

Introducción a los Frameworks en PHP.

DESARROLLO WEB EN ENTORNO SERVIDOR

GARRIDO PORTES, GUILLERMO



Atribución-NoComercial-SinDerivadas
4.0 Internacional (CC BY-NC-ND 4.0)

Introducción a los Frameworks en PHP.

Introducción a los Frameworks en PHP.....	1
1. Patrón MVC.....	2
2. Frameworks.....	3
Introducción.....	3
Laravel	3
Estructura de un proyecto en Laravel	4
Ejecutar un proyecto en Laravel.....	6
3. Rutas.....	7
Rutas simples.....	8
Rutas con nombre o alias	8
Rutas con variables.....	9
Paso de variables a las vistas desde la ruta.....	9
4. Vistas	10
Motor de plantillas Blade.....	10
Reutilizando Código	13
5. Controladores.....	18
Tipos de controladores.....	18

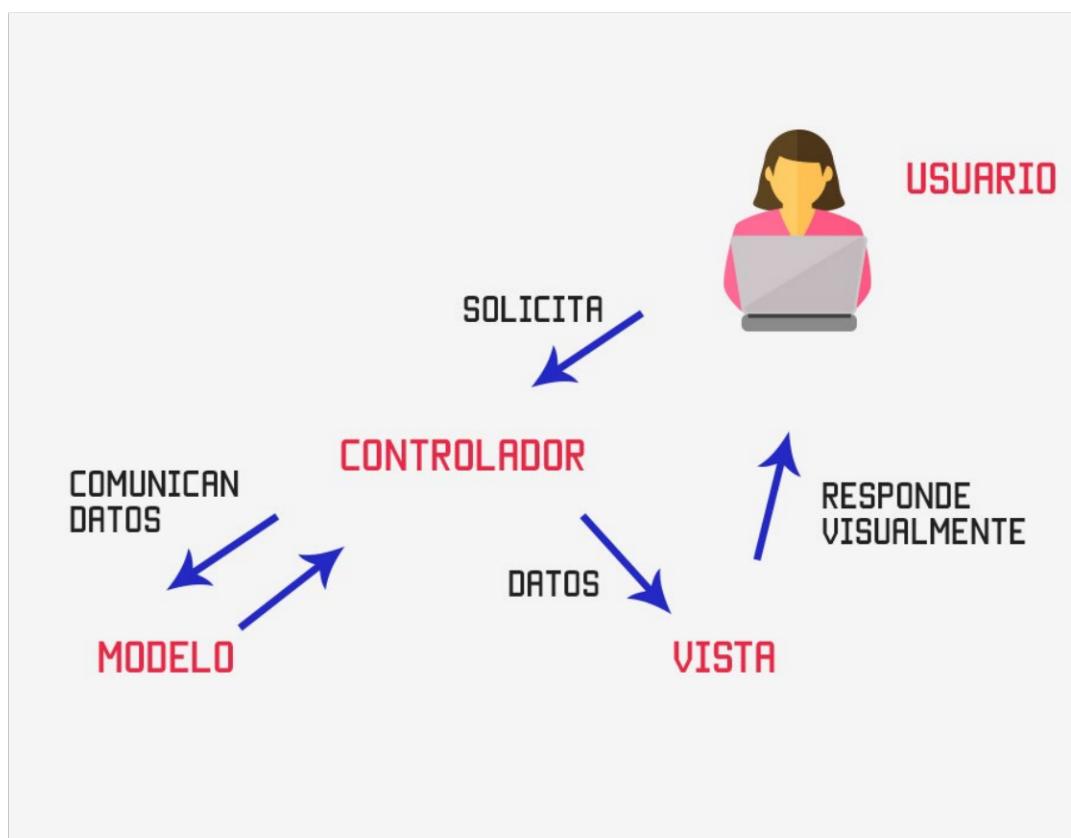
1. Patrón MVC.

Introducción

MVC, parte de las iniciales de Modelo-Vista-Controlador (Model-View-Controller, en inglés), que son las capas o grupos de componentes en los que se organizan las aplicaciones bajo este paradigma.

La arquitectura MVC propone la separación de los componentes de una aplicación en tres grupos (o capas) principales:

- **Modelo:** estará formado por las clases o elementos donde guardaremos los datos, es decir, aquellas entidades que servirán para acceder y gestionar la información del sistema que se está desarrollando. La mayoría de los frameworks disponen de un ORM que es una forma de mapear los datos que se encuentran en la base de datos almacenados en un lenguaje de script SQL a objetos de PHP y viceversa, mediante el cual se realizan las diferentes interacciones con la base de datos, simplificando mucho las operaciones.
- **Vista:** es la presentación final que verá el usuario, del estado del Modelo en un momento concreto y en el contexto de una acción determinada. Por tanto, la Vista genera la interfaz partiendo de los datos que le suministre el controlador.
- **Controlador:** actúa como intermediarios entre el usuario y el sistema. Gestiona las peticiones del usuario recuperando datos desde los modelos, y los prepara para la presentación de la vista al usuario.



Ventajas y desventajas de patrón MVC

El uso del patrón MVC ofrece múltiples ventajas al desarrollador, algunas de ellas son:

- La separación del código del patrón MVC hace que cada parte tenga su propia responsabilidad y permite que las aplicaciones sean más limpias, simples, fácilmente mantenibles y más robustos.
- Mayor velocidad de desarrollo de aplicaciones grandes ya que, al estar separado en tres partes cada desarrollador puede ocuparse de cada parte en al mismo tiempo.
- El controlador puede manejar múltiples vistas asegurando consistencia entre ellas.
- Facilidad para realización de pruebas unitarias.

