', 'usuarios', ['id' => 5446]);

```

... y el archivo `resources/views/usuarios.php` debe tener algo parecido a esto:

```

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

```

Con las plantillas de Laravel **blade.php** veremos cómo simplificar aún más nuestro código.

Para más información acerca de las rutas, parámetros y expresiones regulares en las rutas puedes echar un vistazo a la [documentación oficial de rutas](#) que contiene numerosos ejemplos.

## 6. plantillas o templates

---

A través de las plantillas de Laravel vamos a escribir **menos código** PHP y vamos a tener nuestros archivos **mejor organizados**.

**Blade** es el sistema de plantillas que trae Laravel, por eso los archivos de plantillas que guardamos en el directorio de *views* llevan la extensión `blade.php`.

De esta manera sabemos inmediatamente que se trata de una plantilla de Laravel y que forma parte de una vista que se mostrará en el navegador.

### 6.1. directivas

---

Laravel tiene un gran número de directivas que podemos utilizar para ahorrarnos mucho código repetitivo entre otras funciones.

Digamos que las directivas son pequeñas funciones ya escritas que aceptan parámetros y que cada una de ellas hace una función diferente dentro de Laravel.

- `@yield` define el contenido dinámico que se va a cargar. Se usa conjuntamente con `@section`.
- `@section` y `@endsection` bloque de código dinámico.
- `@extends` importa el contenido de una plantilla ya creada.

### 6.2. separando código

---

Veamos sobre un ejemplo cómo separar el código para no repetirlo.

Podemos observar todo esto en el [ejemplo 03](#).

Realiza también el [ejemplo 04](#).

### 6.3. estructuras de control

---

Como en todo buen lenguaje de programación, en Laravel también tenemos estructuras de control.

En Blade (plantillas de Laravel) siempre que iniciemos un bloque de estructura de control **DEBEMOS** cerrarla.

- `@foreach` ~ `@endforeach` lo usamos para recorrer arrays.
- `@if` ~ `@endif` para comprobar condiciones lógicas.
- `@switch` ~ `@endswitch` en función del valor de una variable ejecutar un código.
- `@case` define la casuística del switch.
- `@break` rompe la ejecución del código en curso.
- `@default` si ninguna casuística se cumple.

```
1 <?php
2     $equipo = ['María', 'Alfredo', 'William', 'Verónica'];
3
4     @foreach ($equipo as $elemento)
5         <p> {{ $elemento }} </p>
6     @endforeach
```

Acordaros que podemos pasar variables a través de las rutas como si fueran parámetros. Pero en este caso, vamos a ver otra directiva más; el uso de `@compact`.

```
1 <?php
2     // Uso de @compact
3     $equipo = ['María', 'Alfredo', 'William', 'Verónica'];
4
5     // Route::view('nosotros', ['equipo' => 'equipo']);
6     Route::view('nosotros', @compact('equipo'));
```

## 7. controladores

Los controladores son el lugar perfecto para definir la **lógica de negocio** de nuestra aplicación o sitio web.

Hace de intermediario entre la *Vista* (lo que vemos con nuestro navegador o cliente) y el servidor donde la app está alojada.

Por defecto, los controladores se guardan en una carpeta específica situada en `app/Http/Controllers` y tienen extensión `.php`.

Para crear un controlador nuevo debemos hacer uso de nuestro querido CLI *artisan* donde le diremos que cree un controlador con el nombre que nosotros queramos.

Abrimos la consola y nos situamos en la raíz de nuestro proyecto:

```
1 | php artisan make:controller PagesController
```

### ABC

Si no podemos ejecutar la sentencia anterior, entonces:

