CREACIÓN CRUD DESDE CERO EN LARAVEL 8

Contenido

<Video Unidades 1, 2 y 3>

1. INTRODUCCIÓN.

Laravel es un framework de código abierto escrito en php que se utiliza para desarrollar aplicaciones y servicios web. Su filosofía es desarrollar código php de forma elegante y simple evitando el código espagueti (estructura de control de flujo compleja e incomprensible). Un framework viene a ser un entorno de trabajo con código escrito por una comunidad de expertos programadores, y entre las principales ventajas tenemos, evitar tener que escribir código desde cero. La mayoría de los proyectos tienen partes comunes necesarias para funcionamiento como por ejemplo acceso a la base de datos, validación de formularios, un sistema de autentificación, paginación, etc. y Laravel nos evita tener que escribir este código desde cero. Otra de las ventajas de trabajar con Laravel es que vamos a generar buenas prácticas a la hora de programar. Laravel está basado en el patrón modelo-vista-controlador, que nos permite separar los datos y la lógica de nuestra aplicación, de la interfaz del usuario y gracias a ello vamos a tener nuestro código mucho más ordenado. En conclusión, con Laravel vamos a poder hacer un proyecto mucho más rápido limpio y seguro.

Laravel es un framework muy amplio, el cual necesita un tiempo mayor al disponible en este curso para conocerlo al completo, así que lo que se pretende en esta guía es describir los pasos más comunes partiendo de cero y siguiendo buenas prácticas, para desarrollar una típica aplicación CRUD, que permita realizar operaciones de consulta y manipulación de datos sobre una tabla. Con el asentamiento de estos conocimientos iniciales, el alumno podrá seguir explorando este framework para conocer aquellas partes no estudiadas, así como nuevos componentes que faciliten el desarrollo de nuevas funcionalidades en un proyecto.

2. ESTRUCTURA DE DIRECTORIOS

Lo primero para empezar a trabajar con Laravel, es conocer donde se almacenan los ficheros del proyecto, por lo que ahora describiremos sus directorios principales, pero no te preocupes si no comprendes del todo lo que se almacena en cada directorio ya que la mayoría no los tocaremos, esto es solo una introducción para hacernos una ligera idea de la estructura general de un proyecto Laravel:

- El directorio App

El directorio app contiene el código central de su aplicación. Casi todas las clases de tu aplicación estarán en este directorio. Contiene varios directorios en su interior, pero el que más nos interesa en principio es el directorio Http, que contiene sus controladores, middleware y solicitudes de formularios. Casi toda la lógica para manejar las peticiones que entren en su aplicación se colocará en este directorio.

- El directorio Bootstrap

El directorio bootstrap contiene el archivo app.php, que inicia el marco. Este directorio también alberga un directorio de cache que contiene los archivos generados por el marco para la optimización del rendimiento, como los archivos de caché de rutas y servicios.

- El directorio Config

El directorio config, como su nombre indica, contiene todos los archivos de configuración de su aplicación. Es una gran idea leer todos estos archivos y familiarizarse con todas las opciones disponibles.

- El directorio Database

El directorio database contiene sus migraciones de bases de datos, fábricas de modelos y semillas. Si lo desea, también puede utilizar este directorio para mantener una base de datos SQLite.

- El directorio Public

El directorio public contiene el archivo index.php, que es el punto de entrada para todas las solicitudes que ingresan a su aplicación y configura la carga automática. Este directorio también alberga los activos, como imágenes, JavaScript y CSS.

- El directorio Resources

El directorio resources contiene sus vistas, así como sus activos crudos no compilados como LESS, SASS o JavaScript. Este directorio también alberga todos los archivos de tu idioma.

- El directorio Routes

El diretorio routes contiene todas las definiciones de rutas para su aplicación. Por defecto, se incluyen varios archivos de rutas con Laravel: web.php, api.php, console.php y channels.php. El único que usaremos en esta guía es el archivo web.php que contiene las rutas que el RouteServiceProvider coloca en el grupo de middleware web, que proporciona el estado de sesión, la protección CSRF y la encriptación de cookies. Si su aplicación no ofrece una API sin estado, RESTful, lo más probable es que todas sus rutas estén definidas en el archivo web.php.

- El directorio Storage

El directorio storage contiene sus plantillas Blade compiladas, sesiones basadas en archivos, cachés de archivos y otros archivos generados por el framework. Este directorio está segregado en los directorios app, framework, y logs. El directorio app puede utilizarse para almacenar cualquier archivo generado por tu aplicación. El directorio framework se utiliza para almacenar los archivos generados por el framework y los cachés. Por último, el directorio logs contiene los archivos de registro de su aplicación.

