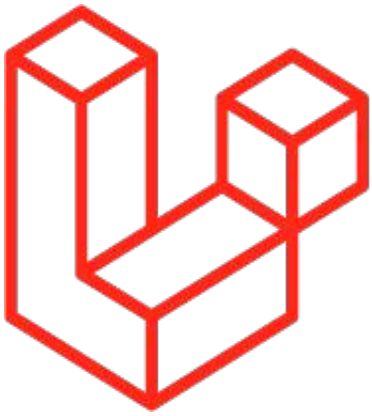


Laravel

CLASE 1 / 6

Indice

¿Qué es Laravel?....2
MVC: Modelo - Vista - Controlador....2
Estructura de un proyecto....3
  Carpeta app....5
Funcionamiento básico....5
Rutas....6
  Rutas básicas....6
  Añadir parámetros a las rutas....8
  Nombrar una ruta....9
  Llamar a controladores desde las rutas....10
  Renderizar vistas con las rutas....10
  Redirecciones de rutas....10
  Prefijos en las rutas....11
  La ruta de fallback....11
  Límite de tasa o rate limiting en las rutas....12
  Binding implicito....13
Vistas....13
  Definir vistas....13
  Referenciar y devolver vistas....14
  Pasar datos a una vista....14
  Vistas dentro de vistas....15
Plantillas Blade....15
  Mostrar datos....16
  Mostrar un dato solo si existe....16
  Comentarios....16
  Renderizando JSON....16
  Estructuras de control....17
  PHP....19
  Incluir una plantilla dentro de otra plantilla....19
  Layouts....19
Diseño de la aplicación web....21
  Incluir Assets....21
  Bootstrap....21
  Material Design....23
  Laravel Vite....23
Licencia....25

¿Qué es Laravel?

Laravel es un *framework* de código abierto para el desarrollo de aplicaciones web en PHP que posee una sintaxis simple, expresiva y elegante. Fue creado en 2011 por Taylor Otwell, inspirándose en Ruby on Rails y Symfony, de los cuales ha adoptado sus principales ventajas.

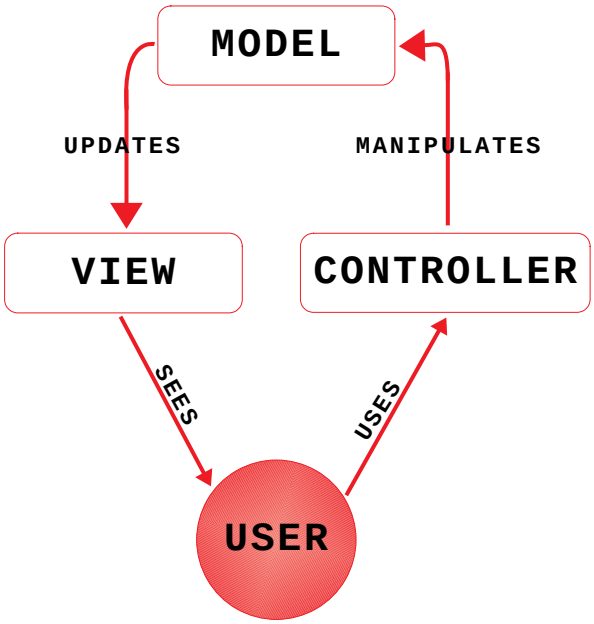
Laravel facilita el desarrollo simplificando el trabajo con tareas comunes como la autenticación, el enrutamiendo, gestión de sesiones, el almacenamiento en caché, etc. Algunas de las principales características y ventajas de Laravel son:

- Esta diseñado para desarrollar bajo el patrón MVC (modelo - vista - controlador), centrándose en la correcta separación y modularización del código. Lo que facilita el trabajo en equipo, así como la claridad, el mantenimiento y la reutilización del código.
- Integra un sistema ORM de mapeado de datos relacional llamado Eloquent aunque también permite la construcción de consultas directas a base de datos mediante su *Query Builder*.
- Permite la gestión de bases de datos y la manipulación de tablas desde código, manteniendo un control de versiones de las mismas mediante su sistema de *Migraciones*.
- Utiliza un sistema de plantillas para las vistas llamado Blade, el cual hace uso de la cache para darle mayor velocidad. Blade facilita la creación de vistas mediante el uso de *layouts*, herencia y secciones.
- Facilita la extensión de funcionalidad mediante paquetes o librerías externas. De esta forma es muy sencillo añadir paquetes que nos faciliten el desarrollo de una aplicación y nos ahorren mucho tiempo de programación.
- Incorpora un intérprete de línea de comandos llamado *Artisan* que nos ayudará con un montón de tareas rutinarias como la creación de distintos componentes de código, trabajo con la base de datos y migraciones, gestión de rutas, cachés, colas, tareas programadas, etc.

MVC: Modelo - Vista - Controlador

El modelo–vista–controlador (MVC) es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.

De manera genérica, los componentes de MVC se podrían definir como sigue:



- El Modelo: Es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones. Las peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'.
- El Controlador: Responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud de información (por ejemplo, editar un documento o un registro en una base de datos). Por tanto se podría decir que el 'controlador' hace de intermediario entre la 'vista' y el 'modelo'.
- La Vista: Presenta el 'modelo' y los datos preparados por el controlador al usuario de forma visual. El usuario podrá interactuar con la vista y realizar otras peticiones que se enviarán al controlador.

ESTRUCTURA DE UN PROYECTO

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

- `app` – Contiene el código principal de la aplicación. Esta dividida en muchas subcarpetas.
- `bootstrap` – En esta carpeta se incluye el código que se carga para procesar cada una de las llamadas a nuestro proyecto.
- `config` – Aquí se encuentran todos los archivos de configuración de la aplicación: base datos, cache, correos, sesiones o cualquier otra configuración general de la aplicación.
- `database` – En esta carpeta se incluye todo lo relacionado con la **definición de la base de datos** del proyecto. Dentro de ella se encuentran a su vez tres carpetas: *factories*, *migrations* y *seeds*.
- `lang` – En esta carpeta se guardan archivos PHP que contienen arrays con los textos del sitio web en diferentes lenguajes, solo será necesario utilizarla en caso que se desee que la aplicación se pueda traducir.
- `public` – Es la única carpeta pública, la única que debería ser **visible** en el servidor web. Todo las peticiones y solicitudes a la aplicación pasan por esta carpeta, ya que en ella se encuentra el `index.php`, este archivo es el que inicia todo el proceso de ejecución del

framework. En este directorio también se alojan los archivos CSS, Javascript, imágenes y otros archivos que se quieran hacer públicos.