```
1 | sudo docker-compose exec myapp php artisan make:controller PagesController
```

Si todo ha salido bien, recibiremos un mensaje por consola con que todo ha ido bien y podremos comprobar que, efectivamente se ha creado el archivo `PagesController.php` con una estructura básica de controlador, dentro de la carpeta `Controllers` que hemos descrito anteriormente.

Ahora podemos modificar nuestro archivo de rutas `web.php` para dejarlo limpio de lógica y trasladar ésta a nuestro nuevo controlador.

La idea de ésto es dejar el archivo `web.php` tan limpio como podamos para que, de un vistazo, se entienda todo perfectamente.

### recuerda

sólo movemos la lógica, mientras que las cláusulas como `where` y `name` las seguimos dejando en el archivo de rutas `web.php`.

### 7.1. convenciones

Laravel tiene una convención a la hora de nombrar los métodos de tus controllers conocida como Resource Controllers.

Esta convención ayuda bastante para tener todo mejor organizado:

verbo HTTP	URI	acción	ruta
GET	/clientes	index	clientes.index
POST	/clientes	store	clientes.store
DELETE	/clientes/{cliente}	destroy	clientes.destroy

En este enlace a la documentación de Laravel vemos las acciones que son controladas (o manejadas) por el controlador [enlace](#).

Veamos cómo quedaría un refactor del archivo de rutas utilizando un Controller como el que acabamos de crear.

Ahora nos quedaría de la siguiente manera:

```

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

y en nuestro archivo controlador lo dejaríamos de la siguiente manera:

```

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



```
22         'Maria',  
23         'Veronica'  
24     ];  
25  
26     return view('nosotros', @compact('equipo', 'nombre'));  
27 }  
28 }
```

Podemos observar todo esto en el [ejemplo 05](#).

## 8. tipos de Request

---

### 8.1. En HTTP y API's

---

En HTTP existen diferentes tipos de Request o tipos de Petición: GET, POST, PUT, PATCH y DELETE.

- **GET** es el más simple; cuando visitas un sitio web por default es un GET, y el método solo se utiliza para recuperar datos pero nunca debe enviar datos.
- **POST** se utiliza cuando mandas datos a un servidor; esto incluye información que llenas en un formulario o buscador.
- **PUT** es utilizado para actualizar un elemento; pero si no existe crea uno nuevo; PUT es un reemplazo total de un registro.
- **PATCH** es utilizado para actualizar parcialmente un elemento o recurso.
- **DELETE** se utiliza para eliminar un recurso o elemento.

Podemos observar todo esto en el [ejemplo 06](#).

## 9. anexo I - instalación de Tailwind CSS

Si hemos decidido instalar `Tailwind CSS` para que nos eche una mano con nuestro css, deberemos de seguir estos pasos:

1. Hay que comprobar la versión de npm y node:

```
1 | npm -v
2 | node -v
```

2. Si la versión de nodejs es inferior a la versión 14 (a la hora de crear este documento) **antes de seguir habremos de reinstalarlo**. Para ello habrá que ir al [anexo II - reinstalación de node](#) en este mismo documento.

**no continuar si la versión de node es inferior a 14.**

3. Si nuestra versión de nodejs es correcta (o hemos procedido a reinstalar node en el punto 2), lo primero será desinstalar bootstrap:

```
1 | npm uninstall bootstrap
```

4. Instalaremos en desarrollo estas tres dependencias:

```
1 | npm install -D tailwindcss postcss autoprefixer
```

5. Generamos ahora el fichero `tailwindcss.config.js` que aparecerá en la raíz del proyecto:

```
1 | npx tailwindcss init -p
```

6. Editar el fichero del proyecto Laravel `tailwindcss.config.js` que se ha generado en el directorio raíz del proyecto y donde indicaremos dónde vamos a utilizarlo:

```
1 | /** @type {import('tailwindcss').Config} */
2 | export default {
3 |   content: [
4 |     "./resources/**/*.blade.php",
5 |     "./resources/**/*.js",
6 |     "./resources/**/*.vue",
7 |   ],
8 |   theme: {
9 |     extend: {},
10 |   },
11 |   plugins: [],
12 | }
```

7. Ahora, en el fichero `/resources/css/app.css` agregar las siguientes líneas:

```
1 | @tailwind base;
2 | @tailwind components;
3 | @tailwind utilities;
```

8. Desde el terminal (y siempre dentro de nuestro proyecto), vamos a ejecutar:

```
1 | npm run dev
```

Si no funciona, prueba:

```
1 | npm run dev -- --host
```

```
o abc@abc-pc:~/Escritorio/IES/DWES/proyectos/my-project$ npm run dev
> dev
> vite
VITE v4.5.0 ready in 470 ms
→ Local:   http://localhost:5173/
→ Network: use --host to expose
→ press h to show help
LARAVEL v10.34.2 plugin v0.8.1
→ APP_URL: http://localhost
```

9. En el fichero `/resources/views/layouts/app.blade.php` hay que indicarle que va a utilizar el fichero `/resources/css/app.css`, para ello hay que añadirlo en:

```
1 | @vite('resources/css/app.css')
```

```
resources > views > layouts > app.blade.php
1 | <!DOCTYPE html>
2 | <html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
3 |   <head>
4 |     <meta charset="utf-8">
5 |     <meta name="viewport" content="width=device-width, initial-scale=1">
6 |
7 |     <title>Mi proyecto - @yield('titulo')</title>
8 |
9 |     @vite(resources/css/app.css)
10 |
11 |   </head>
12 |   <body>
13 |     <h1>@yield('titulo')</h1>
14 |     <hr>
15 |     @yield('contenido')
16 |   </body>
17 | </html>
18 |
```

A partir de ahora, y con este ejemplo, podemos observar que se nos muestra el css:

```

<body>
  <nav>
    <a href="/">Principal</a>
    <a href="/nosotros">Nosotros</a>
    <a href="/tienda">Tienda virtual</a>
  </nav>

  <h1 class="">@yield('titulo')</h1>
  <hr>
  @yield('contenido')
</body>
</html>

```

{} first-letter: &::first-letter  
 {} first-line:  
 {} marker:  
 {} selection:  
 {} file:  
 {} placeholder:  
 {} backdrop:  
 {} before:  
 {} after:  
 {} first:  
 {} last:  
 {} only:

## 10. anexo II - reinstalación de node

Para reinstalar nodejs:

1. Desinstalar node por completo:

```
1 | sudo apt-get purge --auto-remove nodejs
```

2. Eliminar todo resto de node y npm:

a. Antes que nada, debe ejecutar el siguiente comando desde el terminal:

```
1 | sudo rm -rf /usr/local/bin/npm /usr/local/share/man/man1/node*
   /usr/local/lib/dtrace/node.d ~/.npm ~/.node-gyp /opt/local/bin/node
   opt/local/include/node /opt/local/lib/node_modules
```

b. Eliminar los directorios node o node\_modules de /usr/local/lib con la ayuda del siguiente comando:

```
1 | sudo rm -rf /usr/local/lib/node*
```

c. Eliminar los directorios node o node\_modules de /usr/local/include con la ayuda del siguiente comando:

```
1 | sudo rm -rf /usr/local/include/node*
```

d. Eliminar cualquier archivo de nodo o directorio de /usr/local/bin con la ayuda del siguiente comando:

```
1 | sudo rm -rf /usr/local/bin/node
```

3. Instalar otra vez nvm:

a. Instalar NvM (Node Version Manager), desde el directorio de usuario `~`:

```
1 | curl -o- https://raw.githubusercontent.com/nvm-
   sh/nvm/v0.39.0/install.sh | bash
```

b. Actualiza el archivo .bashrc:

```
1 | source .bashrc
```

c. Confirma que el directorio local está configurado:

```
1 | echo $NVM_DIR
2 | /home/username/.nvm
```

4. Instalar node:

a. Revisar qué versiones de Node.js están disponibles:

```
1 | nvm ls-remote
```

b. Instalar la versión que desees (elige la v20.10.0):

```
1 | nvm install v20.10.0
```

5. Comprobar que la nueva versión de node es superior a 14:

```
1 | node -v
```

# 11. ejemplos

## 11.1. ejemplo 01. Hola Mundo

Vamos a eliminar todo el `style` que viene por defecto y a vaciar de contenido de la etiqueta `<body>` de la vista `resources/views/welcome.blade.php` y creamos etiqueta:

```
1 <h1>Página principal</h1>
2 <h2>Hola Mundo.</h2>
```

## 11.2. ejemplo 02. Otras vistas

Creamos un fichero `nosotros.blade.php`, `tienda.blade.php` en `views`.

Añadimos en `web.php`:

```
1 Route::get('/nosotros', function () {
2     return view('nosotros');
3 });
4 Route::get('/tienda-virtual', function () {
5     return view('tienda');
6 });
```

## 11.3. ejemplo 03. Uso de directivas

Cómo hacer uso del poder de Laravel para crear plantillas y no repetir código.

Supongamos que tenemos ciertas estructuras HTML repetidas como puede ser una cabecera *header*, un menú de navegación *nav* y un par de secciones que hacen uso de este mismo código.

Supongamos que tenemos 3 apartados en la web:

- Inicio
- Blog
- Fotos

1. Primero de todo tendremos que generar un archivo que haga de plantilla de nuestro sitio web.

Para ello creamos el archivo `app.blade.php` dentro del nuevo directorio de plantillas `resources/views/layouts`.

2. Dicho archivo va a contener el típico código de una página simple de HTML y en el body añadiremos nuestro contenido estático y dinámico.