Por el contra, también existen ciertas desventajas, entre las que cabe destacar:

- Hay que seguir los estándares y la arquitectura y limitaciones del patrón.
- La división impuesta por el patrón MVC obliga a mantener un mayor número de ficheros.
- Curva de aprendizaje requerirá cierto esfuerzo.

2. Frameworks.

Introducción

Un framework proporciona un conjunto de bibliotecas de código que contienen módulos preprogramados que permiten al usuario construir aplicaciones más rápidamente. Estos ahoran mucho tiempo en la codificación del proyecto y ofrecen al desarrollador una forma más guiada de estructurar los proyectos, además de una serie de herramientas que facilitan, entre otras cosas, la definición de las distintas vistas o páginas de la aplicación, o la conexión y acceso a diferentes fuentes de datos.

Existe una gran cantidad de frameworks PHP que se usan en la actualidad como Laravel, Symfony, Codeigniter, CakePHP, Zend entre otros.

Laravel

Laravel es principalmente un marco de desarrollo de PHP para backend, aunque ofrece algunas funcionalidades de frontend. Este utiliza un lenguaje de scripting en lugar de utilizar PHP estricto por lo que tiene varias diferencias notables, principalmente en la facilidad de uso y la velocidad de ejecución.

Es altamente escalable y cuenta con soporte integrado para sistemas de caché rápidos y distribuidos, ofreciendo un entorno de desarrollo altamente funcional, que sigue el patrón MVC y cuenta con el ORM Eloquent para simplificar el acceso y la manipulación de datos.

También cuenta con una gran comunidad por lo que la biblioteca de aplicaciones y paquetes disponibles tanto oficiales como de terceros es considerable y están fácilmente disponibles utilizando Composer.

Instalación y configuración

Antes de proceder a la creación de un proyecto Laravel, hay que situarse en la carpeta del ordenador donde queremos instalar Laravel.

Una vez situados en el directorio de proyecto, para instalar Laravel se puede utilizar Composer, con un simple comando:

- "composer" que es el gestor de paquetes que se encarga de instalar Laravel en el nuevo proyecto.
- "create-project" es el subcomando de composer para crear un nuevo proyecto.
- "laravel/laravel" es el nombre del proyecto de base que se va a usar para este nuevo proyecto.
- "proyecto-ggarrido-laravel" es el nombre del proyecto.
- Se podría añadir detrás del nombre del proyecto, la versión de Laravel a utilizar. En su ausencia utilizará la última versión.

```
1 composer create-project laravel/laravel proyecto-ggarrido-laravel
```

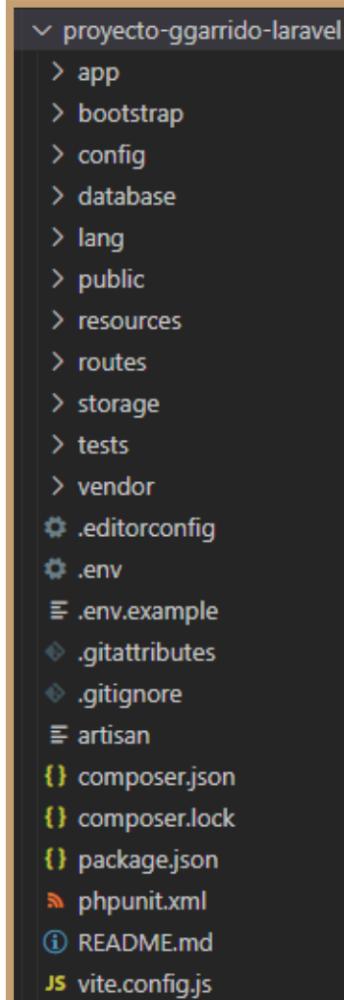
Cuando creamos un proyecto en Laravel automáticamente se instala la herramienta artisan en la raíz del proyecto. Esta permite gestionar los proyectos Laravel crear modelos, migraciones, controladores, etcétera mediante la línea de comandos. Además, nos proporciona un servidor web interno de modo que no es necesario contar con un servidor web adicional como Apache o Nginx.

Por otro lado, para poder cargar librerías externas desde **package.json** que se utilizarán en el frontend de la aplicación (la parte cliente) es necesario instalar **Node.js**. Este permite utilizar la orden **npm** que hace referencia a (Node Packacge Manager) y que permite gestionar, igual que hace composer para las librerías PHP, las librerías JavaScript o CSS como por ejemplo Bootstrap que es ampliamente utilizado.

Estructura de un proyecto en Laravel

Al crear un nuevo proyecto de Laravel se nos generará una estructura de carpetas y ficheros para organizar el código.

En la carpeta raíz se pueden encontrar tres ficheros de configuración que dependen del equipo en que se encuentran. Estos pueden cambiar según si el equipo está en un equipo de un desarrollador o si está en un servidor de producción:



.env contiene las variables globales de configuración. De modo cambiando este fichero se puede cambiar desde la base de datos donde se accede hasta los parámetros de seguridad de la aplicación. Este fichero contiene unas variables predefinidas que hay que completar para la correcta configuración de la base de datos:

- APP_NAME = Nombre del proyecto
- DB_CONNECTION = Tipo de conexión (mysql / mongodb)
- DB_HOST = IP del servidor (127.0.0.1)
- DB_PORT = Número de puerto (3306 / 27017)
- DB_DATABASE = Nombre de la base de datos
- DB_USERNAME = Nombre del usuario de la base de datos
- DB_PASSWORD = Contraseña del usuario de base de datos
- composer.json contiene la información en formato JSON que utiliza Composer para la instalación de Laravel o de otras librerías externas así como las dependencias de la versión de PHP. Estas se pueden generar utilizando el comando **composer install**.
- package.json contiene las dependencias JavaScript (para el frontend) que tiene la aplicación. Estas se pueden regenerar utilizando el comando **npm install**.

