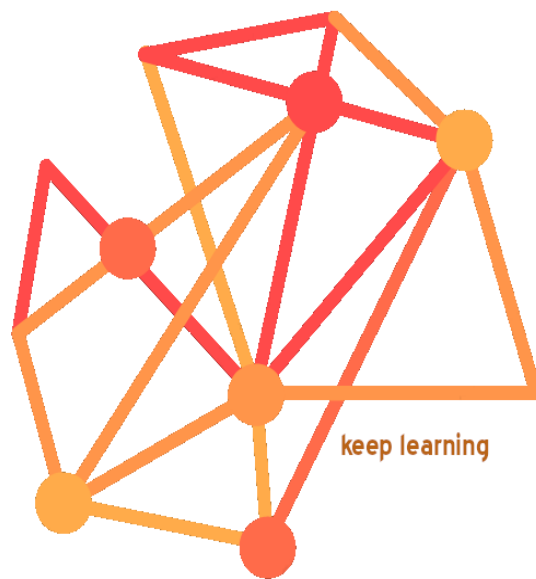


PHP Laravel: básico y ficheros



Juan Carlos Pérez Rodríguez

Sumario

Introducción a Laravel.....	3
Creación nuevo proyecto.....	4
Estructura de un proyecto Laravel.....	5
Creación de rutas en laravel.....	11
Parámetros en el path y enrutado según expresiones regulares.....	15
Creando nuestro primer controlador.....	16
Anotaciones.....	20
.....	20
Comenzando con las Vistas en Laravel (Blade).....	21
¿Qué es Blade?.....	22
Directiva @foreach.....	22
Directivas @if, @else, @elseif.....	24
Directivas @isset, @empty.....	24
Otras directivas: @for, @while.....	25
Mostrar imágenes desde fichero publico.....	26
Laravel Request.....	27
Sesiones en Laravel.....	28
@CSRF en los formularios.....	30
Gestión de Ficheros.....	32
Crear un directorio y listarlo.....	34
Almacenar un fichero recibido por un formulario.....	35
Descargar un fichero.....	37
Borrar un fichero.....	38
Anexo: Visual studio debug.....	39
Anexo: Restaurar proyecto desde github.....	41

Introducción a Laravel

Según wikipedia:

Laravel es un framework de código abierto para desarrollar aplicaciones y servicios web con PHP 5 y PHP 7. Gran parte de Laravel está formado por dependencias, especialmente de Symfony, esto implica que el desarrollo de Laravel dependa también del desarrollo de sus dependencias. Se base en patrón MVC particularizado: Routes with Closures', en lugar de un MVC tradicional con el objetivo de hacer el código más claro. Aun así permite el uso de MVC tradicional

Aunque no vamos a centrarnos en la instalación (queda como autoformación del alumno y hay múltiples manuales en la red, como por ejemplo: <https://www.ecodeup.com/aprende-a-instalar-composer-en-windows-10/>) diremos que esa necesidad de dependencias es lo que hace que sea necesario un gestor de dependencias. Usaremos Composer como gestor de dependencias para PHP como paso previo a la instalación de Laravel

Creación nuevo proyecto

Podemos crear proyecto de forma sencilla recurriendo a composer:

```
composer create-project laravel/laravel nombreproyecto
```

Abrir desde visualstudiocode la carpeta del proyecto generado allí podremos observar la estructura del proyecto que antes se describió.

Arrancamos nuestra aplicación mediante el comando:

```
cd nombreproyecto  
php artisan serve
```

Esta sería otra forma de crear el proyecto:

Primero en la carpeta donde vamos a crear todos nuestros proyectos laravel

```
composer require laravel/installer
```

Una vez instalado ejecutamos en esa misma carpeta:

```
vendor/laravel/installer/bin/laravel new nombreproyecto
```

Para cada nueva instalación de un proyecto usaríamos la última instrucción

Estructura de un proyecto Laravel

Directorio App

El directorio app contiene el código principal de tu aplicación. Exploraremos este directorio con más detalle pronto; sin embargo, casi todas las clases en tu aplicación estarán en este directorio.

Directorio Bootstrap

El directorio bootstrap contiene el archivo app.php que maqueta el framework. Este directorio también almacena un directorio cache que contiene archivos generados por el framework para optimización de rendimiento como los archivos de cache de rutas y servicios.

Directorio Config

El directorio config, como el nombre implica, contiene todos los archivos de configuración de la aplicación.

Directorio Database

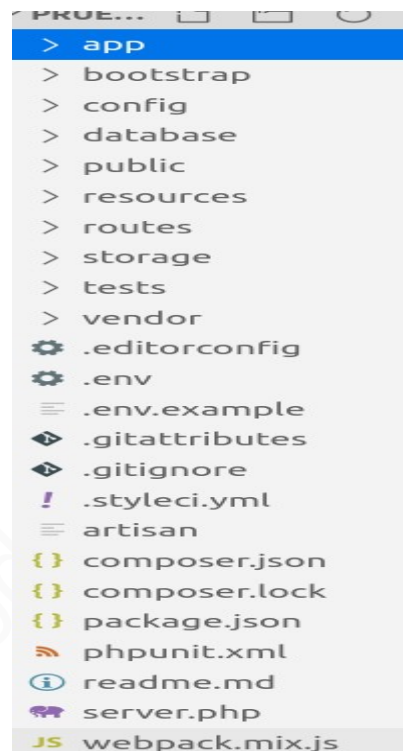
El directorio database contiene las migraciones de tu base de datos, model factories y seeders. Si se desea, puedes también usarse este directorio para almacenar una base de datos SQLite.

Directorio Public

El directorio public contiene el archivo index.php, el cual es el punto de acceso para todas las solicitudes llegan a tu aplicación y configura la autocarga. Este directorio también almacena tus assets, tales como imágenes, JavaScript y CSS.

Directorio Resources

El directorio resources contiene tus vistas así como también tus assets sin compilar tales como LESS, Sass o JavaScript. Este directorio también almacena todos tus archivos de idioma.



Directorio Routes

El directorio routes contiene todas las definiciones de rutas para tu aplicación. Por defecto, algunos archivos de rutas son incluidos con Laravel: web.php, api.php, console.php y channels.php.

El archivo web.php contiene rutas que RouteServiceProvider coloca en el grupo de middleware web, que proporciona estado de sesión, protección CSRF y encriptación de cookies. Si tu aplicación no ofrece una API sin estado, todas tus rutas probablemente serán definidas en el archivo web.php.

El archivo api.php contiene rutas que RouteServiceProvider coloca en el grupo de middleware api, que proporcionan limitación de velocidad. Estas rutas están pensadas para no tener estado, así que las solicitudes que llegan a la aplicación a través de estas rutas están pensadas para ser autenticadas mediante tokens y no tendrán acceso al estado de sesión.

El archivo console.php es donde puedes definir todos los comandos basados en Closures de tu aplicación. Cada Closure está enlazado a una instancia de comando permitiendo una forma simple de interactuar con los métodos de entrada y salida de cada comando. Aunque este archivo no define ninguna ruta HTTP, sí define puntos de entrada en consola (rutas) a tu aplicación.