```
1 <!DOCTYPE html>
2 <html lang="{ str_replace('_', '-', app()->getLocale()) }">
3 <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width, initial-
6         scale=1">
7     <title>Mi proyecto - @yield('titulo')</title>
```



```

8
9     <!-- introducir la siguiente linea para poder utilizar TailwindCSS
-->
10     @vite('resources/css/app.css')
11
12 </head>
13
14 <body class="bg-gray-100">
15     <header class="p-5 border-b bg-white shadow">
16         <div class="container mx-auto flex justify-between items-center">
17             <h1 class="text-3xl font-black">
18                 @yield('titulo')
19             </h1>
20
21             <nav class="flex gap-5 items-center">
22                 <a class="font-bold uppercase text-gray-600 text-sm"
23 href="#">Login</a>
24                 <a class="font-bold uppercase text-gray-600 text-sm"
25 href="#">Crear cuenta</a>
26             </nav>
27         </div>
28     </header>
29
30     <!-- BORRAR MÁS ADELANTE - este menú de navegación -->
31     <nav class="flex gap-5 items-center">
32         <a class="font-bold uppercase text-gray-600 text-sm"
33 href={{ route('inicio') }}>inicio</a> |
34         <a class="font-bold uppercase text-gray-600 text-sm"
35 href={{ route('noticias') }}>blogs</a> |
36         <a class="font-bold uppercase text-gray-600 text-sm"
37 href={{ route('fotos') }}>fotos</a>
38     </nav>
39     <hr>
40
41     <!-- CONTENIDO PRINCIPAL -->
42     <main class="container mx-auto mt-10">
43         <h2 class="font-black text-center text-3xl mb-10">
44             @yield('titulo')
45         </h2>
46         @yield('contenido')
47     </main>
48
49     <!-- FOOTER -->
50     <footer class="text-center p-5 text-gray-500 font-bold
51 uppercase">
52         MiPrimeraWeb - Todos los derechos reservados @php echo
53         date('Y') @endphp
54         <br>
55         <!-- con helpers -->
56         MiPrimeraWeb - Todos los derechos reservados {{ now()->year}}
57     </footer>
58 </body>
59 </html>

```

Cada sección que haga uso de esta plantilla contendrá un menú de navegación con enlaces a cada una de las secciones y el contenido dinámico de cada sección.

3. Ahora crearemos los archivos dinámicos de cada una de las secciones, en nuestro caso:

`inicio.blade.php`:

Importamos el contenido de plantilla bajo la directiva `@extends` para que cargue los elementos estáticos que hemos declarado y con la directiva `@section` y `@endsection` definimos el bloque de código dinámico, en función de la sección que estemos visitando.

```

1  <?php
2      // inicio.blade.php
3      @extends('layouts.app')
4
5      @section('titulo')
6          página principal
7      @endsection
8
9      @section('contenido')
10         contenido de la página principal
11     @endsection

```

`blog.blade.php`:

```

1  <?php
2      // blog.blade.php
3      @extends('layouts.app')
4
5      @section('titulo')
6          noticias
7      @endsection
8
9      @section('contenido')
10         contenido de todas las noticias
11     @endsection

```

`fotos.blade.php`:

```

1  <?php
2      // fotos.blade.html
3      @extends('layouts.app')
4
5      @section('titulo')
6          fotografías
7      @endsection
8
9      @section('contenido')
10         galería de fotografías
11     @endsection

```

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

```

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

```

De esta manera podremos hacer uso del menú de navegación que hemos puesto en nuestra plantilla y gracias a los alias noticias y galeria, la URL será más amigable.

## 11.4. ejemplo 04. Vista registrarse

Antes de continuar con el ejemplo, debes **eliminar**:

- el menú de navegación de *inicio* | *blogs* | *fotos* del fichero `layouts/app.blade.php`.
- las rutas en el fichero `web.php` que afecten a este menú (exceptuando la ruta de inicio).
- las dos vistas `blog.blade.php` y `foto.blade.php`.

Este borrado se debe a que NO vamos a continuar con estas vistas; solo eran un ejemplo de uso de directivas y la separación de código.

Como has observado, nuestro anterior ejemplo contenía en el fichero `resources/views/layouts/app.php` un menú de navegación en el que se indicaba *Login* y *Crear cuenta*.

Crea las vistas para estos dos enlaces. Para ello alojarás sus dos vistas en una carpeta nueva `resources/views/auth` con nombre `register.blade.php` para *Crear cuenta*.

