TEMA

6

2ºDAW

Framework Laravel

Para realizar estos apuntes se ha utilizado de referencia el libro: **https://ajgallego.gitbooks.io/laravel-5/content/**

1 Introducción a Laravel

Laravel es un *framework* de código abierto para el desarrollo de aplicaciones web en PHP 5 que posee una sintaxis simple, expresiva y elegante.

Laravel facilita el desarrollo simplificando el trabajo con tareas comunes como la autenticación, el enrutamiendo, gestión 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.

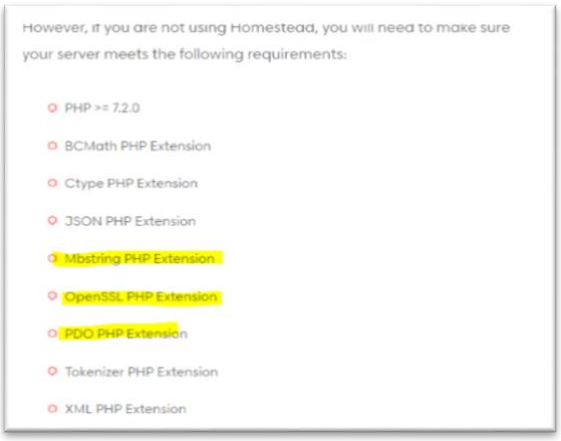
También posee algunas desventajas

- Curva de aprendizaje
- Desconocimiento del nucleo
- Más abstracción y menos rendimiento

2 Instalación

Documentación instalación del Framework https://laravel.com/docs

1. Asegurar que ruta de PHP se encuentre en la variable de entorno $PATH y comprobar que tienes la versión correcta de la versión de php por la terminal



php -v.

2. Comprobar las extensiones de PHP. Las subrayadas en amarillos son las más importantes

Php -m (extensiones activadas) Si algunas de las extensiones no están activas ve a tu php.ini y descomenta la línea de tu extensión. (En el Wamp puedes ir al icono y buscarlas para activarlas dentro extensiones de php) Si no tuviésemos instalada las extensiones tendríamos que buscar su fichero dll en internet y descargarla en la carpeta **ext de php. Después añadiríamos la línea en el php.ini**

3. Instalar Composer: https://getcomposer.org/download/ para el control de dependencias de php

   a. **Windows**

- Descargar el fichero EXE

   b. **MAC**

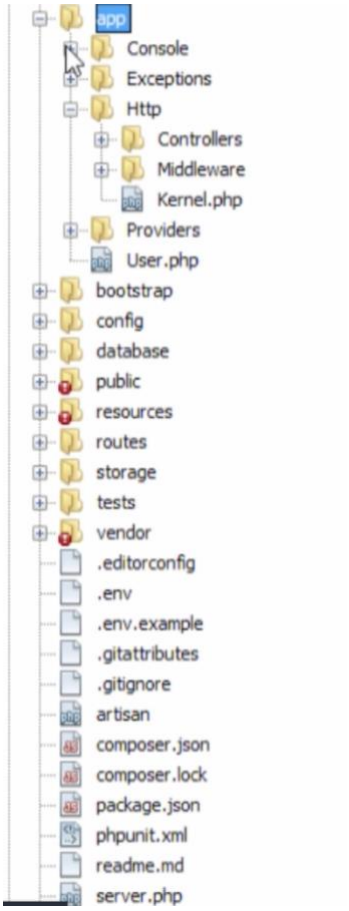
- Descarga **composer.phar** de la web - Moverlo a la carpeta: **sudo mv composer.phar /usr/local/bin/** - Cambiar permiso: **sudo chmod 755 /usr/local/bin/composer.phar** - Permitir al bash ejecutarlo: **nano ~/.bash_profile** - **(**Añade la línea: **alias composer="php /usr/local/bin/composer.phar")** - Ejecutamos : **source ~/.bash_profile** - Comprobamos: **composer --version**

4. **Instalar Laravel via Composer**

Acceder a la ruta de vuestro servidor web donde guardáis los proyectos y ejecutar **Composer create-project laravel/laravel** *nombre-proyecto* **"5.8.*" --prefer-dist**

3 Estructura del proyecto

Al crear un nuevo proyecto de Laravel se nos generará una estructura de carpetas y ficheros para organizar nuestro código. Vamos a explicar brevemente las carpetas que más utilizaremos y las que mejor tendremos que conocer:



`app` – Contiene el código principal de la aplicación. Esta carpeta a su vez está dividida en muchas subcarpetas que analizaremos en la siguiente sección.

`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** de nuestro proyecto. Dentro de ella podemos encontrar a su vez tres carpetas: ***factores***, ***migrations*** **y** ***seeds***.

`public` – Es la única carpeta pública, la única que debería ser **visible** en nuestro 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*** **y** ***lang*****:**

> **`resources/views`** – Este directorio contiene las vistas de nuestra 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`
>
> **`resources/lang`** – En esta carpeta se guardan archivos PHP que contienen arrays con los textos de nuestro sitio web en diferentes lenguajes, solo será necesario utilizarla en caso que se desee que la aplicación se pueda traducir.

`bootstrap` – En esta carpeta se incluye el código que se carga para procesar cada una de las llamadas a nuestro proyecto. Normalmente no tendremos que modificar nada de esta carpeta.

`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. Normalmente tampoco tendremos que tocar nada dentro de esta carpeta, unicamente se suele acceder a ella para consultar los logs.