Además de los ficheros anteriores también se genera el siguiente árbol de directorios:

- **app**: contiene el código fuente de la aplicación. Esta se compone por diferentes directorios muy importantes:
 - **Console** contiene los comandos creados para el desarrollo de la app.
 - **Exceptions** que contiene las excepciones del framework y donde se almacenan las excepciones personalizadas.
 - **HTTP** contiene los **Controladores** y los **Middleware**, así como el fichero Kernel.php que se utiliza para registrarlos.
 - **Models** contiene las clases o modelos necesarios para interactuar con la base de datos.
 - **Providers** que contiene los servicios que va a tener la aplicación.

En app se crearán más directorios según se implemente la aplicación, como las plantillas de los emails.

- **bootstrap**: contiene el fichero app.php que se encarga de arrancar el framework. También contiene una carpeta llamada **cache** que utiliza para almacenar algunos

ficheros ya generados por el framework y así acelerar y optimizar el rendimiento de Laravel.

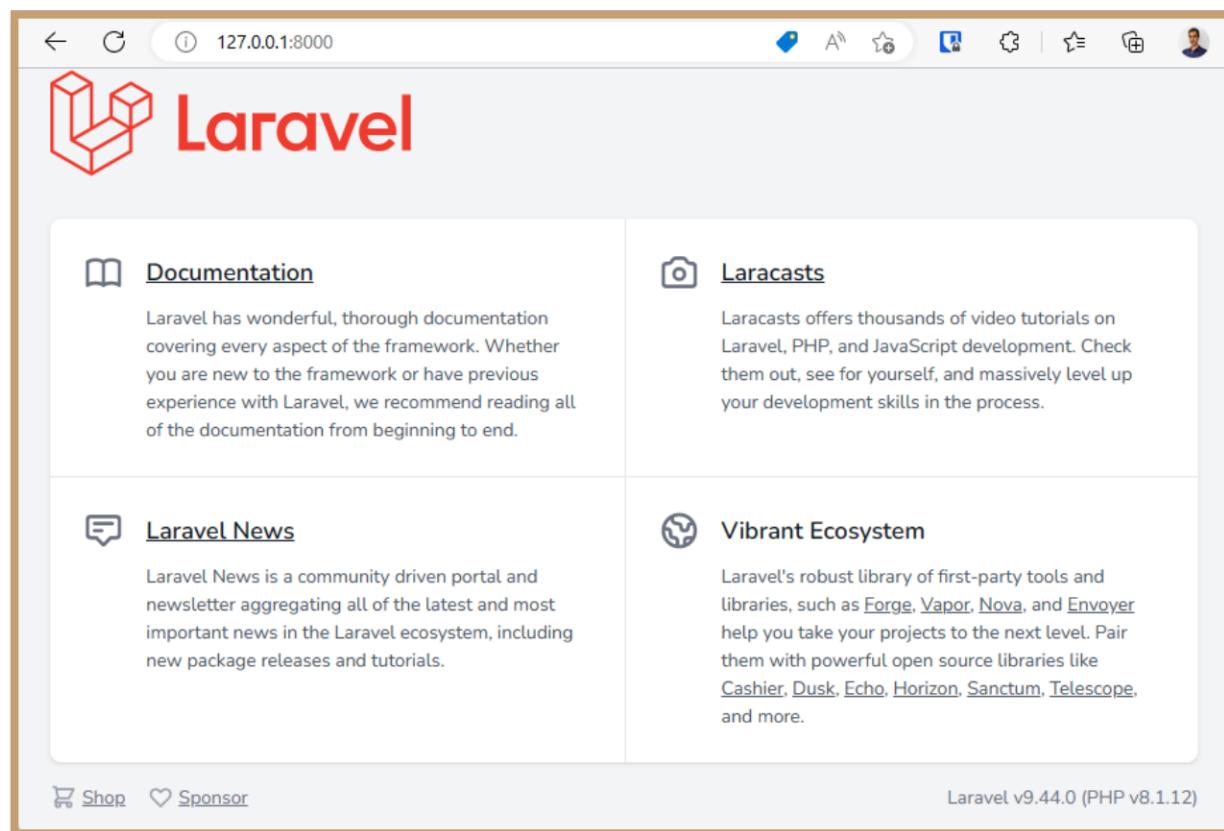
- **config:** contiene la configuración para el framework y para la aplicación.
- **database:** contiene los ficheros relacionados con el manejo de la base de datos. En este directorio se encuentran los subdirectorios **factories**, **migrations** y **seeds**.
- **lang:** contiene los ficheros que contienen los diferentes textos estáticos de la aplicación y con los cuales se puede cambiar de un idioma a otro. Normalmente habrá una carpeta por cada idioma, como por ejemplo “en” (idioma inglés) o “es” (idioma español)
- **public:** contiene los recursos estáticos y accesibles de manera pública de la aplicación, como imágenes y fuentes. También contiene el fichero “index.php” que es llamado cuando se ejecuta la aplicación e inicia el proceso del framework.
- **resources:** contiene los recursos que necesitará la aplicación. Estos se encuentran dentro de los subdirectorios assets, css, js y views.
 - **css y js:** contiene los archivos de estilos y JavaScript de la aplicación.
 - **assets:** contiene los ficheros necesarios para los precompiladores como less o sass.
 - **views:** contiene las vistas Blade de la aplicación.
- **routes:** contiene los ficheros encargados de gestionar las rutas de la aplicación.
- **storage:** contiene el directorio **app** con la información generada por la aplicación y dentro del directorio **framework** que contiene las sesiones, cache, tests, y vistas generadas y el directorio **logs** que contiene los logs que produce la aplicación.
- **tests:** contiene los tests unitarios realizados con PHPUnit donde se prueban las diferentes funciones de la aplicación.
- **vendor:** contiene todas las librerías y dependencias que se generan a partir del fichero composer.json.