El archivo channels.php es donde puedes registrar todos los canales de transmisión de eventos que tu aplicación soporta.

Directorio Storage

El directorio storage contiene tus plantillas compiladas de Blade, sesiones basadas en archivos, archivos de caches y otros archivos generados por el framework. Este directorio está segregado en los directorios app, framework y logs. El directorio app puede ser usado para almacenar cualquier archivo generado por tu aplicación. El directorio framework es usado para almacenar archivos generados por el framework y cache. Finalmente, el directorio logs contiene los archivos de registros de tu aplicación.

El directorio storage/app/public puede ser usado para almacenar archivos generados por el usuario, tales como imágenes de perfil, que deberían ser accesibles públicamente. Debes crear un enlace simbólico en public/storage que apunte a este directorio. Puedes crear el enlace usando el comando `php artisan storage:link`.

El Directorio Tests

El directorio tests contiene tus pruebas automatizadas. Una prueba de ejemplo de PHPUnit es proporcionada. Cada clase de prueba debe estar precedida por la palabra Test. Puedes ejecutar tus pruebas usando los comandos `phpunit` o `php vendor/bin/phpunit`.

Directorio Vendor

El directorio vendor contiene tus dependencias de Composer.

De momento vamos a centrarnos en dos ficheros:

nombreproyecto/**routes/web.php**

nombreproyecto/**resources/views/welcome.blade.php**

web.php le indica a laravel que se debe hacer con cada solicitud que realiza el usuario

- routes/api.php: En este archivo se definen todas las rutas de las APIs que puede llegar a tener nuestra aplicación.

- routes/channels.php: Aquí definimos los canales transmisión de eventos. Por ejemplo, cuando realizamos notificaciones en tiempo real.

- routes/console.php: En el archivo de rutas console.php definimos comandos de consola que pueden interactuar con el usuario u otro sistema.

- routes/web.php: En este archivo de rutas es donde definimos todas las rutas de nuestra aplicación web que pueden ser ingresadas por la barra de direcciones del navegador.

Como ejemplo, podemos observar que en web.php aparece:

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

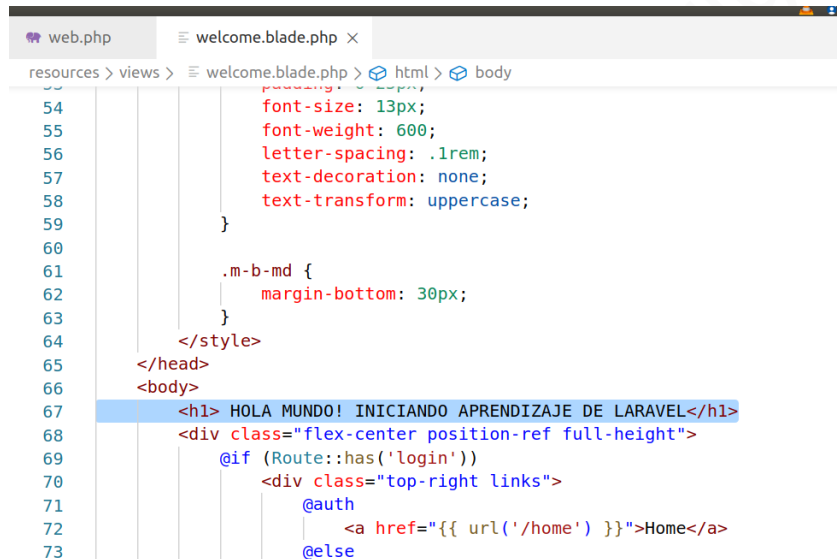
Le estamos indicando que a una petición de la ruta principal de la aplicación: “/” lo encaminaremos a la vista: “welcome” (fichero: **resources/views/welcome.blade.php**) La petición debe ser de tipo **GET** (observar que pusimos: **Route::get**)

Las diferentes opciones que tenemos según el tipo de verbo HTTP serían:

```
Route::get($uri, $callback);
Route::post($uri, $callback);
Route::put($uri, $callback);
Route::patch($uri, $callback);
Route::delete($uri, $callback);
Route::options($uri, $callback);
```

Vemos que como primer parámetro siempre detallamos la url a la que queremos responder (en el caso de nuestro ejemplo inicial era la petición de la carpeta raíz: "/") Y el segundo parámetro es la función que se debe ejecutar

Pongamos un mensaje personalizado al inicio de ese fichero y veamos el efecto en la ejecución de la aplicación:



Una vez arrancado el servicio (el comando `php artisan serve` que dijimos antes) no es preciso hacer otra cosa que grabar el fichero y observar en el navegador las diferencias:

<http://localhost:8000>

(o si está ocupado el puerto ir al siguiente disponible: 8001 etc)

Ocurre que a partir de la versión 6 el módulo de autenticación hay que instalarlo aparte. Vamos a la dirección de la carpeta que creamos con el comando anterior:

```
cd nombreproyecto;
composer require laravel/ui
php artisan ui vue --auth
```



```
npm install && npm run dev  
composer require doctrine/dbal
```

Con los comandos anteriores, por ejemplo, nos genera en las vistas: auth, home.blade.php, layouts . La última línea sirve para hacer actualizaciones en el ORM Eloquent (una vez creadas las tablas en las migraciones, para modificaciones posteriores)

Si miramos en routes/web.php observamos que se ha creado la siguiente línea:

```
Auth::routes();
```

esa línea tiene internamente las siguientes rutas:

```
GET /login  Que lleva al form de login  
POST /login  Que envía el form de login  
POST /logout  que cierra la sesión  
GET /register  para el form de registro  
POST /register  envía el registro
```

Las anteriores rutas tienen que tener un controlador (Laravel se apoya en el modelo MVC). En este caso:

LoginController:

- GET /login , - POST /login , POST /logout

RegisterController:

- GET /register , - POST /register

Podemos ver los anteriores controllers en: `app/http/controllers/auth` y hacer las modificaciones que consideremos pertinentes

En cuanto a las vistas:

crea en resources/views una carpeta auth y una layout con varias vistas:

auth/

login.blade.php

register.blade.php

layouts/

app.blade.php => para el html con estilo bootstrap (lo usan login, register, etc)

A partir de ahora vamos a ir creando un proyecto de una tienda online y explicaremos los diferentes conceptos relacionados con Laravel

Creación de rutas en laravel

Nota: En versiones laravel > 8, si vamos a usar la nomenclatura que usa arroba: “@” para identificar en una ruta los métodos/funciones de una clase debemos incluir todo el namespace

Ejemplo: en lugar de:

```
Route::get('/home', 'HomeController@index')->name('home')
```

pondremos la ruta completa del namespace:

```
Route::get('/home', 'App\Http\Controllers\HomeController@index')->name('home')
```

Aunque la tendencia en laravel > 8 es enviar un array:

```
Route::get('/home', [HomeController::class, 'index'])->name('home')
```

En web.php una vez creado el proyecto aparecerá algo parecido a:

```
Route::get('/', function () {  
    return view('welcome');  
});  
  
Auth::routes();  
  
Route::get('/home', 'HomeController@index')->name('home');
```

Ya dijimos que la primera ruta nos enviaba a una vista llamada welcome.blade.php En el caso que nosotros queremos reproducir no debiera derivarnos a una vista sino a un fichero controlador. Si nos fijamos la última línea que nos ha creado laravel observamos:

```
Route::get('/home', 'HomeController@index')->name('home');
```

Nos está indicando que las rutas a: /home por medio de http GET serán gestionadas por la función: **index** que está dentro del fichero controlador: HomeController **Adicionalmente le ponemos el nombre “home” a la ruta: ->name('home')**

¿ qué utilidad tiene ponerle un nombre a la ruta ?