`tests` – Esta carpeta se utiliza para los ficheros 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. Esta carpeta tampoco la tendremos que modificar, ya que 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 ficheros muy importantes y que también utilizaremos:

> `.env` – Este fichero ya lo hemos mencionado en la sección de instalación, 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 fichero debería estar en el `.gitignore`.
>
> `composer.json` – Este fichero 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.

4 Rutas básicas

Las rutas de nuestra aplicación se tienen que definir en el fichero `routes/web.php`. Este es el punto centralizado para la definición de rutas. Las rutas, en su forma más sencilla, pueden devolver directamente un valor desde el propio fichero de rutas, pero también podrán generar la llamada a una vista o a un controlador.
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 que se pueden utilizar son

- GET
- POST
- PUT
- DELETE

```php
// Ruta principal
Route::get('/', function () {
    //return view('welcome');
    return "HOLA MUNDO";
});

//Utilizar algunos de los 2 recursos get o post
Route::match(['GET', 'POST'], '/uno/dos', function()
{
    return 'iHola mundo!';
});

// Añade un parametro a la ruta. Navegador: localhost/.../usuario/25
Route::get('usuario/{nombre}', function($nombre)
{
    return 'Usuario: '.$nombre;
});

// Utilizar expresiones regulares para validar los parametros
// Utilizamos un parametro edad opcional (?)
Route::get('usuario2/{edad?}', function($edad=0)
{return 'Usuario '.$edad;})
->where('edad', '[0-9]+');
```

En el siguiente apartado vamos a ver como separar el código de las vistas y mas adelante añadiremos los controladores.

5 Vistas básicas

Las vistas se almacenan en la carpeta `resources/views` como ficheros PHP, y por lo tanto tendrán la extensión blade`.php`. Contendrán el código HTML de nuestro 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, como veremos más adelante) para presentar los datos de entrada como un resultado HTML.

Definir vistas

A continuación se incluye un ejemplo de una vista simple, almacenada en el fichero `resources/views/home.blade.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
<html>
    <head>
        <title>Mi Web</title>
    </head>
```

```
    <body>
        <h1>¡Hola <?= $nombre; ?>!</h1>. //Notación PHP
          <h1>¡Hola {{$nombre}}!</h1>. //Notación blade
    </body>
</html>
```

Las vistas se pueden organizar en subcarpetas dentro de la carpeta `resources/views`, por ejemplo podríamos tener una carpeta `resources/views/user` y dentro de esta todas las vistas relacionadas, como por
ejemplo `login.blade.php`, `register.blade.php` o `profile.blade.php`.
En este caso para referenciar las vistas que están dentr**o 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/register.blade.php` la cargaríamos de la forma:

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

<u>Referenciar y devolver vistas</u>

```
// Ruta principal vamos llamar a la vista home con el valor de la variable Javi
Route::get('/', function()
{
    return view('home',['nombre'=>'Javi']);
        });

//llamamos a la vista home con el valor de los 2 parametros
Route::get('usuario3/{nombre}/{edad}', function($nombre, $edad)
{
    return view('home2', ['nombre'=>$nombre,'edad'=>$edad]);})
->where(array('nombre' => '[A-Za-z]+','edad' => '[0-9]+'));

//Añade dos parametros opcionales (?) a la ruta y le manda su valor a la vista
Route::get('usuario4/{nombre?}/{edad?}', function($nombre="Ninguno",$edad=18)
{
    return view('prueba.home3',compact('nombre','edad'));

});

// Utilizar With en lugar de array para mandar los datos a la vista
Route::get('usuario5/{nombre?}/{edad?}', function($nombre="Ninguno",$edad=18)
{
    return view('home2')->with('nombre', $nombre)
                        ->with('edad', $edad);
});
```

6 Consola artisan

Laravel incluye un interfaz de línea de comandos llamado *Artisan*. Esta utilidad nos va a permitir realizar múltiples tareas necesarias durante el proceso de desarrollo o despliegue a producción de una aplicación, por lo que nos facilitará y acelerará el trabajo.
Para ver una lista de todas las opciones que incluye Artisan podemos ejecutar el siguiente comando en un consola o terminal del sistema en la carpeta raíz de nuestro proyecto:

```
php artisan list
# O simplemente:
php artisan
# Ayuda para un comando
php artisan help migrate
# para generar la clave de encriptación
php artisan key:generate
# para ver u listado de todas las rutas
php artisan route:list
# generar código para un controlador
php artisan make:controller UsuarioController
```

7 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 ficheros 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 fichero de rutas o desde un controlador. Es decir, utilizaremos `view('home')` tanto si el fichero 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 nos brinda utilidades que nos ayudarán en el diseño y modularización de las vistas.

Mostrar datos y estructuras de control

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

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

El método más básico que tenemos en *Blade* es el de mostrar datos y llamar funciones , para esto utilizaremos las llaves dobles ({{ }}) y dentro de ellas escribiremos la variable o función con el contenido a mostrar.

Views/usuarios/listado.blade.php

```
{{-- Mostra variables y funciones  --}}

<h1>Hola {{ $name }}.</h1>
<p>La hora actual es {{ time() }}.</p>

{{-- Mostrar un dato solo si existe  --}}

<?=   isset($name) ? $name : 'No existe el nombre'   ?>  PHP SIN BLADE
{{ $name   ??   'No existe el nombre' }}  NOTACIÓN BLADE

{{-- Sentencia isset  --}}
@isset($edad)
        <p>Existe edad </p>
@endisset
{{-- Sentencia si  --}}

@if( count($users) === 1 )
     <p>Solo hay un usuario!</p>
@elseif (count($users) > 1)
    <p> Hay muchos usuarios!</p>
@else
    <p> No hay ningún usuario :(</p>
@endif
{{-- Sentencia si con isset --}}
@if (@isset($asd))
          <p>Existe</p>
@else
          <p>No Existe</p>
 @endif
```