- `resources` – Esta carpeta contiene a su vez tres carpetas: *views*, css *y js*:
    - `resources/views` – Este directorio contiene las vistas de la aplicación. En general serán plantillas de HTML que usan los controladores para mostrar la información. Hay que tener en cuenta que en esta carpeta no se almacenan los Javascript, CSS o imágenes, ese tipo de archivos se tienen que guardar en la carpeta `public`.
- `routes` – Es el sistema de rutas que se encargan de manejar el flujo de solicitudes y respuestas, desde y hacia el cliente (como hacia el navegador, por ejemplo). Definen la dirección URL y el método por el cual se puede ingresar a dicha ruta (GET, POST, etc.)
    - `routes/api.php`: En este archivo se definen todas las rutas de las APIs que puede llegar a tener la aplicación.
    - `routes/channels.php`: Aquí se definen los canales transmisión de eventos. Por ejemplo, cuando se realizan notificaciones en tiempo real o broadcast.
    - `routes/console.php`: Se definen comandos de consola que pueden interactuar con el usuario u otro sistema.
    - `routes/web.php`: En este archivo de rutas es donde se definen todas las rutas de la aplicación web que pueden ser ingresadas por la barra de direcciones del navegador.
- `storage` – En esta carpeta Laravel almacena toda la información interna necesarios para la ejecución de la web, como son los archivos de sesión, la caché, la compilación de las vistas, meta información y los logs del sistema.
- `tests` – Esta carpeta se utiliza para los archivos con las pruebas automatizadas. Laravel incluye un sistema que facilita todo el proceso de pruebas con PHPUnit.
- `vendor` – En esta carpeta se alojan todas las librerías y dependencias que conforman el *framework* de Laravel. Todo el código que contiene son librerías que se instalan y actualizan mediante la herramienta Composer.

Además en la carpeta raíz también podemos encontrar dos archivos muy importantes:

- `.env` – Se utiliza para almacenar los valores de configuración que son propios de la máquina o instalación actual. Lo que nos permite cambiar fácilmente la configuración según la máquina en la que se instale y tener opciones distintas para producción, para distintos desarrolladores, etc. Importante, este archivo debería estar en el `.gitignore`.

- `composer.json` – Este archivo es el utilizado por Composer para realizar la instalación de Laravel. En una instalación inicial únicamente se especificará la instalación de un paquete, el propio *framework* de Laravel, pero podemos especificar la instalación de otras librerías o paquetes externos que añadan funcionalidad a Laravel.

Carpeta app

La carpeta `app` es la que contiene el código principal del proyecto, como son los controladores, filtros y modelos de datos y es donde se crearan las clases a utilizar. Esta carpeta contiene a su vez muchas subcarpetas, pero la principal a utilizar es la carpeta `Http`:

- `app/Http/Controllers` – Contiene todos los archivos con las clases de los controladores que sirven para interactuar con los modelos, las vistas y manejar la lógica de la aplicación.
- `app/Http/Middleware` – Son los filtros o clases intermedias que se pueden utilizar para realizar determinadas acciones, como la validación de permisos, antes o después de la ejecución de una petición a una ruta del proyecto web.

Además de esta carpeta encontraremos muchas otras como *Console*, *Exceptions* y *Providers*.

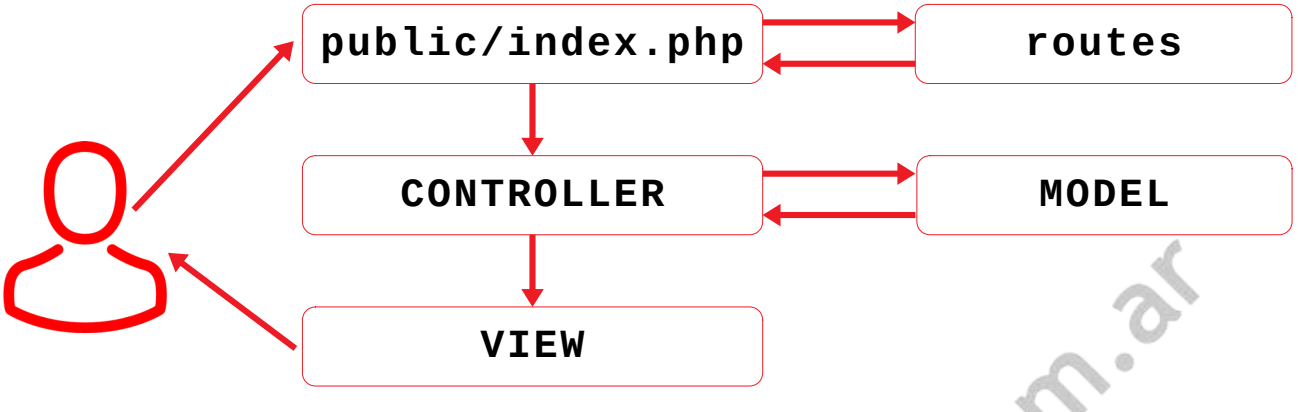
También se encuentra la carpeta `Models` donde se encuentra el archivo `User.php`, es un modelo de datos que viene predefinido por Laravel para trabajar con los usuarios de la web, que incluye métodos para hacer login, registro, etc. En esta carpeta alojaremos los modelos que se requieran utilizar en la aplicación.