Ejecutar un proyecto en Laravel

Para ejecutar el servidor web interno que tiene Laravel, simplemente hay que hacer uso de el comando “artisan” donde le indicaremos que ejecute el servidor. De este modo se iniciará el servidor atendiendo al puerto 8000.

```
proyecto-ggarrido-laravel> php artisan serve
INFO  Server running on [http://127.0.0.1:8000].
Press Ctrl+C to stop the server
```

Posteriormente solo será necesario utilizar un cliente web y utilizar la URL del servidor seguida del puerto 8000.



3. Rutas

Las rutas en Laravel controlan el flujo de las solicitudes HTTP del cliente y hacia este como GET, POST, PATCH, PUT, DELETE, OPTIONS.

Estas están situadas dentro de la **routes**, en ella se encuentran dos ficheros que se encargan de contener las diferentes rutas de la aplicación. El primero es **web.php**, que almacena las rutas de la aplicación que se enlazan con las vistas y el segundo es el fichero **api.php**, el cual contiene las rutas que definen servicios API REST, y que no cuentan con sesiones o cookies.

Las rutas se definen utilizando la palabra reservada **Route::** y a continuación indicando el método que hace referencia al tipo de solicitud HTTP utilizado como **get()**, **post()**, **put()**, etcétera.

También es posible que una ruta se encargue de gestionar múltiples tipos de solicitudes HTTP. Para ello se debe llamar a la función **match()** pasándole como primer parámetro un array de las solicitudes HTTP que va a gestionar.

En caso de querer que una misma ruta gestione todas las solicitudes HTTP a una dirección concreta, se puede hacer uso de la función **any()**.

```

1 Route::get($url, function(){});
2 Route::post($url, function(){});
3 Route::put($url, function(){});
4 Route::patch($url, function(){});
5 Route::delete($url, function(){});
6 Route::options($url, function(){});
7 Route::match(['get', 'post'], $url, function(){});
8 Route::any($uri, function(){});

```

Rutas simples

Son las rutas que asocian el nombre de una ruta con una función.

En el fichero `web.php` contiene una ruta simple predefinida, que utiliza el método `get()` al que se llamará cuando se acceda a la raíz de la aplicación `/`. En este contiene una función que devuelve el resultado a una llamada del método `view()`.

El método `view()` es el encargado de cargar una vista a partir de la ruta y nombre que se le pase como parámetro.

De este modo el ejemplo llama a la ruta `/` que cargará la vista `welcome.blade.php`.

```

1 Route::get('/', function(){
2     return view('welcome');
3 });

```



IMPORTANTE

Si no es necesario ejecutar ninguna operación dentro de la función antes de llamar al método `view()` y este va a ser una solicitud HTTP de tipo GET, es posible realizar la llamada directamente utilizando la función `view()`:

```
1 Route::view('/', 'welcome');
```

Rutas con nombre o alias

Cuando una ruta se llama desde algún enlace o botón de la aplicación, esta busca en el fichero de rutas que ruta es la que debe atender la solicitud.

En ocasiones es posible que las rutas cambien, obligando a tener que revisar todas las referencias que existan en HTML o PHP y modificando en ellas el nombre de la ruta cambiado.

Para evitar esto, es posible a cada ruta generarle un nombre simplemente añadiendo detrás de la ruta la relación flecha con la palabra "name" como el siguiente ejemplo:

```
1 Route::view('sobre-nosotros', 'about')->name('about');
```

Rutas con variables

En las llamadas HTTP, es posible hacer uso de variables para poder transmitir información. Laravel permite realizar esto desde las rutas de la aplicación simplemente incluyendo el nombre de la variable entre llaves "{}". Además, si la ruta cuenta con la función como segundo parámetro, habrá que pasarle a dicha función las variables como argumentos de entrada.

```
1 Route::get('/{usuario}', function($usuario){
2     return view('welcome');
3 });
```

Es posible definir el parámetro como opcional. Para ello se añadirá en la ruta el símbolo "?" detrás del nombre de la variable. Además, para evitar posibles errores, el parámetro de entrada de la función interna deberá tener un valor predeterminado.

```
1 Route::get('/{usuario?}', function($usuario = "Guillermo Garrido"){
2     return view('welcome');
3 });
```

Validaciones para las variables

Se pueden validar las variables que se pasan a la ruta haciendo uso de la función **where()** donde se pasará el nombre de la variable a validar como primer parámetro y la expresión regular o patrón que debe seguir como segundo parámetro.