Imaginemos que queremos hacer una redirección en cualquier momento de nuestra aplicación. Si la ruta tiene nombre podemos usarlo para que nos haga la redirección:

Ruta en web.php:

```
// El comando http: GET usuarios/crear será gestionado por la función
// create() que está dentro del fichero: UserController
// A esta ruta le ponemos el nombre: users.create
Route::get('usuarios/crear', 'UserController@create')
->name('users.create');
```

Y ahora desde alguna parte de la aplicación en la que queramos hacer una redirección:

```
public function store(Request $request)
{
    //blablabla: código que se quiera hacer en esta función
    return redirect()->route('users.create');
}
```

Observamos que hacemos una redirección al nombre de ruta que declaramos antes: users.create

Hemos visto en los ejemplos anteriores que podemos enrutar directamente a una vista:

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

También hemos visto que podemos enrutar a un controlador:

```
Route::get('/home', 'HomeController@index')
```

Pero lo cierto es que podemos resolver desde el propio fichero web.php en la misma función de enrutado:

```
Route::any('/pruebita', function(){
    echo "<h1>
        estamos haciendo la respuesta a la petición desde el propio enrutado
    </h1>";
});
```

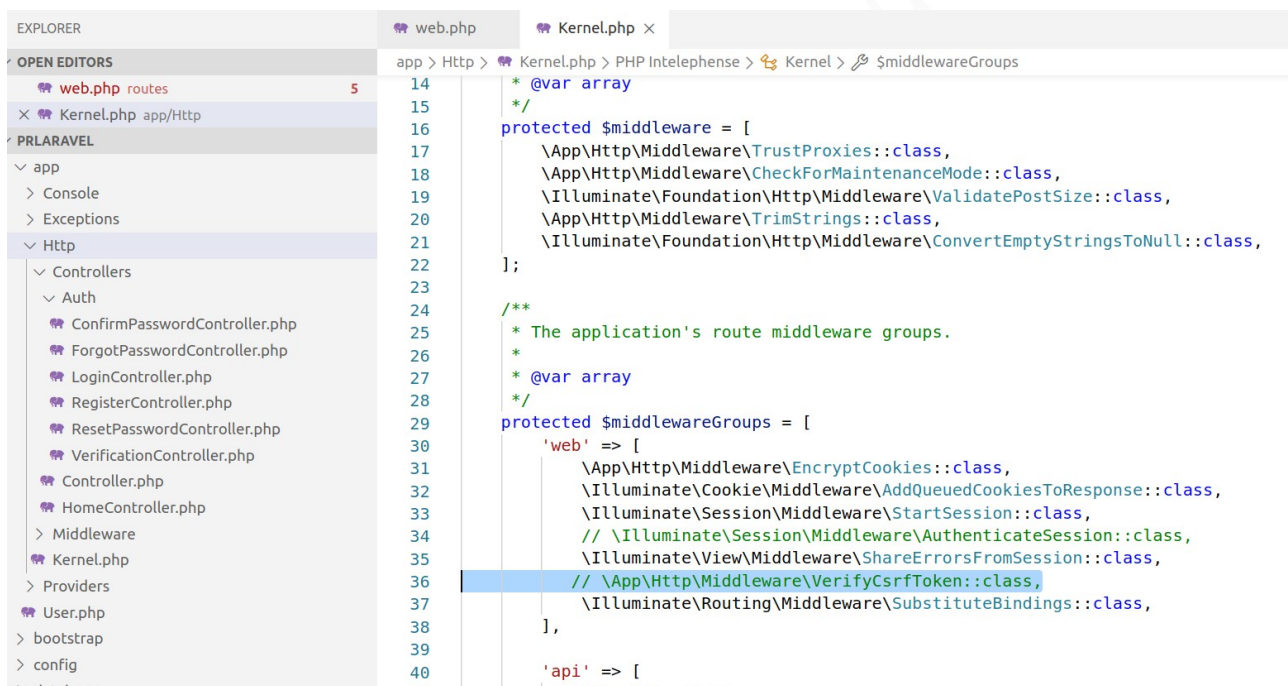
Observar que hemos hecho uso de: `Route::any()` esto significa que cualquier petición que se haga a: /pruebita se resuelve con esta ruta indistintamente del tipo de petición HTTP que sea (da igual que sea GET, POST,...)

Nota: En general tomaremos en todas las prácticas captura de pantalla adicionalmente de la ejecución desde el navegador a lo que se solicite en la actividad

● **Práctica 1:** Modificar el fichero web.php para que las peticiones GET (parecido al ejemplo anterior) al raíz de la aplicación: “/” muestren un mensaje que diga: “Under construction”

Las rutas con peticiones POST tienen protección para el ataque CSRF para hacer pruebas de petición POST en este momento se hace necesario quitar esa protección momentaneamente (**volver a establecerla después de ejecutar la práctica**)

En el fichero: Http/kernel.php se comenta la línea 36 para desactivar la protección CSRF:

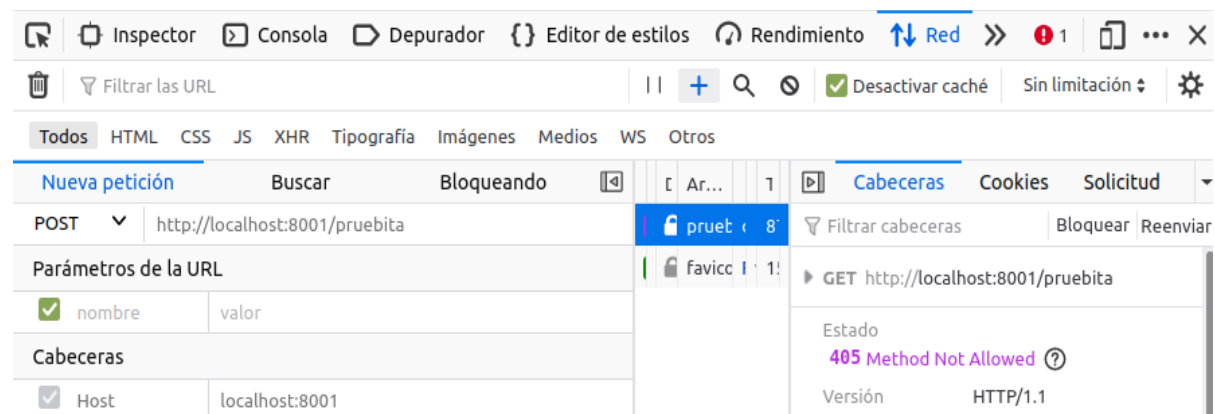


```
14 * @var array
15 */
16 protected $middleware = [
17     \App\Http\Middleware\TrustProxies::class,
18     \App\Http\Middleware\CheckForMaintenanceMode::class,
19     \Illuminate\Foundation\Http\Middleware\ValidatePostSize::class,
20     \App\Http\Middleware\TrimStrings::class,
21     \Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull::class,
22 ];
23
24 /**
25  * The application's route middleware groups.
26  *
27  * @var array
28  */
29 protected $middlewareGroups = [
30     'web' => [
31         \App\Http\Middleware\EncryptCookies::class,
32         \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
33         \Illuminate\Session\Middleware\StartSession::class,
34         // \Illuminate\Session\Middleware\AuthenticateSession::class,
35         \Illuminate\View\Middleware\ShareErrorsFromSession::class,
36         // \App\Http\Middleware\VerifyCsrfToken::class,
37         \Illuminate\Routing\Middleware\SubstituteBindings::class,
38     ],
39     'api' => [
40         // ...
41     ]
42 ];
```