Funcionamiento básico

El funcionamiento básico que sigue Laravel tras una petición web a una URL del sitio es el siguiente:

- Todas las peticiones entran a través del archivo `public/index.php`, el cual en primer lugar comprobará en el archivo de rutas (`routes/web.php`) si la URL es válida y en caso de serlo a que controlador tiene que hacer la petición.
- A continuación se llamará al método del controlador asignado para dicha ruta, el cual, dependiendo de la petición:
    - Accederá a la base de datos (si fuese necesario) a través de los "modelos" para obtener datos (o para añadir, modificar o eliminar).
    - Tras obtener los datos necesarios los preparará para pasárselos a la vista.
- En el tercer paso el controlador llamará a una vista con una serie de datos asociados, la cual se preparará para mostrarse correctamente a partir de los datos de entrada y por último se mostrará al usuario.

A continuación se incluye un pequeño esquema de este funcionamiento:



Rutas

Las rutas son llamadas dependiendo de la URL que el usuario requiera y los archivos que se encargan de ellas están en la carpeta `routes`.

- `api.php`: rutas que tienen el prefijo api, funcionan para cuando se crea una API que cualquiera puede consumir.
- `channels.php`: canales que sirven para propagar mensajes.
- `console.php`: rutas pero ahora desde la consola o terminal, no desde el navegador
- `web.php`: las rutas que son llamadas desde el navegador web.

Las rutas de la aplicación se tienen que definir en el archivo `routes/web.php`. Este es el punto centralizado para la definición de rutas y cualquier ruta no definida en este archivo no será válida, generado una excepción (lo que devolverá un error 404). A estas rutas se les asigna el grupo de middleware web, el cual proporciona algunas características como el estado de la sesión y la protección CSRF.

Las rutas, en su forma más sencilla, pueden devolver directamente un valor desde el propio archivo de rutas, pero también podrán generar la llamada a una vista o a un controlador.

Rutas básicas

Las rutas, además de definir la URL de la petición, también indican el método con el cual se ha de hacer dicha petición.

Los métodos o verbos HTTP indican lo que se desea hacer:

- GET: normalmente para obtener recursos o valores
- PUT: verbo usado para editar
- POST: para agregar algo nuevo, normalmente un insert para una base de datos. Con este método se envían los formularios.

- DELETE: cuando vamos a eliminar algo.

Nota: la diferencia entre GET y POST es que con GET los datos viajan en la url (si alguien ve las urls que visitas y por ellas se manda la contraseña, podría averiguarla) y con POST los mismos viajan en el cuerpo de la petición. Si se combina POST con https se obtiene una app segura.

Los dos métodos más utilizados son las peticiones tipo GET y tipo POST. Por ejemplo, para definir una petición tipo GET habría que añadir el siguiente código al archivo `web.php`:

```
Route::get('/', function()
{
    return '¡Hola mundo!';
});
```

Este código se lanzaría cuando se realice una petición tipo GET a la ruta raíz de la aplicación. Si se esta trabajando en local esta ruta sería `http://localhost/nombre_proyecto/public` pero cuando la web esté en producción se referiría al dominio principal, por ejemplo: `http://www.dirección-de-la-web.com`. Es importante indicar que si se realiza una petición tipo POST o de otro tipo que no sea GET a dicha dirección se devolvería un error ya que esa ruta no está definida.

Para definir una ruta tipo POST se realizaría de la misma forma pero cambiando el verbo GET por POST:

```
Route::post('/inicio', function()
{
    return '¡Hola mundo!';
});
```

De la misma forma podemos definir rutas para peticiones PUT, DELETE, PATCH o OPTIONS:

```
Route::put('/inicio', function () {
    //
});
```

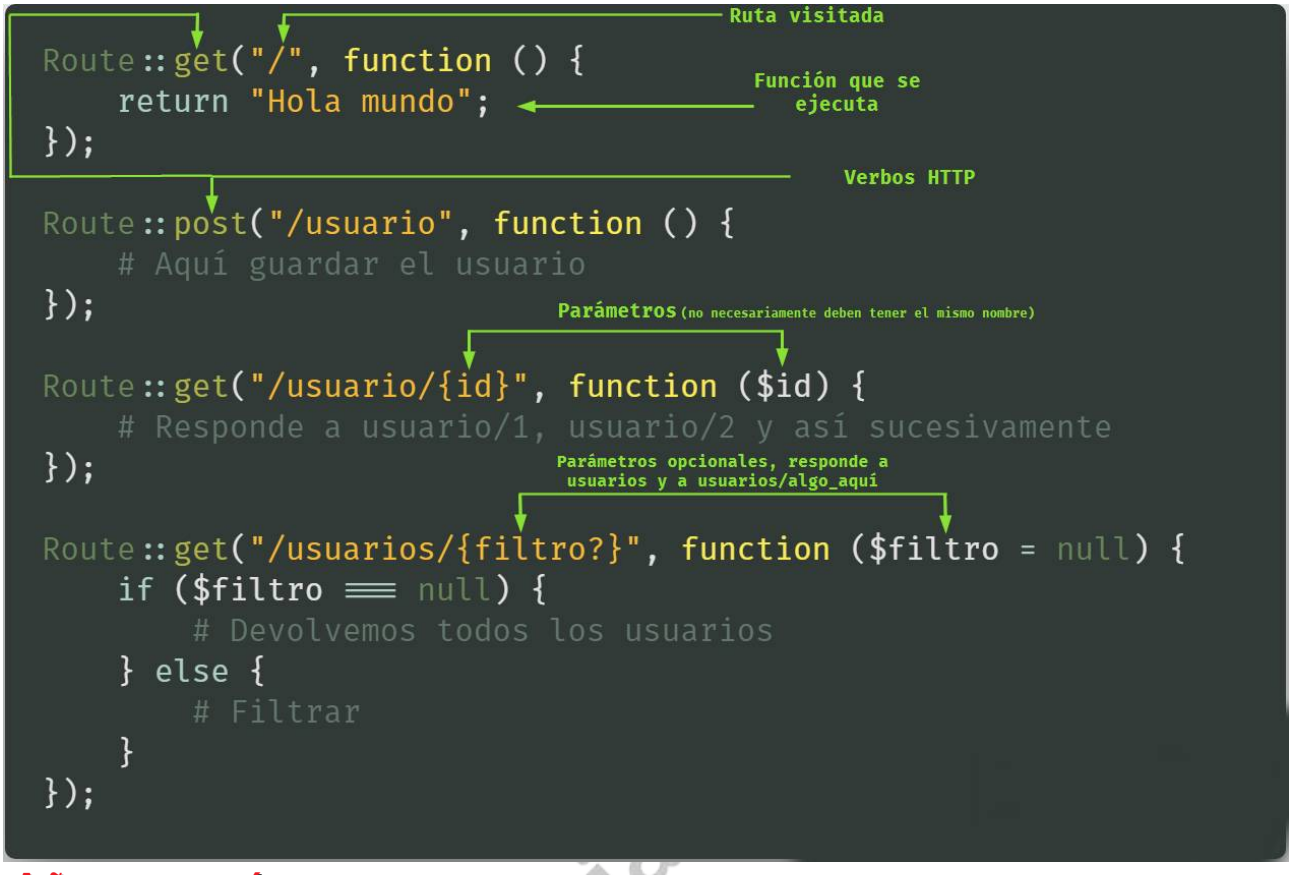
```
Route::delete('/inicio', function () {
    //
});
```