```
{{-- Bucles  --}}

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

<?php $contador=1; ?>
@while ($contador<50)
     @if( $contador %2 ==0 )
     <p>Numero Par {{$contador}}<br/> <p>
@endif
     <?php $contador++; ?>
@endwhile

<h1>{$titulo}</h1>
@foreach ($listado as $user)
     <p>Usuario {{ $user}} <p>
@endforeach
```

Ademas de estas estructuras de control *Blade* define algunas más que podemos ver directamente en su documentación: http://laravel.com/docs/5.7/blade

En nuestro fichero de rutas (web.php) tendríamos lo siguiente para mostrar el listado de usuarios. Tendriamos la vista (**listado.blade.php**) puesta en la **carpeta usuarios**

```
Route::get (‘/listado-usuarios, function(){
            $titulo = “Listado de usuarios”;
            $listado= [‘Antonio’, ‘Pepe’, ‘María’];
            Return view(‘usuarios.listado’)
                        ->with (‘titulo’, $titulo);
                        ->with(‘listado’,$listado);
});
```

Incluir una plantillas con @include

Por ejemplo creamos la carpeta **/views/includes y** creamos los ficheros

- header.blade.php
- footer.blade.php

Y en nuestro fichero **listado.blade.php**

```
@include(includes.header)
…….
@include(includes.footer)
```

Además podemos pasarle un array de datos a la vista a cargar usando el segundo parámetro del método `include`:

```
@include('carpeta.view_name', ['some'=>'data'])
```

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

Layouts

Blade también nos permite la definición de *layouts* para crear una estructura HTML base con secciones que serán rellenadas por otras plantillas o vistas hijas. Por ejemplo, podemos crear un *layout* con el contenido principal o común de nuestra 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 de nuestro sitio web, lo que nos permite que en el resto de plantillas no tengamos que repetir todo este código.

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

```
<html>
    <head>
        <title>Titulo - @yield('titulo')</title>
    </head>
    <body>
        @section('header')
            <h1>Cabecera de la Web</h1>
        @show
        <div class="container">
            @yield('content')
        </div>
          @section('footer')
            <p>Pie de página</p>
           @show

    </body>
</html>
```

Posteriormente, en otra vista (ej **pagina.blade.php**), podemos indicar que extienda el *layout* que hemos creado y que complete las dos secciones de contenido que habíamos definido en el mismo:

```
@extends('layouts.master')
@section('titulo', ‘El título de nuestra vista’)

@section('header')
    @parent
    <p>Este condenido es añadido al header principal.</p>
@stop

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

8 Controladores Laravel

Controladores básicos (app/http/Controllers)

Dentro de la carpeta de tu proyecto crea tu controlador utilizando **artisan**

- php artisan make:controller UsuarioController

```
<?php
Namespace App/Http/Controllers;
Use Iluminate/Http/Request;
```

Controladores resource

Dentro de la carpeta de tu proyecto crea tu controlador tipo resource (sirve para generar un controlador con unas funciones definidas) utilizando **artisan**

- php artisan make:controller UsuarioController --resource

Enlaces

Podemos utilizar en los enlaces los diferentes helpers disponibles**: url(), action(), route()**

```
<a href="{{ url("/usuarios/”.$user->id) }}">Ver detalles</a>
<a href="{{ url()->previous() }}">Regresar</a>
```

```
// otros métodos de url que se pueden utilizar dentro enlace
url()->current(); //URL completa
url()->full(); // URL completa con la cadena de consukta
```

```
<a href="{{ action('UserController@index') }}">Regresar al listado de usuarios</a>
<a href="{{ action('UserController@show', ['id' => $user->id]) }}">Ver detalles</a>
```

```
Route::get('/usuarios', 'UserController@index')->name('users.index');
Route::get('/usuarios/detalles/{id}', 'UserController@details')
        ->where('id', '[0-9]+')
        ->name('users.show');
-----------------------------------------------------------------------------------------------
<a href="{{ route('users.show', ['id' => $user->id]) }}">Ver detalles</a>
```

Nota: Utilizando el método **name** podemos definir un nombre para nuestra ruta, mi recomendación es que siempre concuerde con lo que hace.

Redirecciones

```
Return redirect()-> action('NombreController@metodo');
```

Middlewares

Los middleware, son funciones que nos permiten agregar filtros a cada petición HTTP realizada por un usuario en una aplicación.

Para crear un middleware (app/http/Middleware) desde artisan

- **php artisan make:middleware NombreMiddleware**

Ejemplo de middleware para comprobar la edad

```php
public function handle($request, Closure $next, $age)
{
    if ($request->user()->age <= $age) {
        abort(403, "¡No tienes edad para ver este video! le diremos a tus padres.");
    }
    return $next($request);
}
```

Una vez creado el Middleware, debes **registrarlos** en la aplicación para que puedan ser usados en las peticiones, para ello dentro del archivo app\Http\Middleware\Kernel.php y de la propiedad $routeMiddleware debemos registrar el Middleware que generamos en la consola, de esta forma:

```
    protected $routeMiddleware = [
        'auth' => \App\Http\Middleware\Authenticate::class,
        'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBas
        'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
        //este es el nuevo middleware
        'age' => \App\Http\Middleware\AgeMiddleaware::class,
    ];
```

Para proteger una ruta

```
Route::get('series/game-of-thrones/', ['middleware' => 'age:18', function () {
    return "eres mayor de edad y puedes ver este contenido"
}]);
```