● **Práctica 2:** Modificar el fichero web.php para que las peticiones POST a: /pruebita muestren el mensaje: “se ha ejecutado una petición POST a la dirección: /pruebita ” Probar a hacer la petición POST ¿ muestra lo solicitado ? ¿ qué ocurre si se hace mediante una petición GET ? Volver a reestablecer la protección CSRF y hacer de nuevo la petición POST ¿ qué muestra ahora ?

Nota: Desde Rested se puede enviar la petición POST, así como Postman o cualquier otra utilidad. Incluso desde el inspector de Firefox permite modificar una petición GET a POST y reenviar. **En el inspector de firefox al editar la petición y pasar a post te da la respuesta en el “preview” no en la página renderizada del navegador**

Ejemplo del inspector de firefox editando la petición:



Parámetros en el path y enrutado según expresiones regulares

En ocasiones queremos enrutar teniendo en cuenta parámetros en el path (el clásico: user/id es un buen ejemplo) Para realizar tal acción debemos hacer uso de las llaves: {} ejemplo:

```
Route::get('user/{name}', function ($name) {  
    echo "el parámetro del path recibido es: " . $name;  
    exit();  
})
```

Observar que: `user/{name}` se mapea al parámetro recibido en la función: `function ($name)`

Si únicamente queremos tener en cuenta las rutas bien construidas por una expresión regular podríamos hacer uso de: `where`

```
Route::get('user/{minombre}', function ($minombre) {  
    echo "el parámetro del path: " . $minombre . " está en mayúscula";  
    exit();  
})->where('minombre', '[A-Z]+');
```

● **Práctica 3:** Crear una ruta para TODA petición (ya sea GET, POST, ...) hacia /relatos/numero (recordar que hemos visto una opción para recoger todo tipo de petición) De tal forma que numero deba ser un número y muestre el mensaje: “petición recibida para el parámetro: numero”

● **Práctica 4:** Crear una ruta para el raíz: “/” En una primera implementación mostrará el mensaje: “página raíz de nuestra aplicación” que se resolverá en el propio web.php Haremos una segunda versión de esta actividad en la que redireccionará hacia el controlador y la función pertinente y allí se mostrará un mensaje que indique adicionalmente que se ha respondido desde el controlador

Creando nuestro primer controlador

Los controladores conforman una porcentaje importante de nuestra aplicación. Una vez realizado el enrutado, es en el controlador donde hablamos con el modelo (por ejemplo con todas las clases que reflejan las tablas de nuestra base de datos) y le pasamos la información conveniente a las vistas que se le muestran al usuario

El primer controlador de la aplicación de la tienda debe responder a la solicitud de peticiones a la url raíz: “/” Y tenemos pensado mostrar una lista de productos para que el usuario pueda ver los productos y decidirse o no a comprarlos

Cuando hablamos de los ficheros que nos creó laravel para la autenticación vimos que nos había creado algún controlador (Por ejemplo HomeController) Vamos a echar un vistazo a ese controlador:

```
<?php

namespace App\Http\Controllers;
use Illuminate\Http\Request;

class HomeController extends Controller
{
    /**
     * Create a new controller instance.
     * @return void
     */
    public function __construct()
    {
        $this->middleware('auth');
    }

    /**
     * Show the application dashboard.
     * @return \Illuminate\Contracts\Support\Renderable
     */
    public function index()
    {
        return view('home');
    }
}
```


Lo primero que observamos es que se define el namespace para la clase HomeController

Los namespace son un contenedor que nos permite agrupar nuestro código para darle un uso posterior de esta manera evitamos conflictos de nombre. Por ejemplo, tenemos 2 funciones con el mismo nombre esto generaría un conflicto de nombre pero mediante el uso de namespace damos solución a este problema

Observamos que en este caso coincide la ruta del namespace con la ruta de la carpeta. Esto es una buena política: se recomienda estructurar los namespace por la ubicación en carpetas

El namespace establecido es:

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

Para poder importar la clase HomeController habrá que hacer uso del nombre:

```
App\Http\Controllers\HomeController.php
```

Ahora bien, ¿cómo llamamos entonces a una función de esa clase ?

- Mediante la ruta completa:

```
App\Http\Controllers\HomeController
```

Vamos a comprobarlo. Agregar en HomeController.php el siguiente código:

```
public static function ejecutaprueba()  
{  
    echo "ejecutando código";  
}
```

Ahora en web.php agregamos lo siguiente:

```
Route::get('/prueba', function () {  
    \App\Http\Controllers\HomeController::ejecutaprueba();  
    exit();  
});
```

Ahora si vamos a la ruta: /prueba nos debe aparecer en pantalla: “ejecutando código”

Podemos ver que hemos hecho uso de la ruta completa a la clase HomeController y como ejecutaprueba() es un método estático hacemos uso de los dos puntos: “::” para llamar al método

Otra alternativa más común es mediante el comando use:

```
use \App\Http\Controllers\HomeController;

Route::get('/prueba', function () {
    HomeController::ejecutaprueba();
    exit();
});
```

Vemos que es parecido a un import. Podríamos haber escrito también un alias para el nombre de la clase:

```
use \App\Http\Controllers\HomeController as MiControlador;
```

Con el alias podemos entonces llamar a:

```
MiControlador::ejecutaprueba();
```

Adicionalmente al comando **namespace** y **use** vemos que en este fichero **HomeController** extiende de Controller Esa es la parte fundamental para establecer un controlador

Otra cosa que podemos observar es que está establecido un: **use** para Request (que luego será inyectado en las funciones que queramos tomar la request)

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

Finalmente hay un comando de terminal que nos facilita la creación de un controlador:

```
php artisan make:controller Nombrecontrolador
```

● **Práctica 5:** Crear un controlador llamado: ListarProductos que sea redireccionado en web.php cuando se acceda al raíz: “/” y muestre un mensaje que diga: “Ejecutando el controlador ListarProductos mediante get”. (si la llamada fue get. En el caso de que la llamada fuera post deberá decirlo)