Si se desea que una ruta se defina a la vez para varios verbos se añadirá un array con los tipos, de la siguiente forma:

```
Route::match(['GET', 'POST'], '/', function()
{
    return '¡Hola mundo!';
});
```

O para cualquier tipo de petición HTTP utilizando el método `any`:

```
Route::any('/inicio', function()
```




```php
Route::get("/", function () {
    return "Hola mundo";
});

Route::post("/usuario", function () {
    # Aquí guardar el usuario
});

Route::get("/usuario/{id}", function ($id) {
    # Responde a usuario/1, usuario/2 y así sucesivamente
});

Route::get("/usuarios/{filtro?}", function ($filtro = null) {
    if ($filtro == null) {
        # Devolvemos todos los usuarios
    } else {
        # Filtrar
    }
});
```

AÑADIR PARÁMETROS A LAS RUTAS

Para añadir parámetros a una ruta se indican entre llaves `{}` a continuación de la ruta, de la forma:

```php
Route::get('user/{id}', function($id)
{
    return 'User '.$id;
});
```

En este caso se define la ruta `/user/{id}`, donde `id` es requerido y puede ser cualquier valor. En caso de no especificar ningún `id` se produciría un error. El parámetro se le pasará a la función, el cual se podrá utilizar para por ejemplo obtener datos de la base de datos, almacenar valores, etc.

También se puede indicar que un parámetro es opcional añadiendo el símbolo `?` al final (y en este caso no daría error si no se realiza la petición con dicho parámetro):

```php
Route::get('user/{name?}', function($name = null)
{
    return $name;
});
```

```php
// También podemos poner algún valor por defecto...
Route::get('user/{name?}', function($name = 'Matias')
{
    return $name;
```

```
});
```

Laravel también permite el uso de expresiones regulares para validar los parámetros que se le pasan a una ruta. Por ejemplo, para validar que un parámetro esté formado solo por letras o solo por números:

```
Route::get('user/{name}', function($name)
{
    //
})->where('name', '[A-Za-z]+');
```

```
Route::get('user/{id}', function($id)
{
    //
})->where('id', '[0-9]+');
```

```
// Si hay varios parámetros podemos validarlos usando un array:
Route::get('user/{id}/{name}', function($id, $name)
{
    //
})->where(['id' => '[0-9]+', 'name' => '[A-Za-z]+'])
```

Laravel incorpora para mayor comodidad, algunos patrones de expresiones regulares de uso común:

```
Route::get('/user/{id}/{name}', function ($id, $name) {
    //
})->whereNumber('id')->whereAlpha('name');
```

```
Route::get('/user/{name}', function ($name) {
    //
})->whereAlphaNumeric('name');
```

```
Route::get('/user/{id}', function ($id) {
    //
})->whereUuid('id');
```

```
Route::get('/category/{category}', function ($category) {
    //
})->whereIn('category', ['movie', 'song', 'painting']);
```

NOMBRAR UNA RUTA

Es conveniente darle nombre a una ruta que es frecuentemente utilizada por la aplicación para que sea fácilmente dirigida

```
Route::get('contactanos', function(){
    return 'Sección de contactos';
})->name('contacto');
```

```
Route::get('ayuda', function(){
    echo "<a href='" . route('contacto') . "'> Ayuda 1 </a><br>";
    echo "<a href='" . route('contacto') . "'> Ayuda 2 </a><br>";
    echo "<a href='" . route('contacto') . "'> Ayuda 3 </a><br>";
    echo "<a href='" . route('contacto') . "'> Ayuda 4 </a><br>";
});
```

Llamar a controladores desde las rutas

Es posible llamar a métodos de controladores desde las rutas, sin crear una función.

```
use App\Http\Controllers\ProductosController;

Route::post('/agregar', [ProductosController::class, 'agregarProducto']);
```

Dentro del controlador igualmente es posible recibir datos de la URL o datos de un formulario.

Renderizar vistas con las rutas

Normalmente se renderizaran algunas vistas dependiendo de la URL que se soliciten.

```
Route::view('/contacto', 'contacto');
Route::view('/acerca-de', 'acercaDe', ['anio' => date('Y')]);
Route::view('/quienes-somos', 'quienesSomos');
Route::view('/servicios', 'servicios', ['proveedor' => 'Garcia']);
```

El primer argumento es la URL y el segundo es el nombre de la vista que se va a renderizar.

Esto de las vistas igualmente puede hacerse desde un controlador; se recomienda poner aquí las vistas que son simples y no llevan (o llevan pocos) datos, para evitar la complejidad.