```

1 <?php
2 // fotos.blade.html
3 @extends('layouts.app')
4
5 @section('titulo')
6     Regístrate en tu APP
7 @endsection
8
9 @section('contenido')
10
11 @endsection

```

Añadir al fichero `web.php` la entrada:

```

1 Route::get('/crear-cuenta', function(){
2     return view(auth.register) //la ruta contiene .
3 }) -> name('resgister');

```

## 11.5. ejemplo 05. controlador RegisterController y su formulario

En vez de usar *closures* o *callbacks* en el fichero `web.php` vamos a crear controladores en nuestro ejemplo y trasladar la lógica de negocio a estos últimos. Así, y siguiendo con los ejemplos anteriores, vamos a crear un controlador para gestionar el registro en nuestra APP (vamos a ordenar nuestros ficheros y colocaremos `RegisterController` dentro de la carpeta `Auth` y

añadimos doble diagonal inversa `\\`).

Abrimos el CLI artisan (nuevo terminal en VS Code o `Ctrl+``) en la carpeta de nuestro proyecto y ejecutamos:

```
1 sudo docker-compose exec myapp php artisan make:controller
Auth\\RegisterController
```

En `web.php` modificar la ruta de *Crear cuenta* (quitar el *callback* y añadir el controlador y su método). Se recomienda como convención el nombre del método `index`:

```
1 Route::get('/crear-cuenta', [RegisterController::class, 'index']) ->
name('resgister');
```

Y en `RegisterController.php`, trasladamos la lógica de negocio que ejecutaba el *callback*, añadiendo la palabras reservadas `public` y añadiendo también un nombre a esta función `index()`:

```
1 public function index() {
2     return view('auth.register');
3 }
```

Para la vista de register `register.blade.php` vamos a introducir el código:

```
1 @extends('layouts.app')
2
3 @section('titulo')
4     Regístrate en la APP
5 @endsection
6
7 @section('contenido')
8     <div class="md:flex md:justify-center md:gap-10 md:items-center">
9         <div class="md:w-6/12 p-5">
10             
11         </div>
12
13         <div class="md:w-4/12 bg-white p-6 rounded-lg shadow-xl">
14             <form action="">
15                 <div>
16                     <label for="name" class="mb-2 block uppercase text-gray-500 font-
bold">
17                         Nombre
18                     </label>
19                     <input
20                         id="name"
21                         name="name"
22                         type="text"
23                         placeholder="tu nombre"
24                         class="border p-3 w-full rounded-lg"
25                     />
26                 </div>
27
```

```

28         <div>
29             <label for="username" class="mb-2 block uppercase text-gray-500
font-bold">
30                 Nombre
31             </label>
32             <input
33                 id="username"
34                 name="username"
35                 type="text"
36                 placeholder="tu nombre de usuario"
37                 class="border p-3 w-full rounded-lg"
38             />
39         </div>
40
41         <div>
42             <label for="email" class="mb-2 block uppercase text-gray-500 font-
bold">
43                 Email
44             </label>
45             <input
46                 id="email"
47                 name="email"
48                 type="text"
49                 placeholder="tu email de registro"
50                 class="border p-3 w-full rounded-lg"
51             />
52         </div>
53
54         <div>
55             <label for="password" class="mb-2 block uppercase text-gray-500
font-bold">
56                 Password
57             </label>
58             <input
59                 id="password"
60                 name="password"
61                 type="password"
62                 placeholder="tu contraseña de registro"
63                 class="border p-3 w-full rounded-lg"
64             />
65         </div>
66
67         <div>
68             <label for="password_confirmation" class="mb-2 block uppercase
text-gray-500 font-bold">
69                 Repetir password
70             </label>
71             <input
72                 id="password_confirmation"
73                 name="password_confirmation"
74                 type="password"
75                 placeholder="repite la contraseña"
76                 class="border p-3 w-full rounded-lg"
77             />
78         </div>
79         <br>

```