9 Formularios

Web.php

Configuramos 2 nuevas rutas get y post para nuestro formulario

```
Route::get ('formulario','UsuarioController@formulario');
Route::post ('recibir','UsuarioController@recibir');
```

Users/formulario.blade.php

Todos los formularios tienen protección CSRF y en action tendremos que invocar al metodo del controlador. El fichero CSS está dentro

```
<html>

<head>
        <title>Mi primer formulario</title>
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
        <link href="assets/css/estilos.css" rel="stylesheet" type="text/css" />
</head>
<body>
<form name="formul1" method="POST" action="{{action('UsuarioController@recibir')}}" >
 {{ csrf_field() }}
<fieldset>
  <legend>Formulario</legend>
 <div>
        <label for="nombre">Nombre</label>
        <input type="text" name="nombre" />
 </div>
 <div>
        <label for="apellidos">Apellidos</label>
        <input type="text" name="apellidos" size="50" />
 </div>
 <div>
        <label for="email">Email</label>
        <input type="email" name="email" size="50" />
 </div>
 <div>
  <input type="submit" value="Darme de alta" />

  </div>
</fieldset>
</form>
</body>
</html>
```

de public

App/http/Controllers/UsuarioControllers.php

A través de la variable $request recibimos los valores del formulario HTTP.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UsuarioController extends Controller
{

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

    public function recibir(Request $request)
    {
        $nombre= $request -> input('nombre');
         $apellidos= $request -> input('apellidos');
         $email= $request -> input('email');

         return "$nombre $apellidos $email";
         // Sería mas correcto return view ('users.nuestravista',['nombre'=>$nombre,....]);
    }


}
```

Si queremos añadir un valor por defecto porque no se ha insertado ningún dato desde el formulario se introduce **$nombre= $request -> input ('nombre', 'sin valor');**

10 Base de Datos y Query builder

Configura el fichero **.env** para conectar con tu base de datos. El fichero **/config/database** recoge los datos que hemos modificado en **.env**

***Migraciones*(database/migrations)**

Las migraciones son un sistema de control de versiones para bases de datos. Permiten que un equipo trabaje sobre una base de datos añadiendo y modificando tablas, campos, manteniendo un histórico de los cambios realizados y del estado actual de la base de datos. Las migraciones se utilizan de forma conjunta con la herramienta ***Schema builder***

Creamos una migración para crear una tabla

- php artisan make:migration create_usuarios_table --create=frutas

Creamos una migración para modificar una tabla

- php artisan make:migration add_usuarios --table=frutas

Estructura de las migraciones

```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class CreateFrutasTable extends Migration
{

    public function up()
    {
        Schema::create('frutas', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('nombre',255);
            $table->string('descripcion',255);
            $table->integer('precio');
            $table->date('fecha');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::dropIfExists('frutas');
    }
}
```

Ejecutamos todas las migraciones

- php artisan migrate

Deshacer las últimas migraciones

- php artisan migrate:rollback

También Podemos realizar las modificaciones por medio de SQL introduciéndolo en el método up() de la siguiente manera

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

Con este comando deshacemos los cambios y volvemos a aplicar las migraciones

- php artisan migrate:refresh

Seeders (database/seeds)

Los seeders (semillas) sirven para rellenar datos en las tablas. Creamos el seeder

- php artisan make:seed frutasSeeder

y completamos el metodo **run() de frutasSeeder**

```php
<?php

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;

class frutasSeeder extends Seeder
{
    public function run()
    {
        for ($i=0;$i<=20;$i++){
            DB::table('frutas')->insert([
             'nombre' => 'Cereza'.$i,
             'description' => 'Descripción de la fruta'.$i,
             'precio' => $i,
             'fecha' => date('Y-m-d')
        ]);
                }
     $this-> command->info('La tabla ha sido rellenada');
                }

        }
```

Completamos el DatabaseSeeder para que ejecute nuestro frutasSeeder

```php
<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {

        $this->call(frutasSeeder::class);
                }
}
```

Ejecutamos el seed

Primero actualizaremos la información del cargador automático de clases

- composer dump-autoload

Y después cargamos nuestro Seed

- php artisan db:seed --class=frutasSeeder

Operaciones con la Base de datos (QUERY BUILDER)

Para los ejemplos crearemos un nuevo controlador llamado **frutasController** y construiremos cada una de las rutas para cada una de los metodos dentro de un grupo con el prefijo frutas. En la imagen puedes un ejemplo con ver con la ruta de index

```php
Route::group(['prefix'=>'frutas'], function(){
    Route::get('index', 'FrutaController@index');
});
```

Listar datos

El metodo listar llamamos a una vista (dentro de la carpeta frutas) para que nos muestre el listado de todos los nombres de las frutas. Con el metodo **get()** obtenemos todos los registros de la tabla frutas.

```php
use Illuminate\Http\Request;
use Illuminate\Support\Facades\DB;

class FrutaController extends Controller
{

    public function listar() {

        $frutas = DB::table('frutas')
        ->orderBy('id','desc')
         ->get();

        return view('frutas.index',compact('frutas'));
    }
```

Mostrar una fila

A partir del ID vamos a mostrar a partir de una vista el nombre y los detalles de la fruta. Cuando creemos la ruta tendremos que pasarle el id y en el método utilizamos **first()** en lugar de **get()** para obtener un objeto limpio.