Por ejemplo, para validar que la variable que se pasa tiene un valor de tipo cadena con únicamente letras mayúsculas y minúsculas se añadirá el siguiente la llamada al **where()**.

```
1 Route::get('/{usuario?}', function($usuario = "Guillermo Garrido"){
2     return view('welcome');
3 })->where('usuario', "[A-Za-z]+") ;
```

Paso de variables a las vistas desde la ruta

Del mismo modo que con las rutas, también es posible pasar variables a las vistas.

Para ello es posible hacer uso de la función **with()**, después de la llamada a la función **view()**, donde recibirá un array asociativo con los valores para la vista.

```
1 Route::get('/{usuario}', function($usuario){
2     return view('welcome')->with(['usuario' => $usuario]);
3 });
```

O bien es posible introducir el array como segundo parámetro de la función **view()**. Este caso también acepta la posibilidad de utilizar la función **compact()**, esta crea un array asociativo partiendo del nombre de las variables que se le indique.

La ventaja de utilizar la función **compact()** es no tener que añadir en cada elemento del array la clave y el valor, sino que, con simplemente indicar la clave y si esta coincide con una variable ya declarada en la función, el valor de la variable se asignara automáticamente.

```
1 Route::get('/{usuario}', function ($usuario) {  
2     return view('welcome', compact('usuario'));  
3 });
```



IMPORTANTE

En el caso de utilizar la ruta directamente con la función **view()** el array de variables se pasará como tercer parámetro de la función.

```
1 Route::view('/{usuario}', 'welcome', compact('usuario'));
```

4. Vistas

Las vistas son la capa más superficial de la aplicación. En ellas se muestra la interfaz al usuario, de modo que pueda interactuar con la aplicación.

Las vistas en Laravel pueden contener código HTML, CSS y PHP. El PHP que este en la vista, será aquel que únicamente sirva para mostrar, ocultar o filtrar los elementos y el contenido de la interfaz.

Para cargar las vistas, como se ha visto anteriormente, es necesario hacer uso del método **view()**. Este método buscará la ruta o el archivo dentro de la carpeta **resources->views**.

Dentro de views (y sus subcarpetas) Laravel aceptara los archivos cuya extensión sea **".php"** o bien **".blade.php"**. Esta última, proviene del motor de plantillas de Laravel llamado Blade.

Motor de plantillas Blade.

Una Plantilla Web es una herramienta empleada para separar el contenido, de la aplicación, del diseño de esta. Estas permiten que la misma interfaz pueda ser utilizada en diferentes partes de la aplicación.

En este caso, Laravel cuenta con un motor de plantillas que permite simplificar el código PHP de la aplicación.

Mientras que en PHP es necesario indicar que empieza un bloque de dicho lenguaje y realizar las operaciones o el echo para mostrar e imprimir un contenido, Blade ofrece la posibilidad de utilizar los símbolos llaves dobles “{{ }}” y dentro de estos indicar la variable a imprimir.

```

1 <!-- Estructura PHP nativo -->
2 <h1>Bienvenido/da <?php echo $usuario; ?></h1>
3 <!-- Estructura Blade -->
4 <h1>Bienvenido/da {{ $usuario }}</h1>
```

La diferencia fundamental es que Laravel incluye con sus etiquetas de doble llave una llamada al método **e()** que realiza una llamada de forma automática a la función **htmlspecialchars()** del contenido mejorando la seguridad y evitando la inyección de código.

Para evitar esta llamada automática, se puede añadir la doble exclamación “{!! !!}” de modo que no realice la comprobación.

Para añadir comentarios en Blade se deberá usar la sintaxis de dobles llaves dobles guiones “{{-- --}}”.

Dentro de las dobles llaves también se pueden llamar funciones de Laravel. Por ejemplo, una de las más importantes es la llamada desde una vista a una ruta de Laravel. Para ello, simplemente hay que hacer uso de la función **route()** dentro de las dobles llaves indicando como primer parámetro de la función el nombre de la ruta que se desea llamar y como segundo parámetro un array asociativo de las variables que se van a enviar.

```
1 <a href="{{ route('/', [ 'usuario' => "Guillermo Garrido" ]) }}></a>
```

Por otro lado, para llamar a las diferentes directivas de Laravel, ya sea un “include” o una estructura de control de flujo, es necesario utilizar el símbolo arroba “@”.

ESTRUCTURAS DE CONTROL DE FLUJO EN BLADE

De igual modo que PHP permite simplificar el código con el uso de sintaxis alternativa, como se vio en el tema 2. Blade permite crear bloques de código utilizando el símbolo arroba “@” delante de la estructura de control.

Por tanto, para crear estructuras de control if elseif la sintaxis sería la siguiente:

Sintaxis PHP alternativa

```

1 <html>
2   <body>
3     <?php if ($hora == 8): ?>
4       <h2>Es hora del desayuno.</h2>
5     <?php elseif ($hora == 14): ?>
6       <h2>Es hora de la comida.</h2>
7     <?php elseif ($hora == 21): ?>
8       <h2>Es hora de la cena.</h2>
9     <?php else: ?>
10      <h2>Ahora no toca comer.</h2>
11    <?php endif; ?>
12  </body>
13 </html>
```

Sintaxis Blade

```

1 <html>
2   <body>
3     @if ($hora == 8)
4       <h2>Es hora del desayuno.</h2>
5     @elseif ($hora == 14)
6       <h2>Es hora de la comida.</h2>
7     @elseif ($hora == 21)
8       <h2>Es hora de la cena.</h2>
9     @else
10      <h2>Ahora no toca comer.</h2>
11    @endif
12  </body>
13 </html>
```

Del mismo modo, las estructuras repetitivas también se forman utilizando la @ seguida de la estructura repetitiva deseada.