Redirecciones de rutas

Si se quiere redireccionar, se podría hacer esto:

```
Route::get('/pagina-antigua', function(){
    return redirect('pagina-nueva');
});
```

Pero Laravel proporciona una mejor manera:

```
Route::redirect('/pagina-antigua', '/pagina-nueva');
```

O si se quiere volver a la ruta anterior simplemente usar el método `back`:

```
return back();
```

O si se desdea pasar parametros a una ruta con nombre:

```
// para una ruta con la siguiente URI: /profile/{id}

return redirect()->route('profile', ['id' => 1]);
```

Para redireccionar a controladores:

```
use App\Http\Controllers\UserController;

return redirect()->action([UserController::class, 'index']);

return redirect()->action(
    [UserController::class, 'profile'], ['id' => 1]
);
```

PREFIJOS EN LAS RUTAS

Dependiendo de la aplicación puede que sea necesario prefijar o añadir una cadena constante al inicio de cada ruta. Por ejemplo, si hay algunas rutas que tienen que ver con el administrador, así:

```
administrador/escritorio
administrador/ajustes
administrador/cuenta
administrador/factura/1
```

Laravel provee una manera de prefijar y responder a cada ruta a partir de ese prefijo llamando a `Route::prefix()` y pasándole un argumento que es el prefijo.

```
Route::prefix("administrador")->group(function(){
    // Aquí llamar a Route::método como lo solíamos hacer
    // podemos pasar parámetros, poner nombres, expresiones
    // regulares y todo eso
    Route::get("/", function(){
        return "Yo respondo a administrador/";
    });
    Route::get("/escritorio", function(){
        return "Yo respondo a administrador/escritorio";
    });
    Route::get("/ajustes", function(){
        return "Yo respondo a administrador/ajustes";
    });
    Route::get("/cuenta", function(){
        return "Yo respondo a administrador/cuenta";
    });
    Route::get("/factura/{id}", function($id){
        return "Yo respondo a administrador/factura y puedo ver que el id
es $id";
    });
```

```
});
```

La ruta de fallback

La ruta de fallback permite personalizar o cambiar la forma en la que se maneja un error 404 en Laravel.

Para personalizar el comportamiento de cuando no se encuentra o resuelve ninguna ruta, se llamara a `Route::fallback` se le pasara una función que se encargará de mostrar un mensaje de error o redirigir, eso depende de la app.

```
Route::fallback(function () {
    // En este caso pongo muchos return para ejemplificar pero la función
    // se termina en el primer return que encuentre
    # Regresar una simple cadena
    return "No encontré la página que buscabas";

    # Renderizar una vista
    return view("nombre_de_la_vista");

    # Redireccionar
    return redirect("/");

    # Redireccionar con datos
    return redirect("/")->with("mensaje", "Hola mundo soy un mensaje");

    # Y recuerda que puedes hacer cualquier cosa como loguear que no
    # se encontró una página, enviar un correo o lo que sea
});
```

Nota: la documentación oficial dice que esta debería ser la última ruta en el archivo de rutas; es decir, hay que escribirla al final.

Límite de tasa o rate limiting en las rutas

Permite poner un límite de peticiones a las rutas. ejemplo cómo limitar las peticiones cada cierto tiempo, para que el usuario pueda solicitar los recursos de manera controlada.

```
Route::middleware("throttle:2,1")->group(function () {
    Route::get("/limitada", function () {
        return "Hola usuario. Solamente puedes verme 2 veces por minuto";
    });
    // Aquí abajo podrías agregar rutas que igualmente serán limitadas
    // con el throttle de arriba
```

```
});

Route::middleware("throttle:100,2")->group(function () {
    Route::get("/menos_limitada", function () {
        return "Hola usuario. Puedes verme 100 veces cada 2 minutos";
    });
    // Aquí abajo podrías agregar rutas que igualmente serán limitadas
    // con el throttle de arriba
});

Route::middleware("throttle")->group(function () {
    Route::get("/limitada_defecto", function () {
        return "Hola usuario. Puedes verme 60 veces cada 1 minuto,
        lo cual es la configuración por defecto";
    });
    // Aquí abajo podrías agregar rutas que igualmente serán limitadas
    // con el throttle de arriba
});
```

El **middleware de throttle** recibe dos argumentos: el **número de intentos** y el **número de minutos** en los que se puede acceder dentro del número de intentos.

En el primer ejemplo el usuario puede ver la página **2 veces por minuto**, en el segundo 100 veces por minuto y en el último se usa la configuración por defecto.

Por cierto, se podría agrupar las rutas de modo que el throttle se aplique a cada una de las rutas.

Binding implicito

Laravel resuelve automáticamente los modelos de eloquent definidos en las rutas o acciones de los controladores cuyos nombres de las variables con insinuación de tipo coinciden con el nombre de un segmento de ruta.

```
use App\Models\User;

Route::get('api/users/{user}', function (User $user) {
    return $user->email;
});
```

Vistas

Las vistas son la forma de presentar el resultado (una pantalla del sitio web) de forma visual al usuario, el cual podrá interactuar con él y volver a realizar una petición. Las vistas además permiten separar toda la parte de presentación de resultados de la lógica (controladores) y de la base de datos (modelos). Por lo tanto no tendrán que realizar ningún tipo de consulta ni procesamiento de datos, simplemente recibirán datos y los prepararán para mostrarlos como HTML.

DEFINIR VISTAS

Las vistas se almacenan en la carpeta `resources/views` como archivos PHP, y por lo tanto tendrán la extensión `.php`. Contendrán el código HTML del sitio web, mezclado con los assets (CSS, imágenes, Javascripts, etc. que estarán almacenados en la carpeta `public`) y algo de código PHP (o código *Blade* de plantillas) para presentar los datos de entrada como un resultado HTML.