```php
public function detail($id) {

        $fruta = DB::table('frutas')
          ->where('id',$id)->first();

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

Insertar un registro

Tenemos 2 metodos

- **create:** contiene la vista del formulario que no se nos puede olvidar protegerlo con {{csrf_field()}}
- **save:** Inserta los datos en la base de datos y nos redirige al listado (index)

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

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

    return redirect()->action('FrutaController@index');
}
```

Borrado de un registro

Hemos añadido una sesión flash llamada status para indicar que el registro se borrado correctamente.

```
public function delete($id){
    $fruta = DB::table('frutas')->where('id', $id)->delete();
    return redirect()->action('FrutaController@index')->with('status', 'Fruta borrada correctamente');
}
```

En la vista de la lista (index) controlaremos con un if si existe la sesión y mostraremos el mensaje

```
@if (session('status'))
<p style=….>{{session('status')}}</p>
@endif
```

Actualizar un registro

Introduciremos el id a través de un campo hidden en el formulario de la vista y una sesión flash donde informaremos si ha sido actualizado.

```php
public function update(Request $request, $id) {

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

          ]);

        return redirect()->action('FrutaController@index')->with('status','Registro actualizado correctamente');
    }
```

11 Modelos y Eloquent ORM

Con el comando make:model, seguido del nombre del nuevo modelo, creamos un modelo vacío.

- **php artisan make:model Nombre_modelo**

Eso nos crea en el directorio "app" el correspondiente modelo de Eloquent, que contiene un código como el que puedes ver a continuación.

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Nombre_modelo extends Model
{
    //
}
```

Eso es todo lo que necesita un modelo básico en Laravel. Como te puedes imaginar, la explicación de su sencillez es que esta clase extiende la clase Model de Laravel.

En la siguiente página web puedes encontrar información sobre las relaciones de las tablas en la Base de datos
https://laravel.com/docs/5.8/eloquent-relationships

Operaciones con la Base de datos (ELOQUENT)

Namespace del modelo

Recuerda que los modelos están creados dentro de la carpeta "app" de la aplicación, sueltos ahí. Sus clases forman parte del Namespace App y para usarlos primero debemos indicar que vamos a usar tal espacio de nombres.

```
use App\Fruta;
```

Consultar Datos

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Request;
use App\Fruta;

class FrutaController extends Controller
{

    public function Index() {

    $frutas = Fruta::all();
    return view('frutas.index',['frutas'=>$frutas]);


    }
}
```

```
//EJEMPLOS
  $frutas=Fruta::find($id);
  $frutas= Fruta::findOrFail($id); //Sirve para capturar excepciones

  $frutas = Fruta::where('precio', '>', 100)->take(10)->get();
  $frutas = Fruta::where('precio', '>', 100)->first();


 return view('frutas.edit',['frutas'=>$frutas]);
```

Insertar un registro

```php
$fruta=new Fruta();
$fruta->nombre= $request->input('nombre');
$fruta->save();
```

Actualizar un registro dado

```php
$fruta= Fruta::find($id);
$fruta->nombre=$request->input('nombre');

$fruta->save();
```

Borrar un registro

```php
$fruta = Fruta::find($id);
$fruta->delete();
// Otro Ejemplo
$affectedRows = Fruta::where('precio', '>', 100)->delete();
```

12 Control de usuario

Laravel incluye una serie de métodos y clases que harán que la implementación del control de usuarios sea muy rápida y sencilla.

1) La configuración predeterminada en **config/auth.php.**
   a) Sistema de autenticación por defecto **Eloquent**
   b) Modelo datos por defecto **users**
   c) Tabla de usuarios por defecto **users (database/migrations)**

2) La migración para la base de datos de la tabla de usuarios con todos los campos necesarios.

3) El modelo de datos de usuario (**User.php**) dentro de la carpeta app con toda la implementación necesaria.

4) Los controladores para gestionar todas las acciones relacionadas con el control de usuarios (dentro de **App\Http\Controllers\Auth).**

Generar rutas y vistas para realizar login, registro.y recuperar contraseña

LARAVEL 5
➢ **Php artisab make:auth**
LARAVEL 6
➢ **Composer require laravel/ui** ➢ **Php artisan ui vue - -auth** Para generar el estilo tener instalado nodejs ➢ **Npm install** ➢ **Npm run dev**

En la siguiente tabla puedes ver el resumen de las rutas creadas que están enlazadas con los controladores y métodos

- php artisan route:list

Method	Url	Acción
GET	login	LoginController@showLoginForm
POST	login	LoginController@login
POST	logout	LoginController@logout
GET	register	RegisterController@showRegistrationForm
POST	register	RegisterController@register
POST	password/email	ForgotPasswordController@sendResetLinkEmail
GET	password/reset	ForgotPasswordController@showLinkRequestForm
POST	password/reset	ResetPasswordController@reset
GET	password/reset/{token}	ResetPasswordController@showResetForm
GET	home	HomeController@index

Las vistas generadas están en la carpeta **resource/views/auth**

Acceder a los datos del usuario autenticado

Una vez que el usuario está autenticado podemos acceder a los datos del mismo a través del método `Auth::user()`, por ejemplo:

```
$user = Auth::user();
```

Este método nos devolverá `null` en caso de que no esté autenticado. Si estamos seguros de que el usuario está autenticado (porque estamos en una ruta protegida) podremos acceder directamente a sus propiedades:

```
$email = Auth::user()->email;
```

Importante: para utilizar la clase `Auth` tenemos que añadir el espacio de nombres `use Illuminate\Support\Facades\Auth;`, de otra forma nos aparecerá un error indicando que no puede encontrar la clase.

El usuario también se inyecta en los parámetros de entrada de la petición (en la clase `Request`). Por lo tanto, si en un método de un controlador usamos la inyección de dependencias también podremos acceder a los datos del usuario:

```
use Illuminate\Http\Request;

class ProfileController extends Controller {
    public function updateProfile(Request $request) {
        if ($request->user()) {
            $email = $request->user()->email;
        }
    }
}
```

Comprobar si un usuario está autenticado

Para comprobar si el usuario actual se ha autenticado en la aplicación podemos utilizar el método `Auth::check()` de la forma:

```
if (Auth::check()) {
    // El usuario está correctamente autenticado
}
```

Sin embargo, lo recomendable es utilizar *Middleware* (como veremos a continuación) para realizar esta comprobación antes de permitir el acceso a determinadas rutas.

Importante: Recuerda que para utilizar la clase `Auth` tenemos que añadir el espacio de nombres `use Illuminate\Support\Facades\Auth;`, de otra forma nos aparecerá un error indicando que no puede encontrar la clase.

Proteger rutas

El sistema de autenticación de Laravel también incorpora una serie de filtros o *Middleware* (ver carpeta `app/Http/Middleware` y el fichero `app/Http/Kernel.php`) para comprobar que el usuario que accede a una determinada ruta o grupo de rutas esté autenticado. En concreto para proteger el

acceso a rutas y solo permitir su visualización por usuarios correctamente autenticados usaremos el *middleware* `\Illuminate\Auth\Middleware\Authenticate.php` cuyo alias es `auth`. Para utilizar este *middleware* tenemos que editar el fichero `routes/web.php` y modificar las rutas que queramos proteger, por ejemplo:

```
// Para proteger una clausula:
Route::get('admin/catalog', function() {
    // Solo se permite el acceso a usuarios autenticados
})->middleware('auth');

// Para proteger una acción de un controlador:
Route::get('profile', 'ProfileController@show')->middleware('auth');
```

Si el usuario que accede no está validado se generará una excepción que le redirigirá a la ruta `login`. Si deseamos cambiar esta dirección tendremos que modificar el método que gestiona la excepción, el cual lo podremos encontrar en `App\Exceptions\Handler@unauthenticated`.

Si deseamos proteger el acceso a toda una zona de nuestro sitio web (por ejemplo la parte de administración o la gestión de un recurso), lo más cómodo es crear un grupo con todas esas rutas que utilice el *middleware* `auth`, por ejemplo:

```
Route::group(['middleware' => 'auth'], function() {
    Route::get('catalog',        'CatalogController@getIndex');
    Route::get('catalog/create', 'CatalogController@getCreate');
    // ...
});
```

Si lo deseamos también podemos especificar el uso de este *middleware* desde el constructor del controlador:

```
public function __construct() {
    $this->middleware('auth');
}
```

Sin embargo, lo más recomendable es indicarlo desde el fichero de rutas pues así tendremos todas las rutas y filtros centralizados en un único fichero.

Ejercicios

Para cada ejercicio tendrás que crear un proyecto nuevo. Recuerda que la versión la puedes comprobar en el github de laravel

Composer create-project laravel/laravel *nombre-proyecto* **"5.8.*" --prefer-dist**

1. **FRUTERÍA**

   (Nota: Utiliza Includes para el estilo / Query Builder para el acceso a datos)

   En este primer ejercicio vamos a realizar la implantación del ejemplo de la **frutería** que se ha visto en el apartado de Base de Datos utilizando Laravel. Consiste en una pantalla principal con Header, el footer y un nav con un menú
   - Principal
   - Listado de frutas
   - Crear fruta nueva

   Las características de las opciones son
   - En la página principal aparecerá una foto de una frutería
   - El listado de frutas es una lista no ordenada(ul) de los nombres de la frutas
   - Cada uno de los nombres será un enlace para **ver los detalles de la fruta**. Al lado de los nombres de las frutas estarán los enlaces para **borrar y actualizar**.

Para empezar sigue los siguientes apartados

a. Crear una Base de datos llamada **frutería** y modificar el fichero **.env** para la conexión con la BD
b. Utilizar una migración para crear la **tabla frutas** con los campos (id, nombre, descripción, precio y fecha). Utiliza los tipos de datos mas adecuado para cada campo
c. Utilizar un seeder con un for para introducir 50 datos dentro de la tabla frutas.
d. Crea un **nuevo controlador llamado FruteriaController** utilizando artisan.
e. Modifica la ruta principal (/) que apuntará a una vista index.blade.php con header, nav (principal, listado de frutas y crea nueva fruta) y footer (Utilizando **includes**).
   La página principal incluirá una foto de una frutería.
f. Para todas las funciones de la tabla frutas tiene que declarar una ruta que llame al método correspondiente del controlador FruteriaController **Ejemplo** de frutas /fruta/update (Utiliza un grupo).

Importante: No llames directamente a la vista desde la ruta es recomendable que cada Ruta llame al método que le corresponda dentro del controlador y dentro del método se llame a la vista.

g. Implementa cada uno de los métodos que vienen en los apuntes (con sus rutas) y crea las vistas necesarias (incluye header, nav y footer en cada vista). Las vistas estarán en la carpeta **views/frutas/….**

2. VIDEOCLUB

(Nota :Utiliza Layouts para el estilo / Eloquent ORM para acceso a datos)

Vamos a desarrollar una pequeña web para la gestión interna de un **videoclub**, el cual estará protegido mediante usuario y contraseña. Una vez autorizado el acceso, el usuario podrá listar el catálogo de películas, ver información detallada de una película, realizar búsquedas o filtrados y algunas operaciones más de gestión.

a) Instalación de Laravel y crear un proyecto llamado Videoclub. Configúralo (clave de seguridad, permisos, etc.) y probaremos que todo funcione correctamente.

b) Crea una base de datos llamada **videoclub** utilizando phpmyadmin y configura la conexión con la BD (.env)

c) Crea **el modelo Movie** de datos asociado con la tabla *movies* (utilizando Artisan). Vamos a utilizar **Eloquent** para añadir datos a través del seeder.

d) **Crear el controlador catalogController** (utilizando Artisan)

e) **Crea una migración** utilizando artisan llamada **create_movies_table** con la tabla **movies**

Campo	Tipo	Valor por defecto
id	Autoincremental	
title	String	
year	String de longitud 8	
director	String de longitud 64	
poster	String	
rented	Booleano	false
synopsis	Text	
timestamps	Timestamps de Eloquent	

Nota: El campo boolean es TinyInt(1) el valor por defecto es 0 (false).

f) Crea un nuevo seed (Semilla) con artisan llamado **CatalogSeeder** para rellenar los datos de la tabla movies. Copia la información del fichero **array_peliculas.php** (plataforma) dentro del método run y declara como privada el atributo $arrayPeliculas dentro de la clase. Modifica el fichero **DatabaseSeeder.php.**