En este caso Laravel proporciona las típicas estructuras de PHP como for, while o foreach.

```

1 <ul>
2   @for($i = 0; $i < count($usuarios); $i++)
3     <li>{{ $usuarios[$i] }}</li>
4   @endfor
5
6   $i=0;
7   @while($i < count($usuarios))
8     <li>{{ $usuarios[$i] }}</li>
9   @endwhile
10
11  @foreach($usuarios as $usuario)
12    <li>{{ $usuario }}</li>
13  @endforeach
14 </ul>
```

En la mayoría de las ocasiones, los diseñadores deben crear casos especiales comprobando si la variable está definida con estructuras de control como el siguiente caso:

```

1 <ul>
2 ...
3   @if(isset($usuarios))
4     @foreach($usuarios as $usuario)
5       <li>{{ $usuario }}</li>
6     @endforeach
7   @else
8     <li>No existen usuarios</li>
9   @endif
10 </ul>
```

En caso de necesitar utilizar dicha comprobación, Laravel también proporciona la directiva `isset` y su cierre `endisset`. Estas realmente sustituyen y funciona del mismo modo que lo haría un if que llamase en su condición a la función `isset()`.

```

1 <ul>
2   ...
3   @isset($usuarios)
4     @foreach($usuarios as $usuario)
5       <li>{{ $usuario }}</li>
6     @endforeach
7   @else
8     <li>No existen usuarios</li>
9   @endisset
10 </ul>
```

Para evitar que se deban hacer estas comprobaciones antes de realiza un bucle `foreach`, Laravel proporciona una estructura llamada “`forelse`”.

La directiva `forelse` es similar a la `foreach` pero en este caso, `forelse` permite una cláusula adicional `empty` que se ejecutará si la colección no tiene elementos o está sin definir.

```

1 <ul>
2   ...
3   @forelse($usuarios as $usuario)
4     <li>{{ $usuario }}</li>
5   @empty
6     <li>No existen usuarios</li>
7   @endforelse
8 </ul>
```

Reutilizando Código

Es posible tener bloques de código más cortos que pueden reutilizarse en la aplicación.

Estos bloques de código, por convenio se llaman **partials**, ya que realmente son una parte que compone la web y se deben crear en una subcarpeta dentro de la carpeta `views`.

Por ejemplo se ha creado el archivo “navegacion.blade.php” como un **partial** de la interfaz. En este se ha creado una etiqueta “nav” con un listado de enlaces.

```

1 <nav>
2   <ul>
3     <li><a href="http://">Enlace 1</a></li>
4     <li><a href="http://">Enlace 2</a></li>
5     <li><a href="http://">Enlace 3</a></li>
6   </ul>
7 </nav>
```

Para incluir este partial en cualquier interfaz, solo sería necesario realizar un @include() del partial que se desee crear.



IMPORTANTE

Para poder referenciar subrutas en el código Blade de Laravel no se utiliza el símbolo "/" para separar los diferentes directorios. Sino que se hace utilizando el símbolo punto "."

```
1 @include("partials.navegacion")
```

Para bloques de código que se repiten de modo que refieren secciones de a todas o algunas páginas de la aplicación se generan las plantillas o como se llaman por convenio **layouts** que también se deberá crear la carpeta con dicho nombre dentro de **views**.

Estas plantillas cargarán diferentes secciones dentro de su código utilizando la directiva **@yield()** con el nombre de la sección que se desee cargar en esta parte como primer parámetro.

Por ejemplo, la plantilla "plantilla_inicial.blade.php" contiene el código de la estructura inicial de una página web html5. Dentro, el título y el contenido de la etiqueta "body" irán cambiando.

```
1 <!DOCTYPE html>
2 <html lang="es">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="author" content="Guillermo Garrido">
6   <title>@yield('titulo')</title>
7 </head>
8 <body>
9   @yield('contenido')
10 </body>
11 </html>
```

Por otro lado, se crearán archivos que utilicen estas secciones. Para ello deberán "heredar" la plantilla, de modo que harán uso de la directiva **@extends()** indicando como primer parámetro nombre de la plantilla a utilizar.

En estos archivos, se pueden generar diferentes secciones con la directiva **@section()** indicándole como primer parámetro el nombre de la sección. Para cerrar la sección se utilizará **@endsection**, de modo que el bloque contenido entre estas etiquetas podrá cargarse en las diferentes partes de la plantilla que cargue con **@yield()** el nombre de la sección creado.

Si el contenido de la sección no contiene HTML y solo es texto plano, se puede pasar como segundo parámetro de la directiva **@section()**.

Por ejemplo, la página “`index.blade.php`” hereda de la plantilla “`plantilla_inicial.blade.php`” y se compone de dos secciones. La primera sección “`contenido`” es la que se cargará dentro de la etiqueta “`body`” de la plantilla. Mientras que la segunda sección “`titulo`” solo es texto que se asignara al título de la página en la etiqueta “`title`”.

```

1 @extends('layouts.plantilla_inicial')
2
3 @section('contenido')
4     <h1>Indice</h1>
5     <p>Texto del bloque</p>
6 @endsection
7
8 @section('titulo', 'Página de Inicio')

```

Componentes

Los componentes son un estándar que se está extendiendo entre diferentes frameworks de programación.