Aún hay código por explicar en el controlador HomeController. Observamos cosas como:

```
/**  
 * Show the application dashboard.  
 *  
 * @return \Illuminate\Contracts\Support\Renderable  
 */
```

Si nos fijamos en la parte de comentarios estamos introduciendo algo de código. En este caso: `@return` le está indicando al ide que sepa que la función devuelve un objeto del tipo: `Renderable`. De esa forma nos puede ayudar al codificar. Lo anterior forma parte de PHPdoc (para documentar php) Debemos tener instalado: PHP Intelephense para que visual studio code entienda ese tipo de anotaciones y nos ayude a codificar

Lo anterior nos introduce las anotaciones:

Anotaciones

Hay diversas anotaciones que se soportan en PHP para obtener documentación. Estas se basan en el estido de Javadoc y podemos encontrarlas en: <http://phpdoc.org/>

Quizás la más importante sea la anotación: @var que nos permite establecer el tipo de la variable

```
@var ["Type"] [$element_name] [<description>]
```

Observamos que damos el tipo de dato primero y luego el nombre de la variable. Adicionalmente se puede dar una descripción de la variable

Veamos un ejemplo:

```
class Foo
{
    /**
     * @var string $name Should contain a description
     * @var string $description Should contain a description
     */
    protected $name, $description;
}
```

En el código anterior estamos diciendo que \$name es de tipo string y documentamos que utilidad tiene ese atributo. Lo mismo ocurre con \$description

● **Práctica 6:** Comprobar que la anotación @var para un objeto permite que el ide con inteliphense nos ayude con los atributos y los métodos

Comenzando con las Vistas en Laravel (Blade)

Supongamos que tenemos en nuestro controlador el siguiente código:

```
public function primos()
{
    $coleccion = collect([1,2,3,5,7,11,13,17,19]);
    return view('listarprimos',compact('coleccion'));
}
```

Ese es un código parecido a lo que nos construyó en su momento laravel para la página de inicio de nuestra aplicación en web.php . Vamos a recordarlo:

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

Así pues en aquel momento se llamaba directamente a una vista (función view()) desde el enrutador (estaba en web.php) y ahora la estamos llamando desde nuestro controlador. En el caso actual: `return view('listarprimos',compact('coleccion'));` nos está diciendo que se va a llamar al fichero: **resources/views/listarProductos.blade.php** y le estamos pasando un **array** con los productos que hemos obtenido de la base de datos: **compact('coleccion')** La orden compact está descrita en el manual de php: <https://www.php.net/manual/es/function.compact.php>

Nosotros le pasamos el nombre de las variables y **compact** nos construye un array clave/valor donde la clave es el nombre de la variable y el valor aquello que contenga la variables

La función view() espera que le demos como primer parámetro el nombre de la vista que queremos y como segundo parámetro un array que contenga los datos que queremos pasarle a la vista. Debemos tener en cuenta que \$productos es un objeto especial del tipo: Collection que es muy útil en Laravel. A nosotros nos interesa que la vista tenga esa “colección” de productos así que debemos pasarla entera dentro del array de datos. Es por eso que recurrimos a compact()

Vamos a copiar el fichero: **resources/views/welcome.blade.php** a **resources/views/listarProductos.blade.php**

¿Qué es Blade?

Según la documentación de laravel:

“Blade es un motor de plantillas simple y a la vez poderoso proporcionado por Laravel. A diferencia de otros motores de plantillas populares de PHP, Blade no te impide utilizar código PHP plano en sus vistas. De hecho, todas las vistas de Blade son compiladas en código PHP plano y almacenadas en caché hasta que sean modificadas, lo que significa que Blade no añade sobrecarga a tu aplicación. Los archivos de las vistas de Blade tienen la extensión **.blade.php** y son usualmente almacenados en el directorio **resources/views**”

tomado de: <https://documentacion-laravel.com/blade.html>

Directiva @foreach

En el fichero que hemos copiado vamos a hacer que recorra los datos que le hemos enviado desde el controlador. Para ellos vamos a hacer uso de la sentencia: **@foreach**

(dejamos los estilos por si luego queremos hacer uso de ellos pero borramos el contenido de body por el nuevo para no ensuciar tanto la salida en pantalla)

```
</style>
</head>
<body>

    @foreach ($coleccion as $primo)
        <p> primo: {{ $primo }} </p>
    @endforeach

</body>
</html>
```

Observamos que Blade usa directivas mediante la arroba: “@” y en este caso estamos usando un bucle for each. Luego, como en otros gestores de plantillas, vemos que se usa los brackets para mostrar el contenido de una variable: **{{ \$primo }}** Así se recorre la colección: \$coleccion que hemos recibido del controlador y vamos tomando un \$producto en cada iteración

● **Práctica 7:** Reproducir la vista descrita. Crear una tabla html por cada primo con un encabezado en la tabla que nos diga que campo estamos visualizando.

Hemos visto que con las llaves podemos visualizar el contenido de una variable. Realmente se puede ejecutar el código PHP que se quiera dentro de las llaves: {{ }}

● **Práctica 8:** Agregar al comienzo de la vista el mensaje(sustituye por la hora/día actual):
Son las: 17:53 del día: 29-11-2020
Nota: buscar información y usar la función PHP date()

El uso de brackets{{}} para ejecutar código PHP puede tener la desventaja de las ocasiones que no queramos mostrar nada en pantalla. Simplemente hacer operaciones PHP (por ejemplo asignar a una variable alguna información para usarlo más adelante) Para esos casos podemos usar la directiva: @php

```
<body>
    @php
    $dato = time()
    @endphp
```

Desde el 1-01-1970 han pasado: {{ \$dato }} segundos

```
</body>
```

● **Práctica 9:** El comando sleep() en php permite pausar la ejecución la cantidad de segundo especificada como parámetro. Modificar el ejemplo anterior para que lo muestre 3 veces con una espera de 1 segundo entre una iteración y la siguiente, mostrando de forma actualizada la información de los segundos desde 1970

Directivas @if, @else, @elseif

Podemos hacer uso de las sentencias: **if** que acostumbramos en php mediante las directivas @if, @else, @elseif y @endif

Veamos un ejemplo:

```
<body>
@php
$cantidad = count($productos);
@endphp

la cantidad es: {{ $cantidad }}
@if($cantidad %2 == 0 && $cantidad > 0)
    la cantidad de productos es par y positiva
@elseif( $cantidad > 0 )
    la cantidad de productos es impar positiva
@else
    este caso debiera ser imposible ( cantidad negativa de productos)
@endif
</body>
```

● **Práctica 10:** Generar una lista de números aleatorios de 0 a 100 en el controlador. Desde nuestra plantilla blade mostraremos primero la lista de números obtenidos menores de 50 y un poco más abajo en la página los mayores que 50. Hacer uso de las directivas @if para que al mostrar aquellos que sean mayores de 50

Directivas @isset, @empty

Dada la frecuencia de isset() y empty() en php tenemos su equivalente como directiva blade. Tienen el mismo efecto que si hubiéramos puesto el correspondiente if:

@if(isset(\$records)) equivale a: @isset(\$records)

La forma de finalizar estas directivas es con su correspondiente end:

@endisset, @endempty

Otras directivas: @for, @while

Podemos usar los bucles for habituales:


```
@for ($i = 0; $i < 10; $i++)  
    The current value is {{ $i }}  
@endfor
```

Igualmente con while:

```
@while (true)  
    <p>I'm looping forever.</p>  
@endwhile
```

Hay muchas más directivas para una referencia completa: <https://laravel.com/docs/5.8/blade>

Es habitual que cuando hacemos un sitio web queramos que las diferentes páginas respondan a una misma estética, introduciendo únicamente los cambios pertinentes, para eso usaremos:

-  **Práctica 11:** Enviar en un textarea una lista de palabras separadas por comas. Mostrar en una lista html esas palabras recibidas (una palabra por cada de la lista) convertidas todas a mayúsculas. Para ello se usará el bucle: @for (cuidado! No el foreach) Observar que eso implicará “contar” el número de elementos que tiene la colección de palabras

Mostrar imágenes desde fichero publico

Las imágenes y ficheros de acceso público pueden alojarse en:

`carpetabaseproyecto/public`

(cuidado... no es dentro de app es una carpeta llamada public que cuelga directamente de la carpeta base)

Así pues, si creamos una carpeta img su ruta sería:

`carpetabaseproyecto/public/img`

Como public es la carpeta raíz para el usuario que accede a nuestra aplicación las rutas de acceso a ella serán del tipo: ``

● **Práctica 12:** Ubicar imágenes en la carpeta descrita para las imágenes que quieras mostrar (mínimo 5). Hacer que se visualicen en el navegador las imágenes en nuestra vista

nota: Poner el renderizado que se considere más apropiado.

Laravel Request

Para que laravel nos inyecte un objeto Request: `Illuminate\Http\Request` basta que lo definamos como un parámetro a recibir en el método del controlador que queramos:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
class UserController extends Controller
{
    public function store(Request $request)
    {
        $name = $request->input('name');
        //
    }
}
```

Como vemos, declaramos: `Request $mivariable` y luego usamos la función `input()` para tomar los parámetros: `$mivariable->input('id')` Es importante que tengamos en cuenta que este comportamiento es el mismo indistintamente si es GET, POST o cualquier tipo de consulta HTTP

Sesiones en Laravel

El comportamiento de las sesiones lo podemos establecer en:

carpetabaseproyecto → config → session.php

- En la parte que dice: `'driver' => env('SESSION_DRIVER', 'file')` podemos decirle que guarde la sesión en database, cookie, file, etc. Simplemente cambiando: 'file' por lo que queramos

- la cantidad de tiempo que se mantiene ociosa la sesión antes de cerrarse:

`'lifetime' => env('SESSION_LIFETIME', 120)`

En la línea anterior contemplamos 120 minutos sin interacción del usuario antes de cerrar la sesión.

Laravel crea automáticamente siempre una sesión y se puede acceder directamente al helper: `session()` para acceso a las variables. Si se prefiere también se puede hacer desde Request:

```
//mediante el helper:
$idCarrito = session("idCarrito"); //o también: $idCarrito = session()→get("idCarrito");

//mediante Request
public function index(Request $request) {
    $idCarrito= $request->session()->get('idCarrito');
}
```

Vemos que: `$request→session()` hace el mismo efecto que `session()` Entonces ¿ cuál es la diferencia ? Request tiene el comportamiento esperado, ya que podemos acceder a la información de la sesión en la que está englobada la solicitud: `$request→session()` Por otro lado: `session()` es un helper que ha creado Laravel para hacer las cosas más sencillas

¿Qué es un helper para Laravel?: Una función global que tenemos disponible en cualquier parte de la aplicación. Así, sin necesidad de tener un Request ni ninguna otra cosa siempre podemos acceder a la información almacenada en la sesión simplemente escribiendo: `session()`

- Saber si existe un elemento en la sesión:

```
session()->has('usuario');//devuelve true si está presente y no es nulo
```

- Guardar en la sesión (se puede hacer mediante Request o el helper session)

```
$request->session->put('key' , 'value');  
session(['key' => 'value']); //también vale: session()->put('key','value');
```

En el código anterior vemos que el helper session() soporta que se le pase un array clave/valor para almacenar en sesión la información deseada.

- Tomar todos los datos de la sesión:

```
$data = session()->all();
```

- Borrar un elemento de la sesión:

```
session()->forget('orderId');
```

- Y si queremos obtener el dato y borrarlo a la vez del array de session:

```
$value = $request->session()->pull('key', 'default');
```

- Borrar todos los elementos de la sesión:

```
$request->session()->flush();
```

- Y si queremos borrar todos los datos de la sesión y tener un nuevo identificador de sesión:

```
$request->session()->flush();  
$request->session()->regenerate();
```

- si queremos borrar la información de usuario de la sesión:

```
//clear the authentication information in the user's session  
auth()->logout();
```

En el código anterior se ha introducido otro helper de Laravel: **auth()** Este helper guarda la información del usuario

● **Práctica 13:** Crear un formulario que envíe nombres de colores en cada ejecución del usuario. Obtendrá por respuesta una página con la lista de colores que ha ido introduciendo (usar session() para almacenar la lista de colores) (es un formulario post tendremos que tener en cuenta @csrf leer más abajo)

Para la última acción (crear el formulario post) es necesario poner una anotación: @csrf Veamos de que estamos hablando

@CSRF en los formularios

CSRF es un ataque: falsificación de petición de sitios cruzados. En el que comandos no autorizados son transmitidos por un usuario (sin éste saberlo) que nuestro sitio confía. Esto ocurre por ejemplo, si un usuario que se ha validado correctamente envía una petición que la página atacante dirige hacia nosotros sin que el usuario lo sepa. Como el usuario está correctamente autenticado, nosotros aceptamos la petición y el ataque tiene lugar. Veamos un ejemplo:

Un ejemplo muy clásico se da cuando un sitio web, llamémoslo "example1.com", posee un sistema de administración de usuarios. En dicho sistema, cuando un administrador se conecta y ejecuta el siguiente REQUEST GET, elimina al usuario de ID: "63":
<http://example1.com/usuarios/eliminar/63>

Una forma de ejecutar la vulnerabilidad CSRF, se daría si otro sitio web, llamemos "example2.com", en su sitio web añade el siguiente código HTML:

```

```

Cuando el usuario administrador (conectado en example1.com) navegue por este sitio atacante, su navegador web intentará buscar una imagen en la URL y al realizarse el REQUEST GET hacia esa URL eliminará al usuario 63.

Laravel por defecto genera un token por cada usuario del sistema y ese token es el que utiliza el middleware: **VerifyCsrfToken** para verificar si una petición es legítima

Cuando generemos un formulario, se debe incluir un campo: @csrf para que el middleware valide el token:

```
<form method="POST" action="/profile">  
    @csrf  
    ...  
</form>
```

Hemos nombrado los middleware ¿y qué son ?

Los Middleware proveen un mecanismo eficiente para el filtro de peticiones HTTP que ingresen a tu aplicación. Es un puente entre la solicitud y la respuesta que ejecuta un filtrado. Por ejemplo, Laravel incluye un middleware que permite verificar si un usuario está autenticado

cuando acceda a tu aplicación. Si el usuario no lo estuviera, el middleware lo redirigiría a la pantalla de login. Y por el contrario, si lo estuviera, el middleware permitiría el acceso a la aplicación

● **Práctica 14:** Una forma fácil de visualizar el token csrf es mediante: `{{ csrf_token() }}`
Introducir en la práctica 12 ese código y comprobar que está activo.

● **Práctica 15:** Crear un formulario POST Con los datos de un posible usuario (nombre, edad, gustos, etc) En cada ejecución de este formulario se le muestra al usuario la información almacenada del usuario en session() Observar que si se envía el formulario sin rellenar algún campo, se mantendrá la información anterior respecto a ese campo

Gestión de Ficheros

Con laravel la gestión de ficheros es muy simple. Crearemos unos discos: “disk” que hacen referencia al sitio donde queremos los ficheros (sistema de ficheros local, amazon, etc) y tendremos una capa de abstracción: Storage que nos permite hacer todas las gestiones de ficheros de igual forma indistintamente del sistema final de almacenaje (local, ftp, amazon, ...)

Para crear un disco podemos seguir fácilmente la información oficial.

Tenemos toda la información en la página oficial: <https://laravel.com/docs/8.x/filesystem>

Como el disco por defecto es “local” apuntando a la carpeta: storage de nuestra aplicación podemos trabajar directamente en local sin crear nuevos discos

Vamos a suponer que tenemos los datos en un fichero csv que esté almacenado en la carpeta storage. Entonces el siguiente código nos permite recuperar todo el contenido del fichero:

```
function leerFichero(string $nombrefichero){  
  
    if (($open = fopen(storage_path() . "/" . $nombrefichero, "r")) !== FALSE) {  
  
        while (($data = fgetcsv($open, 1000, ",")) !== FALSE) {  
            $contenido[] = $data;  
        }  
  
        fclose($open);  
        return $contenido;  
    }  
    return null;  
}
```

Las funciones fopen() y fgetcsv() son de php La parte que introduce laravel es la función: storage_path() .

Vamos a ver algunas partes de lo anterior:

`fgetcsv($open, 1000, ",")` Como podemos ver abrimos (`fopen`) el fichero para lectura (“r”) y vamos tomando líneas (observar que el parámetro `length: 1000` no es obligatorio. Pero de ponerlo debe ser un tamaño mayor que la línea más larga) También vemos que hemos establecido el carácter de separación de campos (`delimiter`) siendo la coma: “,”

Lo que lee `fgetcsv()` es una línea del fichero en cada ejecución (parseando según el delimitador csv etc) y termina devolviendo un array con la forma correspondiente al csv leído

También vemos que devuelve `false` cuando finaliza el fichero

Cómo es habitual debemos abrir y cerrar correctamente un fichero para trabajar con él y no dejar ficheros abiertos al terminar, para evitar gasto de recursos innecesario y posibles inconsistencias entre lo que tenemos en ram y lo guardado en fichero: `fopen()`, `fclose()`

Como dijimos antes lo que introduce laravel es `storage_path()` que nos permite tener un apuntador directo a la ruta de almacenaje (el directorio `storage` es su ubicación por defecto)

Sabemos que por MVC lo lógico es que la función que hemos puesto antes forme parte de una clase del modelo (el manejo de los datos persistentes pertenecen al modelo)

● **Práctica 16:** Crear un fichero con nombre y dirección de correo por fila (en formato csv) almacenado en Storage Leer el fichero y mostrarlo en pantalla

Crear un directorio y listarlo

`Storage::makeDirectory()` nos crea un directorio (si ya existe no hace nada). Para verlo en funcionamiento, supongamos que tenemos guardada en una variable de sesión: `nombreusuario`. Entonces crear un directorio para ese usuario es:

```
public function crearListarCarpetaUsuario(Request $request){
    $nombreusuario = $request->session()->get('nombreusuario');
    Storage::makeDirectory("/".$nombreusuario , 0755, true);
    $ficheros = Storage::allFiles("/".$nombreusuario);
    return view('listarficheros',compact('ficheros'));
}
```

En el código anterior se crea una carpeta con: `$nombreusuario` en la ubicación:

`directoriaravel/storage/app/$nombreusuario`

Observar que **la ubicación es: storage/app** NO es directamente en storage a diferencia del ejemplo anterior de uso de `storage_path()`

Vemos que le ha asignado todos los permisos al propietario y lectura y ejecución al resto: 0755 . También vemos que podemos crear de forma recursiva una ruta completa que no exista:

`Storage::makeDirectory("/carpetanoexiste1/carpetanoexiste2" , 0755, true);` gracias al parámetro `true` que crearía primero: `carpetanoexiste1` y luego anidaría dentro de esa carpeta la siguiente: `carpetanoexiste2`

Para listar los ficheros que hay en una carpeta usamos: `Storage::allFiles` Nosotros le pasamos la dirección de la carpeta y nos devuelve un array con la lista de ficheros. Finalmente en el ejemplo vemos que hemos enviado esa lista de ficheros a la vista para mostrárselos al usuario:

```
return view('listarficheros',compact('ficheros'));
```

● **Práctica 17:** Crear un formulario que se introduzca un nombre y cree un directorio en storage con ese nombre

Almacenar un fichero recibido por un formulario

Veamos primero el formulario:

```
<form action="/fileupload" enctype='multipart/form-data' method="post">
    @csrf
    <label for="fichero">sube fichero</label>
    <input type="file" name="myfile" id="fichero">
    <input type="submit" value="Subir">
</form>
```

En el ejemplo anterior se ha puesto un name diferente del id para que distingamos mejor.

Es importante que especifiquemos enctype para que podamos subir el fichero. Pondremos

`enctype='multipart/form-data'` habitualmente cuando queramos subir fichero

Observar: `@csrf` Laravel nos protege de ataques csrf Para que nuestros formularios sean aceptados pondremos en nuestros formularios siempre esa anotación, en otro caso laravel los rechazará

Observar finalmente que el name del input file es: `name="myfile"` ese nombre será fundamental en el controlador

Una ruta posible en web.php para el formulario anterior:

```
Route::post('/fileupload', 'App\Http\Controllers\GestionarFichero@subir');
```

Vemos que se ha mapeado el atributo “action” del formulario anterior: /fileupload al método: subir() de la clase controller: GestionarFichero

Pues bien, Recoger el fichero enviado por el formulario en el controlador, será tan sencillo como usar el name puesto en el formulario (myfile en este caso): `$request->myfile`

Veamos ejemplo completo:

```
public function subir(Request $request){

    $file = $request->myfile;
    echo 'File Name: '.$file->getClientOriginalName(); //Display File Name
    echo '<br>';

    echo 'File Extension: '.$file->getClientOriginalExtension(); //Display File Extension
    echo '<br>';

    echo 'File Real Path: '.$file->getRealPath(); //Display File Real Path
    echo '<br>';

    echo 'File Size: '.$file->getSize(); //Display File Size
    echo '<br>';

    echo 'File Mime Type: '.$file->getMimeType().'<br>'; //Display File Mime Type

    $nombrefichero = $file->getClientOriginalName();
    $path = $file->storeAs("/", $nombrefichero);
    echo $path;
}
```

En el código anterior observamos que recibimos en: `$request->myfile` el fichero porque en el formulario el atributo name era ese mismo: myfile

También vemos algunos métodos que nos dan información sobre el fichero: el nombre con el que fue subido, su extensión, el path, el tamaño, el tipo mime al que responde

Finalmente vemos como lo podemos almacenar allí donde queramos:

```
$path = $file->storeAs("/", $nombrefichero);
```

Así, vemos que el método: **storeAs()** nos permite decirle donde lo queremos guardar

● **Práctica 18:** Crear un fichero con nombre y dirección de correo por fila (en formato csv) almacenado en Storage Leer el fichero y mostrarlo en pantalla

Descargar un fichero

Veamos la forma más sencilla mediante un ejemplo de descarga desde un controlador:

```
class GestionarFichero extends Controller
{
    public function descargar(Request $request){
        return response()->download(storage_path('app/paco/mifichero.txt'));
    }
}
```

En el ejemplo anterior vemos que estamos usando: `response()->download()` para ejecutar la descarga. Ese método es el que hará la descarga del fichero que le indiquemos. Observar que hemos usado: `storage_path()` para que nos de la dirección donde está storage y concatenarle el resto de la ruta.

● **Práctica 19:** Mostrar en una página una lista de ficheros de una carpeta en storage. Cuando se pulse en el nombre del fichero se descargará

Borrar un fichero

Podemos validar que el fichero existe mediante: `Storage::exists()` y borrarlo mediante `Storage::delete()` (todos estos comandos pueden tener una referencia a un disco en concreto: `Storage->disc("local")::delete()` etc)

Ej:

```
if( Storage::exists($fichero))  
    Storage::delete($fichero);
```

● **Práctica 20:** Continuando con el ejemplo anterior crear una opción para borrar los ficheros listados

Anexo: Visual studio debug

Extensiones:

laravel extension pack

php inteliphense

php extension pack (el debug básicamente)

php debug (el extensión pack debiera incorporarlo)

Veamos para el caso de una instalación linuxbrew (en otro caso será similar)

Aprovecharemos que linuxbrew ya tiene pecl:

```
pecl install xdebug
```

Al finalizar el comando, en la salida en pantalla se observa la ruta donde queda instalado: xdebug.so (necesitaremos esa ruta para ponerla en el fichero php.ini)

Ejecutando una página php que incorpore phpinfo() sabremos exactamente

donde está nuestro fichero de php.ini. También en el volcado que hace se ve si está instalado xdebug

Allí pondremos las líneas:

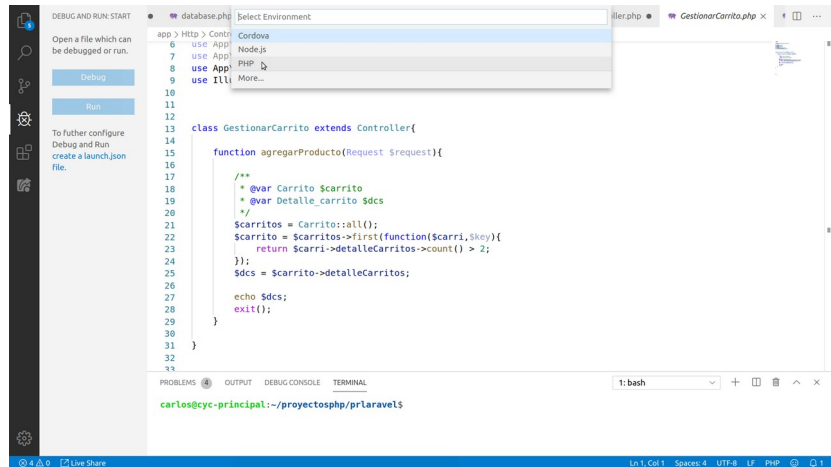
(sustituir **20180731** por lo que corresponda También podemos cambiar el puerto al 9003 si lo preferimos para que haya coincidencia con la extensión php debug de vscode.)

agregar al final del fichero php.ini:

```
zend_extension = /home/linuxbrew/.linuxbrew/lib/php/pecl/20180731/xdebug.so
xdebug.mode = debug
xdebug.start_with_request = yes
xdebug.client_port = 9000
```

Ahora en vscode tenemos que tener instalada la extensión php debug

Desde la parte de debug le diremos que cree una nueva configuración (un nuevo fichero launch.json) aparecerá un desplegable y allí elegimos: **php**. Nos generará la configuración precisa. Observar allí el puerto. Debe haber coincidencia con el que hemos puesto en php.ini (por ejemplo el 9000 de antes)



(aparece a la derecha opción para agregar para php etc)

El json generado por el ide es apropiado

→ observar que se tiene xdebug con la palabra: STUDIO puesta en la configuración de php

basta con arrancar el servidor web: php artisan serve

y luego desde visual studio lanzar el debugger (habiendo establecido un breakpoint sobre la línea que queramos)

Anexo: Restaurar proyecto desde github

El gitignore de laravel incluye la carpeta Vendor (las dependencias) Para restaurar un proyecto podemos hacerlo con composer:

```
composer install
```

En el caso que haya algún problema con actualizaciones necesarias:

```
composer update
```

```
composer install
```

En el caso de que se hayan generado dependencias de npm (carpeta node_modules) también hay que restaurarlo:

```
npm install
```

Finalmente hay que tener en cuenta que hay un fichero de configuración que para muchas aplicaciones es fundamental: .env En este fichero, por ejemplo, se detallan la base de datos etc. Así ahí puede haber información de usuarios/password de conexión. Se considera un problema de seguridad subir ese fichero a github así que el .gitignore puede estar establecido para que no sea subido. En ese caso hay que crearlo. Para que no sea todo el trabajo desde cero se puede aprovechar que siempre tenemos uno de ejemplo: .env.example

Podemos copiar el .env.example a .env y luego modificarlo

Veamos las modificaciones:

En la parte de base de datos (si corresponde) reemplazar por el sgbd, nombre ddbb, usuario y pass:

```
DB_CONNECTION=mysql
```

```
DB_HOST=localhost
```

```
DB_PORT=3306
```

```
DB_DATABASE=laravel_project
```

```
DB_USERNAME=root
```

```
DB_PASSWORD=secret
```

Se precisa también una clave de aplicación que de forma automática se genera al crear el proyecto laravel en el fichero .env como nosotros lo hemos clonado de github vamos a tener que recrear la clave:

```
php artisan key:generate
```

Es posible que queden algunas modificaciones (especialmente si trabajamos con bases de datos y hay que crear la estructura de la base de datos mediante las migraciones de laravel) Pero lo básico ya está detallado y soluciona la mayoría de los problemas posibles