A continuación se incluye un ejemplo de una vista simple, almacenada en el archivo `resources/views/home.php`, que simplemente mostrará por pantalla `¡Hola <nombre>!`, donde `<nombre>` es una variable de PHP que la vista tiene que recibir como entrada para poder mostrarla.

```
<html>
    <head>
        <title>Mi Web</title>
    </head>
    <body>
        <h1>¡Hola <?php echo $nombre; ?>!</h1>
    </body>
</html>
```

REFERENCIAR Y DEVOLVER VISTAS

Las vistas deben asociarse a una ruta para poder mostrarse. En el archivo `routes.php`

```
Route::get('/', function()
{
    return view('home', ['nombre' => 'Matias']);
});
```

En este caso se define que la vista se devuelva cuando se haga una petición tipo GET a la raíz del sitio, la función genera la vista usando el método `view` y la devuelve. Esta función recibe como parámetros:

- El nombre de la vista (en este caso `home`), el cual será un archivo almacenado en la carpeta `resources/views/home.php`. Para indicar el nombre de la vista se utiliza el mismo nombre del archivo pero sin la extensión `.php`.
- Como segundo parámetro recibe un array de datos que se le pasarán a la vista. En este caso la vista recibirá una variable llamada `$nombre` con valor "Matias".

También se puede utilizar la facade `View`:

```
use Illuminate\Support\Facades\View;

return View::make('home', ['apellido' => 'Garcia']);
```

Las vistas se pueden organizar en subcarpetas dentro de la carpeta `resources/views`, por ejemplo se podría tener una carpeta `resources/views/user` y dentro de esta todas las vistas relacionadas, como por ejemplo `login.php`, `register.php` o `profile.php`. En este caso

para referenciar las vistas que están dentro de subcarpetas tenemos que utilizar la notación tipo "*dot*", en la que las barras que separan las carpetas se sustituyen por puntos. Por ejemplo, para referenciar la vista `resources/views/user/login.php` usaríamos el nombre `user.login`, o la vista `resources/views/user/register.php` la cargaríamos de la forma:

```
Route::get('register', function()
{
    return view('user.register');
});
```

Pasar datos a una vista

Para pasar datos a una vista se utiliza el segundo parámetro del método `view`, el cual acepta un array asociativo. En este array se pueden añadir todas la variables que se necesiten utilizar dentro de la vista, ya sean de tipo variable normal (cadena, entero, etc.) u otro array u objeto con más datos. Por ejemplo, para enviar a la vista `profile` todos los datos del usuario cuyo `id` se recibe a través de la ruta:

```
Route::get('user/profile/{id}', function($id)
{
    $user = // Cargar los datos del usuario a partir de $id
    return view('user.profile', ['user' => $user]);
});
```

Laravel además ofrece una alternativa que crea una notación un poco más clara. En lugar de pasar un array como segundo parámetro podemos utilizar el método `with` para indicar una a una las variables o contenidos que se quieren enviar a la vista:

```
$view1 = view('home')->with('nombre', 'Matias');
```

```
$view2 = view('user.profile')
            ->with('user', $user)
            ->with('editable', false);
```

Vistas dentro de vistas

Laravel también permite anidar vistas, es decir, renderizar una vista dentro de otra vista. Por ejemplo, si se desea mostrar la vista almacenada en `resources/views/partials/view1.php` dentro de la vista anterior (`/resources/views/home.php`), hacer lo siguiente:

```
//códigos en web.php
$view = View::make('home')->nest('content', 'partials.view1');

// También podemos pasarle datos a la vista hija...
$view = View::make('home')->nest('content', 'partials.view1', $data);
```

El código anterior generaría la vista padre (`home`) y en su variable `content` colocaría la vista hija `partials.view1`.

```
//Código de home.php
<html>
    <body>
        <h1>¡Hola!</h1>
        <?php echo $content; ?>
    </body>
</html>
```

Plantillas Blade

Laravel utiliza *Blade* para la definición de plantillas en las vistas. Esta librería permite realizar todo tipo de operaciones con los datos, además de la sustitución de secciones de las plantillas por otro contenido, herencia entre plantillas, definición de *layouts* o plantillas base, etc.

Los archivos de vistas que utilizan el sistema de plantillas *Blade* tienen que tener la extensión `.blade.php`. Esta extensión tampoco se tendrá que incluir a la hora de referenciar una vista desde el archivo de rutas o desde un controlador. Es decir, se utiliza `view('home')` tanto si el archivo se llama `home.php` como `home.blade.php`.

En general el código que incluye *Blade* en una vista empezará por los símbolos `@` o `{{`, el cual posteriormente será procesado y preparado para mostrarse por pantalla. Blade no añade sobrecarga de procesamiento, ya que todas las vistas son preprocesadas y cacheadas, por el contrario brinda utilidades que ayudarán en el diseño y modularización de las vistas.

Mostrar datos

El método más básico de *Blade* es el de mostrar datos, para esto se utilizan las llaves dobles ({{ }}) y dentro de ellas se escribe la variable o función PHP con el contenido a mostrar:

```
Hola {{ $name }}.
La hora actual es {{ time() }}.
```

Blade se encarga de verificar el resultado llamando a `htmlentities` para prevenir errores y ataques de tipo XSS. Si en algún caso no se quiere verificar los datos se llamará:

```
Hola {!! $name !!}.
```

Nota: En general siempre se usan las llaves dobles, en especial si se mostraran datos que son proporcionados por los usuarios de la aplicación. Esto evitará que inyecten símbolos que produzcan errores o inyecten código javascript que se ejecute sin que nosotros queramos. Por lo tanto, este último método solo debe ser utilizado si es seguro el contenido que se recibirá.

Mostrar un dato solo si existe

Para comprobar que una variable existe o tiene un determinado valor podemos utilizar el operador ternario de la forma:

```
{{ isset($name) ? $name : 'Valor por defecto' }}
```

O simplemente usar la notación que incluye *Blade* para este fin:

```
{{ $name or 'Valor por defecto' }}
```

Comentarios

Para escribir comentarios en *Blade* se utilizan los símbolos `{{--` y `--}}`, por ejemplo:

```
{{-- Este comentario no se mostrará en HTML --}}
```

Renderizando JSON

A veces puede pasar un array a su vista con la intención de renderizarlo como JSON para inicializar una variable JavaScript. Sin embargo, en lugar de llamar manualmente a json_encode, puede usar la directiva del método Illuminate\Support\Js::from. El método from acepta los mismos argumentos que la función json_encode de PHP; sin embargo, se asegurará de que el JSON resultante se escape correctamente para su inclusión dentro de las comillas HTML. El método from devolverá una declaración JSON.parse de cadena que convertirá el objeto o la matriz dados en un objeto JavaScript válido:

```
<script>
    var app = {{ Illuminate\Support\Js::from($array) }};
</script>
```