```
use Illuminate\Database\Seeder;
use App\Movie;
class seedCatalog extends Seeder{
    private $arrayPeliculas;
    public function run()
    { // ……Copia la información del fichero……}
```

Nota: Añade la línea **use App\Movie** dentro de **CatalogSeeder** para que puedar usar el Modelo

g) Se va instalar el **control de usuario** (utilizando artisan)

h) Ejecuta todas las migraciones y seeds

i) Vamos a definir las rutas principales que va a tener nuestro sitio web. A continuación se incluye una tabla con las rutas a definir (todas de tipo GET) y el controlador con el método asociado

Ruta	Controlador	Método
/	HomeController	getHome
catalog	CatalogController	getIndex
catalog/show/{id}	CatalogController	getShow
catalog/create	CatalogController	getCreate
catalog/edit/{id}	CatalogController	getEdit

Nota: Para comprobar que las rutas se hayan creado correctamente utiliza el comando de **artisan** que devuelve un listado de rutas y además prueba también las rutas en el navegador.

j) En este ejercicio vamos a crear el *layout* base que van a utilizar el resto de vistas del sitio web y además incluiremos el código base Bootstrap (https://getbootstrap.com/docs/4.0/getting-started/introduction/#starter-template) para utilizarla como estilo base.

- Descargar de la plataforma **navbar.blade.php** y almacenarla en la carpeta **resource/views/partials**
- A continuación, vamos a crear el *layout* principal de nuestro sitio. Para esto descargamos el fichero **resources/views/layouts/master.blade (**ya está incluido el código base de Bootstrap con nuestro navbar.blade.php incluido)

k) Se crean las vistas asociadas a cada ruta, las cuales tendrán que extender de **layouts.master. Mejora las secciones de header y footer**

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

@section('titulo', 'Página principal')
@section('content')

    <p> En este apartado realizaré la vista correspondiente</p>
    <p>Las secciones header y footer serán iguales a las del master y no se ponen </p>
@stop
```

Vista	Carpeta	Ruta asociada
home.blade.php	resources/view/	/
login.blade.php	resources/view/auth/	Login ()
Sin vista		logout
index.blade.php	resources/view/catalog/	catalog
show.blade.php	resources/view/catalog/	catalog/show/{id}
create.blade.php	resources/view/catalog/	catalog/create
edit.blade.php	resources/view/catalog/	catalog/edit/{id}

Nota: Recuerda tendremos que pasar datos en algunas de las vistas, ejemplo 2 formas diferentes

Return view('catalog.show',['id'=>$id]');

Return view('catalog.show', compact('id'));

Cada controlador accederá a la base de datos utilizando ELOQUENT ORM. Recuerda que tienes que haber creado el modelo Movie (apartado C)

- **HomeController@getHome:** En este método de momento solo vamos a hacer una redirección a la acción que muestra el listado de películas del catálogo CatalogController@getIndex comprobando si el usuario esta logueado o no, y en caso de que no lo esté redirigirle al formulario de login.

- **CatalogController@getIndex:** Este método tiene que mostrar un listado de todas las películas que tiene el videoclub y pasarlas a la vista.

  Completar la vista **index.blade.php**

```
<div class="row">

    @foreach( $arrayPeliculas as $key => $pelicula )
    <div class="col-xs-6 col-sm-4 col-md-3 text-center">

      COMPLETAR

    </div>
    @endforeach

</div>
```

  El código, en primer lugar tiene que crear una fila (usando el sistema de rejilla de Bootstrap) y a continuación se realiza un bucle *foreach* utilizando la notación de *Blade* para iterar por todas las películas. Para cada película obtenemos su posición en el array y sus datos asociados, y generamos una columna para mostrarlos. Es importante que nos fijemos en como se itera por los elementos de un array de datos y en la forma de acceder a los valores. Además se tiene que incluir un enlace para que al pulsar sobre una película nos lleve a la dirección `/catalog/show/{$id}`.