Estos permiten crear plantillas, pero simplificando el código de la aplicación.

Para crear los componentes en primer lugar se debe crear una carpeta especial dentro de `views` llamada “`components`”. En esta se podrá crear la carpeta `layouts` y `partials` igual que se hacía directamente en `views` para agrupar las plantillas de la aplicación.

Dentro de las plantillas se utilizarán variables que se van a llamar `slots`. En concreto existe una variable reservada llamada `$slot` que imprimirá todo lo que se encuentre en el archivo que cargue el componente a lo que no se le haya asignado un nombre diferente. Estas son similares al `@yield()` que se utiliza en las plantillas.

```

1 <!DOCTYPE html>
2 <html lang="es">
3 <head>
4     <meta charset="UTF-8">
5     <meta name="author" content="Guillermo Garrido">
6     <title>Inicio</title>
7 </head>
8 <body>
9     {{ $slot }}
10 </body>
11 </html>

```

Para poder utilizar un componente existen dos opciones, se añade en la página donde se desea utilizar, la directiva `@component()` donde se le pasa como parámetro el componente que se desea cargar o bien es posible hacer uso de la etiqueta reservada `<x-...> </x-...>` donde los puntos suspensivos referencian el nombre del componente que se quiera cargar.

Dentro de este componente, podrá tener etiquetas HTML que mostraran el bloque de código.

Por ejemplo, si se crea la carpeta “partials” dentro de “components” se podrá llamar desde la página “plantilla_inicial.blade.php” como si fuese un **@include()**.

```

1 <!DOCTYPE html>
2 <html lang="es">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="author" content="Guillermo Garrido">
6   <title>Inicio</title>
7 </head>
8 <body>
9   <x-partials.navegacion/>
10  {{ $slot }}
11 </body>
12 </html>
```

Dentro de una plantilla es posible tener más de un slot a imprimir en diferentes lugares. De ser así la etiqueta se deberán crear más etiquetas slot utilizando la etiqueta **<x-slot>** pero en este caso se añadirá el atributo “**name**” al que se le definirá el nombre del slot. Estos slots con nombre suelen llamarse propiedades.

Por ejemplo si en la plantilla se añade un segundo slot llamado “pie”:

```

1 <!DOCTYPE html>
2 <html lang="es">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="author" content="Guillermo Garrido">
6   <title>Inicio</title>
7 </head>
8 <body>
9   <x-partials.navegacion/>
10  {{ $slot }}
11  {{ $pie }}
12 </body>
13 </html>
```

En el archivo “index.blade.php” se podrá utilizar la plantilla donde cargará la etiqueta “**<h1>**” dentro de la variable **\$slot**. Mientras que dentro de la variable \$pie cargara el HTML comprendido entre las etiquetas **<x-slot>** que tienen el atributo “**name='pie'**”

```

1 <x-plantilla_inicial>
2   <h1>Texto a imprimir en slot sin nombre</h1>
3   <x-slot name='pie'>
4     <footer>
5       <p>Creado por Guillermo Garrido.
6           <span>&copy; </span> 2022</p>
7     </footer>
8   </x-slot>
9 </x-plantilla_inicial>
```

En caso de no tener que imprimir HTML dentro del slot, se puede incorporar estas propiedades como atributos del componente. Para ello se añadirá el nombre de la propiedad como atributo y el valor de esta será el valor a mostrar en la propiedad.

Por ejemplo, si añadimos a la plantilla “`plantilla_inicial.blade.php`” dos variables, `$title` y `$metaAutor`, puesto que solo contendrán texto y no necesitan ninguna etiqueta HTML, podrán mostrarse como atributos de la plantilla.

```

1 ...
2   <meta name="author" content="{{ $metaAutor }}>
3   <title>{{ $title }}</title>
4 ...
```

De este modo, cuando se haga uso de la plantilla en “`index.blade.php`” se deberán añadir los dos atributos a dicha plantilla.

```

1 <x-plantilla_inicial titulo="Inicio"
2   meta-autor="Guillermo Garrido">
```



IMPORTANTE

Laravel utiliza la convención **camelCase** para definir el nombre de las variables. Pero para indicar el nombre de los atributos de una etiqueta HTML, como los atributos de una plantilla, utiliza **kebab-case** que es igual que el **snake_case** pero unido con guiones en vez de barra baja.

Si el atributo que se pasa dentro de la etiqueta este solo mostrara texto. Pero existen ocasiones que se desea que se compile como PHP de modo que se pueda concatenar o que se muestre el cálculo de varios valores o variables. En este caso es necesario utilizar delante del nombre del atributo el símbolo dos puntos “`:`”.

Por ejemplo, en este caso la etiqueta meta “`charset`” provendrá de una concatenación de texto con un calculo de una suma de dos enteros:

```
1 <x-plantilla_inicial titulo="Inicio" meta-autor="Guillermo Garrido" :charset="UTF-". (4 + 4)">
```

5. Controladores

Cuando un usuario interactúa con una vista, puede realizar ciertas peticiones al servidor que se realizan a través de las rutas. Estas rutas pueden contener cierta lógica de la aplicación proporcionando datos a las vistas, realizando operaciones, etcétera. Este funcionamiento de las rutas no es apropiado ya que al final se está mezclando toda la lógica de la aplicación en el archivo donde se almacenan todas las solicitudes HTTP.