```

80         <input
81             type="submit"
82             value="Crear cuenta"
83             class="bg-sky-600 hover:bg-sky-700 transition-colors
84             cursor-pointer uppercase font-bold w-full p-3 text-white
rounded-lg"
85         />
86     </form>
87 </div>
88 </div>
89 @endsection

```

## 11.6. ejemplo 06. petición post

Vamos a crear ahora el enlace registrarse. Para ello accedemos a `web.php` e introducimos la línea con el método POST:

```

1 Route::get('/crear-cuenta', [RegisterController::class, 'index']) ->
    name('register');
2 Route::post('/crear-cuenta' [RegisterController::class, 'store'])

```

**No** ponemos un alias en esta ruta porque va a tomar también el alias del anterior ruta.

Al mismo tiempo, en nuestro controlador `RegisterController.php` agregamos la función `store` :

```

1 public function store() {
2     dd('formulario...');
3 }

```

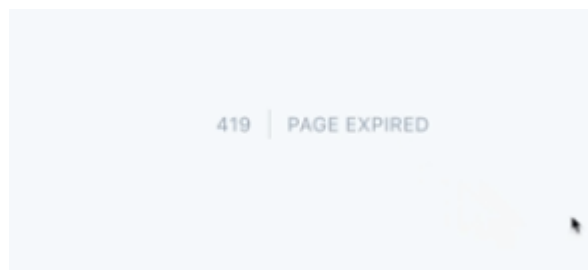
Para terminar este punto, en el fichero `.blade` `register.blade.php` vamos a modificar la etiqueta `form` para que redirija al action correspondiente con el método en cuestión. Fíjate que en action ponemos la función `route` y el nombre de la ruta:

```

1 ...
2 <form action="{{route('register')}}" method="POST">
3 ...

```

Si recargamos la página y accionamos el enlace vemos que nos muestra el siguiente error:



¿Qué es página expirada?

Laravel es un framework enfocado a la seguridad (en este caso, se asegura que no suframos ataques del tipo XSRF o Cross Site Request Forgery). Así que Laravel tiene consideraciones de seguridad.

Para evitar estos ataques usaremos la directiva `@csrf` justo después de la línea de la etiqueta

```
1 ...
2 <form action="{{route('register')}}" method="POST">
3     @csrf
4 ...
```

Si pulsamos F12 para ver el código se mostrará un campo oculto con un token para validar la cadena y evitar este tipo de ataques:

```
<div class="md:w-4/12 bg-white p-6 rounded-lg shadow-xl">
  <form action="/crear-cuenta" method="POST">
    <input type="hidden" name="_token" value="IArNWYpdGJLCoLw53eD2CGaEhfe8QriFRul3nOng">
    <label for="name" class="mb-2 block uppercase text-gray-500 font-bold">
      Nombre
    </label>
```

Modificamos la función `store` para pasarle la clase Request:

```
1 public function store(Request $request) {
2     dd(request);
3     // dd(request->get('email'));
4 }
```

Si accedemos a localhost/register:

```
lluminate\Http\Request {#37 ▼ // app/Http/Controllers/RegisterController.php:18
+attributes: Symfony...\ParameterBag {#42 ▶}
+request: Symfony...\InputBag {#38 ▼
  #parameters: array:6 [▼
    "_token" => "LThnoGG85KAmg3Pwgm80Z2Ee21XkQSaHC1LtD3DB"
    "name" => "arturo"
    "username" => "arturo76"
    "email" => null
    "password" => null
    "password_confirmation" => null
  ]
}
+query: Symfony...\InputBag {#45 ▶}
+server: Symfony...\ServerBag {#40 ▶}
+files: Symfony...\FileBag {#44 ▶}
+cookies: Symfony...\InputBag {#43 ▶}
+headers: Symfony...\HeaderBag {#39 ▶}
#content: null
#languages: null
#charset: null
#encodings: null
#acceptableContentTypes: null
#pathInfo: "/register"
#requestUri: "/register"
```

## 12. referencias

---

- [Tutorial de Composer](#)
- [Web Scraping with PHP – How to Crawl Web Pages Using Open Source Tools](#)
- [PHP Monolog](#)
- [Unit Testing con PHPUnit — Parte 1.](#), de Emiliano Zublena.