- **CatalogController@getShow:** Este método se utiliza para mostrar la vista detalle de una película (Utiliza el método **findOrFail**). En esta vista vamos a crear dos columnas, la primera columna para mostrar la imagen de la película y la segunda para incluir todos los detalles. A continuación se incluye la estructura HTML que tendría que tener esta pantalla:

```
<div class="row">

    <div class="col-sm-4">

        {{-- TODO: Imagen de la película --}}

    </div>
    <div class="col-sm-8">

        {{-- TODO: Datos de la película --}}

    </div>
</div>
```

En la columna de la izquierda completamos el TODO para insertar la imagen de la película. En la columna de la derecha se tendrán que mostrar todos los datos de la película: título, año, director, resumen y su estado. Para mostrar el estado de la película consultaremos el valor `rented` del array, el cual podrá tener dos casos:

- En caso de estar disponible (false) aparecerá el estado "Película disponible" y un botón azul para "Alquilar película".
- En caso de estar alquilada (true) aparecerá el estado "Película actualmente alquilada" y un botón rojo para "Devolver película".

Además tenemos que incluir dos botones más, un botón que nos llevará a editar la película y otro para volver al listado de películas.

Nota: Acordaros que en **Bootstrap** podemos transformar un enlace en un botón, simplemente aplicando las clases **"btn btn-default"**

La chaqueta metálica

Año: 1987

Director: Stanley Kubrick

Resumen: Un grupo de reclutas se prepara en Parish Island, centro de entrenamiento de la marina norteamericana. Allí está el sargento Hartman, duro e implacable, cuya única misión en la vida es endurecer el cuerpo y el alma de los novatos, para que puedan defenderse del enemigo. Pero no todos los jóvenes están preparados para soportar sus métodos.

Estado: Película actualmente alquilada.

Devolver película | Editar película | Volver al listado

- **CatalogController@getCreate**

  Este método devuelve la vista "`catalog.form`" para añadir una nueva película. Para crear este formulario en la vista correspondiente nos podemos basar en el contenido de la plantilla "`catalog_create.php`". Esta plantilla tiene una serie de TODOs que hay que completar. En total tendrá que tener los siguientes campos:

Label	Name	Tipo de campo
Título	title	text
Año	year	text
Director	director	text
Poster	poster	text
Resumen	synopsis	textarea

  Además tendrá un botón al final con el texto "Añadir película".

CatalogController@getEdit: Este método permitirá modificar el contenido de una película (Utiliza **findOrFail)**. Utilizaremos el mismo formulario que para crear (**catalog.form**), modificando el código para que cambie las siguientes puntos.

- El título por "Modificar película".
- El texto del botón de envío por "Modificar película".
- Añadir justo debajo de la apertura del formulario el campo oculto para indicar que se va a enviar por PUT. Recordad que Laravel incluye el método `method_field('PUT')` que nos ayudará a hacer esto.

l) Modificar los siguientes ficheros para el sistema de autenticación

- Edita el fichero `routes/web.php` y realiza las siguientes acciones:
  - Añade un *middleware* de tipo grupo que aplique el filtro `auth` para proteger todas las rutas del catálogo (menos la raíz `/` y las de autenticación).
  - Revisa mediante el comando de Artisan `php artisan route:list` las nuevas rutas y que el filtro `auth` se aplique correctamente.
  - Modifica el controlador `LoginController` para que cuando se realice el *login* te redirija a la ruta `/catalog`. Para esto tienes que modificar su propiedad `redirectTo` para añadir la ruta de redirección
- Modifica la vista de *login* generada por Laravel (`resources/views/auth/login.blade.php`) para que en lugar de utilizar su *layout* utilice el que creamos en los primeros ejercicios (`resources/views/layouts/master.blade.php`).

- A continuación abrimos el controlador `HomeController` para completar el método `getHome`. Este método de momento solo realiza una redirección a `/catalog`. Modifica el código para que en caso de que el usuario no esté autenticado le redirija a la ruta `/login`
- Por último edita la vista `resources/views/partials/navbar.blade.php` que habíamos copiado de las plantillas y cambia la línea `@if( true || Auth::check() )` por `@if( Auth::check() )`. De esta forma el menú solo se mostrará cuando el usuario esté autenticado.

m) Añadir y editar películas

En primer lugar vamos a añadir las rutas que nos van a hacer falta para recoger los datos al enviar los formularios. Para esto editamos el fichero de rutas y añadimos dos rutas (también protegidas por el filtro `auth`):

- Una ruta de tipo POST para la url `catalog/create` que apuntará al método `postCreate` del controlador `CatalogController`.
- Y otra ruta tipo POST para la url `catalog/edit/{id}` que apuntará al método `postEdit` del controlador `CatalogController`.

A continuación, vamos a editar la vista `catalog/edit.blade.php` con los siguientes cambios:

- Revisar que el método de envío del formulario sea tipo PUT.
- Tenemos que modificar todos los *inputs* para que como valor del campo ponga el valor correspondiente de la película. Por ejemplo, en el primer *input* tendríamos que añadir `value="{{$pelicula->title}}"`. Realiza lo mismo para el resto de campos: *year*, *director*, *poster* y *synopsis*. El único campo distinto será el de *synopsis* ya que el *input* es tipo *textarea*, en este caso el valor lo tendremos que poner directamente entre la etiqueta de apertura y la de cierre.

Por último tenemos que actualizar el controlador `CatalogController` con los dos nuevos métodos. En ambos casos tenemos que usar la inyección de dependencias para añadir la clase `Request` como parámetro de entrada (revisa la sección "Datos de entrada" de la teoría). Además para cada método haremos:

- En el método `postCreate` creamos una nueva instancia del modelo `Movie`, asignamos el valor de todos los campos de entrada (*title*, *year*, *director*, *poster* y *synopsis*) y los guardamos. Por último, después de guardar, hacemos una redirección a la ruta `/catalog`.

- En el método `postEdit` buscamos la película con el identificador pasado por parámetro, actualizamos sus campos y los guardamos. Por último realizamos una redirección a la pantalla con la vista detalle de la película editada.