Para poder desacoplar este código, existe una capa intermedia entre las vistas y los modelos llamada controladores.

Estos son el componente donde se agrupan las funciones que se ejecutan cuando se realizan las peticiones HTTP encargándose así de realizar la lógica del negocio y controles necesarios de las solicitudes que llegan.

De este modo la petición proveniente de la vista a través de una ruta se envía al controlador apropiado, el cual realizará las peticiones a base de datos (a través de los modelos), realizará los cálculos necesarios, preparará los datos para la vista, etcétera.

Tipos de controladores

Existen varios tipos de controladores. En concreto existen dos tipos que son los más utilizados en las aplicaciones de forma general.

Controladores de una sola función.

Estos controladores solo contienen una única función llamada “**__invoke()**” y que será la llamada cuando la ruta utilice el controlador.

Estos controladores solo podrán realizar una única tarea y devolverán, habitualmente a la vista, los datos procesados en esta.

Para crear un controlador de una sola función de un modo automático se puede utilizar la herramienta “**artisan**”. Si se ejecuta el comando el siguiente comando, se generará un controlador automáticamente dentro de la carpeta **app/http/controllers** con el nombre que se le indica

```
\ proyecto-garrido-laravel> php artisan make:controller UserController -i|
```

Con la última opción del comando “-i” generará la función “**__invoke()**” creada sin implementar dentro del controller.

```
1 public function __invoke(Request $request){
2     $usuario = "Guillermo Garrido";
3     return view('welcome', compact('usuario'));
4 }
```

Para llamar desde una ruta al controlador, simplemente es necesario añadir el nombre del controlador como segundo parámetro de la llamada.

```
1 Route::get('/', UserController::class);
```

Controladores de recursos

Los controladores de recursos o resources son aquellos que contienen siete métodos predefinidos que se utilizarán para controlar el flujo del CRUD (Create, Read, Update, Delete) de un modelo.

Para crear un controlador de tipo recursos solo es necesario cambiar en el comando de creación la opción “-i” por “-r”

```
\ proyecto-garrido-laravel> php artisan make:controller UserController -r
```

Los métodos que se generan al crear este tipo de controladores son los siguientes:

- index - muestra un listado de los elementos de ese modelo.
- create - muestra el formulario para dar de alta nuevos elementos del modelo.
- store - almacena en la base de datos el elemento creado con el formulario de create.
- show - muestra los datos de un modelo específico, habitualmente por su identificador.
- edit - muestra el formulario para editar un elemento existente del modelo.
- update - actualiza en la base de datos el elemento editado con el formulario de edit.
- destroy - elimina un elemento por su identificador.

```
1 <?php
2 namespace App\Http\Controllers;
3 use App\Models\User;
4 use Illuminate\Http\Request;
5 class UserController extends Controller
6 {
7     public function index(){}
8     public function create(){}
9     public function store(Request $request){}
10    public function show(User $user){}
11    public function edit(User $user){}
12    public function update(Request $request, User $user){}
13    public function destroy(User $user){}
14 }
```

Para poder hacer uso una función específica dentro de un controlador hay que indicar en la ruta a que función se llama dentro del controlador. Para ello como segundo parámetro de la ruta se enviará un array dos elementos, uno es el nombre del controlador y el segundo es el nombre de la función que se llama.

Es posible que varias de las rutas que se establezcan interactúen con el mismo controlador. Es por ello que Laravel proporciona la posibilidad de agrupar las rutas de modo que sea más sencillo el mantenimiento y la lectura de las mismas.

Para las funciones anteriores las rutas que se necesitarían serían las siguientes:

```
1 Route::controller(UserController::class)->group(function(){
2     Route::get('users', 'index')->name('users.index');
3     Route::post('users', 'store')->name('users.store');
4     Route::get('users/create', 'create')->name('users.create');
5     Route::get('users/{user}', 'show')->name('users.show');
6     Route::put('users/{user}', 'update')->name('users.update');
7     Route::delete('users/{user}', 'destroy')->name('users.destroy');
8     Route::get('users/{user}/edit', 'edit')->name('users.edit');
9 });
```

Puesto que es muy común tener controladores de recursos, Laravel cuenta con un método que crea todas estas rutas de forma automática simplemente creando una ruta especial que utiliza la función resource().

```
1 Route::resource('users', UserController::class);
```

Por otro lado, es posible crear un controlador de recursos, pero limitado para la API, de modo que solo atiende llamadas HTTP que reciban o devuelvan algún resultado pero elimina aquellos métodos encargados de mostrar vistas.

Para realizar un controlador de recursos para la API, se puede usar la opción “--api” en lugar de la opción de recursos “-r”.

```
\ proyecto-ggarrido-laravel> php artisan make:controller UserController --api|
```

Para utilizar estos métodos, se debe utilizar las rutas que se encuentran en el archivo api.php.

Todas las rutas de este archivo van a tener el prefijo “/api/”. Este se genera automáticamente si se utiliza la función resource().

```
1 Route::resource("/users", "UserController");
```

Esta creará internamente las rutas que equivaldrían a las siguientes:

```
1 Route::get('/api/users', 'index')->name('users.index');
2 Route::post('/api/users', 'store')->name('users.store');
3 Route::get('/api/users/{user}', 'show')->name('users.show');
4 Route::put('/api/users/{user}', 'update')->name('users.update');
5 Route::delete('/api/users/{user}', 'destroy')->name('users.destroy');
```