Estructuras de control

Blade permite utilizar la estructura `if` de las siguientes formas:

```
@if (count($users) === 1 )
    Solo hay un usuario!
@elseif (count($users) > 1)
    Hay muchos usuarios!
@else
    No hay ningún usuario :(
@endif
```

Para mayor comodidad, Blade también proporciona una directiva `@unless`:

```
@unless (Auth::check())
    Ud. no se encuentra logueado.
@endunless
```

Además de las directivas condicionales ya discutidas, las directivas @isset y @empty pueden usarse como atajos convenientes para sus respectivas funciones de PHP.

```
@isset($records)
    // $records Está definido y no es nulo...
@endisset
```

```
@empty($records)
    // $records es "vacio"...
@endempty
```

En los siguientes ejemplos se puede ver como realizar bucles tipo `for`, `while` o `foreach`:

```
@for ($i = 0; $i < 10; $i++)
    El valor actual es {{ $i }}
@endfor

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No existen usuarios</p>
@endforelse

@while (true)
    <p>Soy un bucle while infinito!</p>
@endwhile

@foreach ($users as $user)
    <p>Usuario {{ $user->name }} con identificador: {{ $user->id }}</p>
@endforeach
```

Al usar bucles, también puede omitir la iteración actual o finalizar el bucle usando las directivas `@continue` y `@break`:

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif

    <li>{{ $user->name }}</li>

    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

Cuando se utiliza un `foreach` automáticamente se va a crear la variable `$loop` que es un objeto con determinadas propiedades:

- `index`: el índice, comienza en 0. `$loop->index`.
- `iteration`: como el índice pero comienza en 1. Útil para mostrar el número pero en la forma no programadora.
- `remaining`: el número de iteraciones que faltan para terminar el ciclo.

- `count`: longitud del arreglo que está siendo recorrido.
- `first`: va a estar en true en el primer elemento.
- `last`: va a estar en true en el último elemento.
- `depth`: la profundidad del ciclo. Esto es útil en ciclos anidados.
- `parent`: si estamos en un ciclo anidado, `$loop->parent` se refiere al `$loop` del ciclo padre o del ciclo superior.

También hay directivas tomadas de PHP como `@isset()` y `@empty()` que indican si esta o no seteado el valor de una variable. Switch funciona igual que el switch de PHP. Se comienza con `@switch($valor)`, se termina con `@endswitch` y para los casos se utiliza `@case($unValor)`. Abajo de cada `@case` debe ir un `@break` y para el valor por defecto se usa `@default`.

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break

    @default
        Default case...
@endswitch
```

Estas son las estructuras de control más utilizadas. Ademas de estas *Blade* define algunas más que se pueden ver directamente en su documentación: https://laravel.com/docs/9.x/blade

PHP

En algunas situaciones, es útil incrustar código PHP en sus vistas Blade, utilizar @php para ejecutar un bloque de PHP plano dentro de tu plantilla:

```
@php
    //
@endphp
```

INCLUIR UNA PLANTILLA DENTRO DE OTRA PLANTILLA

En *Blade* indicar que se incluya una plantilla dentro de otra plantilla con la instrucción `@include`:

```
@include('view_name')
```

Ademas se puede pasar un array de datos a la vista a cargar usando el segundo parámetro del método `include`:

```
@include('view.name', ['status' => 'complete'])
```

Esta opción es muy útil para crear vistas que sean reutilizables o para separar el contenido de una vista en varios archivos.

LAYOUTS

Blade también permite la definición de *layouts* para crear una estructura HTML base con secciones que serán rellenadas por otras plantillas o vistas hijas. Por ejemplo, crear un *layout* con el contenido principal o común de la web (*head*, *body*, etc.) y definir una serie de secciones que serán rellenados por otras plantillas para completar el código. Este *layout* puede ser utilizado para todas las pantallas del sitio web, lo que permite que en el resto de plantillas no se repita todo este código.

A continuación de incluye un ejemplo de una plantilla tipo *layout* almacenada en el archivo `resources/views/layouts/master.blade.php`:

```
<html>
    <head>
        <title>Mi Web</title>
    </head>
    <body>
        @section('menu')
            Contenido del menu
        @show

        <div class="container">
            @yield('content')
        </div>
    </body>
</html>
```

Posteriormente, en otra plantilla o vista, indicar que extienda el *layout* que se ha creado (con `@extends('layouts.master')`) y que complete las dos secciones de contenido que se definido en el mismo:

```
@extends('layouts.master')

@section('menu')
    @parent
    <p>Este contenido es añadido al menú principal.</p>
@endsection

@section('content')
    <p>Este apartado aparecerá en la sección "content".</p>
@endsection
```

Como se puede ver, las vistas que extienden un *layout* simplemente tienen que sobrescribir las secciones del *layout*. La directiva `@section` permite ir añadiendo contenido en las plantillas hijas, mientras que `@yield` será sustituido por el contenido que se indique. El método `@parent` carga en la posición indicada el contenido definido por el padre para dicha sección.

El método `@yield` también permite establecer un contenido por defecto mediante su segundo parámetro:

```
@yield('section', 'Contenido por defecto')
```

Diseño de la aplicación web

Incluir Assets

En Laravel se denomina asset a cada uno de los recursos que son utilizados por la aplicación, generalmente son archivos estáticos, que ayudan a definir el comportamiento y aspecto de la misma y se encuentran dentro del directorio `public` (CSS, JS e imágenes), y se cargan sin problemas como rutas absolutas utilizando el helper `asset()`, independientemente del entorno donde está la aplicación, si se utiliza la función del mismo nombre. Así para:

Cargar una hoja de estilos:

```
<link rel="stylesheet" href="{!! asset('css/estilos.css') !!}">
```

Cargar un icono (para los favoritos del navegador):

```
<link rel="icon" href="{!! asset('favicon.ico') !!}" type="image/x-icon">
```

Cargar librería JS:

```
<script type="text/javascript" src="{!! asset('js/api.js') !!}"
async></script>
```

Cargar una imagen:

```
<img src="{{ asset('imagenes/foto.JPG') }}" alt="mi foto">
```

Bootstrap

Bootstrap (https://getbootstrap.com/) es una biblioteca multiplataforma o conjunto de herramientas de código abierto para diseño de sitios y aplicaciones web. Contiene plantillas de diseño con tipografía, formularios, botones, cuadros, menús de navegación y otros elementos de diseño basado en HTML y CSS, así como extensiones de JavaScript adicionales. A diferencia de muchos frameworks web, solo se ocupa del desarrollo front-end.

Bootstrap proporciona un conjunto de hojas de estilo que proveen definiciones básicas de estilo para todos los componentes de HTML. Esto otorga una uniformidad al navegador y al sistema de anchura, da una apariencia moderna para el formateo de los elementos de texto, tablas y formularios.

Existen dos formas de utilizarlo en un proyecto Laravel:

- Instalar en la carpeta del proyecto utilizando composer:

```
composer require components/jquery
composer require rsportella/popper
composer require twbs/bootstrap:5.2.0
```

Luego habrá que editar el archivo `composer.json` y agregar:

```
"scripts": {
        "post-update-cmd": [
            "rm -rf public/bootstrap",
            "cp -R vendor/twbs/bootstrap/dist public/bootstrap",
            "rm -rf public/jquery",
            "cp -R vendor/components/jquery public/jquery"
        ]
    }
```

Habrá que actualizar composer.

```
composer update
```

Por ultimo habrá que agregar en el archivo de la vista los link para hacer referencia a la ubicación de los estilos de bootstrap

```
<link href="{!! asset('bootstrap/css/bootstrap.min.css') !!}"
rel="stylesheet">

<script src="{!! asset('jquery/jquery.min.js') !!}"></script>
<script src="{!! asset('bootstrap/js/bootstrap.min.js') !!}"></script>
```

- Utilizarlo por CDN

  Sin tener que descargar el framework, utilizando un servidor donde se alojan todos los archivos de Bootstrap, usando una CDN (Content Delivery Network - red de entrega de contenidos).

  Solo se requiere agregar en el archivo de la vista los link que hacen referencia a la CDN de Bootstrap, Jquery y Popper.js

```
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0/dist/css/bootstrap.min.
css" rel="stylesheet"
integrity="sha384-gH2yIJqKdNHPEq0n4Mqa/HGKIhSkIHeL5AyhkYV8i59U5AR6csBvApHH
Nl/vI1Bx" crossorigin="anonymous">

<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.0/dist/js/bootstrap.bundle
.min.js" integrity="sha384-
A3rJD856KowSb7dwlZdYEkO39Gagi7vIsF0jrRAoQmDKKtQBHUuLZ9AsSv4jD4Xa"
crossorigin="anonymous"></script>
```

Material Design

Material design es una normativa de diseño enfocado en la visualización del sistema operativo Android, además en la web y en cualquier plataforma. Fue desarrollado por Google. https://material.io/

```
<link
href="https://unpkg.com/material-components-web@latest/dist/material-
components-web.min.css" rel="stylesheet">

<script
src="https://unpkg.com/material-components-web@latest/dist/material-
components-web.min.js"></script>

<link rel="stylesheet" href="https://fonts.googleapis.com/icon?
family=Material+Icons">
```

Laravel Vite

Vite es una herramienta de construcción de interfaz moderna que proporciona un entorno de desarrollo extremadamente rápido y empaqueta su código para la producción. Al crear aplicaciones con Laravel, normalmente usará Vite para agrupar los archivos CSS y JavaScript de su aplicación en activos listos para producción.

El objetivo es, entre otras cosas, procesar todo el código CSS, minificarlo y combinarlo en un solo archivo. Lo mismo para el código javascript, minificar, ofuscar y combinar todo el código en un solo archivo, haciendo las páginas web mas seguras y rápidas.

Primero de todo debe tener instalado Node.js y NPM

```
sudo apt install nodejs npm
```

En la raíz del proyecto y ejecutar el siguiente comando para instalar los paquetes que están en el archivo package.json . Serán los necesarios para poder utilizar Laravel Vite:

```
npm install
```

Vite se configura a través de un archivo vite.config.js en la raíz de su proyecto. Puede personalizar este archivo según sus necesidades y también puede instalar cualquier otro complemento que requiera su aplicación, como `@vitejs/plugin-vue` o `@vitejs/plugin-react.`

El complemento Laravel Vite requiere que especifique los puntos de entrada para su aplicación. Estos pueden ser archivos JavaScript o CSS e incluyen lenguajes preprocesados como TypeScript, JSX, TSX y Sass.

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel([
            'resources/css/app.css',
```

```
            'resources/js/app.js',
        ]),
    ],
});
```

debe importar su CSS a través de JavaScript. Por lo general, esto se haría en el archivo resources/js/app.js de su aplicación:

```
import './bootstrap';
import '../css/app.css';
```

Con los puntos de entrada de Vite configurados, solo necesita hacer referencia a ellos en una directiva Blade @vite() que agrega al <head> de la plantilla raíz de su aplicación:

```
<!doctype html>
<head>
    {{-- ... --}}

    @vite(['resources/css/app.css', 'resources/js/app.js'])
</head>
```

Hay dos formas de ejecutar Vite. Puede ejecutar el servidor de desarrollo a través del comando dev, que es útil al desarrollar localmente. El servidor de desarrollo detectará automáticamente los cambios en sus archivos y los reflejará instantáneamente en cualquier ventana abierta del navegador.

O bien, ejecutar el comando build creará una versión y empaquetará los activos de su aplicación y los preparará para que los implemente en producción:

```
# Run the Vite development server...
npm run dev

# Build and version the assets for production...
npm run build
```

Mas información sobre Laravel Vite en https://vitejs.dev

LICENCIA

Matías E. García

Prof. & Tec. en Informática Aplicada
www.profmatiasgarcia.com.ar
info@profmatiasgarcia.com.ar