El directorio storage/app/public puede utilizarse para almacenar archivos generados por el usuario, como avatares de perfiles, que deben ser accesibles al público. Debería 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 Testsy

El directorio tests contiene sus pruebas automatizadas. Cada clase de prueba debe ser sufijada con la palabra Test. Puede ejecutar sus pruebas usando los comandos phpunit o php vendor/bin/phpunit.

- El directorio Vendor

El directorio vendor contiene las dependencias de Composer.

Carpetas y archivos principales en los que trabajaremos nosotros:

- routes/web.php -> archivo con las rutas de acceso a todas las páginas del proyecto
- app/http/Controllers/ -> Controladores (lógica) del proyecto
- app/Models/ -> Modelos de las tablas de la BD
- resources/views/layouts/ -> archivos plantillas (.blade.php), parámetros con @yield('nombre')
- resources/views/ -> vistas(.blade.php) de controladores: @extends(layouts.plantilla), @section para rellenar huecos y {{$nombreParámetro}} para parámetros
- config/database.php -> configuración tipo BD del proyecto y parámetros de acceso (usar credenciales env)
- .env -> credenciales de acceso a BD (no son incluidas al subir el proyecto a un repositorio)
- database/migrations/ -> migraciones para la creación de tablas de la base de datos
- phpMyAdmin: La tabla migrations contiene el historial de todas las migraciones en cada batch (son los procesos de migración numerados)
- resources/lang/en -> Se almacenan mensajes en ingles de los errores y otras notificaciones

3. CREACIÓN DEL PROYECTO.

Para poder crear un proyecto Laravel necesitamos preparar nuestro entorno de desarrollo, por lo que primero debes realizar los siguientes pasos:

1. Instalar si no se ha hecho previamente XAMPP (servidores), COMPOSER (gestor de dependencias) y Git bash (terminal de comandos), con las opciones por defecto.

2. Instalar laravel en servidor local, para ello desde la consola 'Git Bash' situarse en la carpeta htdocs y escribir el siguiente comando:

```
composer global require laravel/installer
```

3. Para crear un proyecto, desde la carpeta htdocs en la consola 'Git Bash', ejecutar el siguiente comando, que creara la carpeta del proyecto con la estructura de directorios y ficheros iniciales:

```
Laravel new nombreProyecto
```

4. Comprobar la creación del proyecto desde `htdocs/nombreProyecto/public` en el navegador web, lo que abrirá una página con enlaces a la documentación de Laravel.
5. Si el proyecto hace tiempo que fue desarrollado, actualizar las dependencias ejecutando en la consola 'Git Bach' desde la carpeta raíz del proyecto `composer update`

<Video Unidad 4 Rutas>

4. RUTAS. Archivo web.php

Laravel se basa en el front controller, que consiste en que vamos a tener un solo punto de entrada a nuestra aplicación. Los únicos archivos a los que el usuario va a poder acceder, van a ser los que se encuentren dentro de la carpeta public y ese archivo de partida, ubicado en dicha carpeta es index.php. Por tanto, para incluir un archivo css o un archivo javascript, se debe ubicar también en la carpeta public, ya que en otro lado el usuario no podría acceder a ellos por un motivo de seguridad. Dado que la única carpeta accesible es public, para definir las urls de los distintos archivos del proyecto que estén ubicados en otras carpetas, se hace uso del archivo routes/web.php, donde se define una correspondencia de cada url pública que ve el cliente en la barra de direcciones del navegador, con la respuesta que se enviará a dicho cliente, que generalmente será la ejecución de un método de algún controlador ubicado en un archivo dentro del proyecto, y este archivo de rutas es consultado en cada petición web. En cada ruta definida se establece el protocolo usado (get, post, put, delete), la url externa que usará el cliente en la petición al servidor y el método o función que se va a ejecutar y que será el que devuelva con la sentencia `return` el resultado al cliente. Se puede usar parte de la url como parámetro para la página de destino, para lo cual habrá que encerrar entre llaves {} dicha información

```
Route::get('hola', function (){ return 'Hello World'; }); //sin parámetro
Route::get('hola/{nombre}', function ($nombre){ return 'Hello '.$nombre; }); //con parámetro
```

Para definir un redireccionamiento de ruta que redirige a otra url, puede usar el método `Route::redirect()` para que no tener que definir una ruta completa y un controlador para realizar una redirección simple:

```
Route::redirect('/urlorigen', '/urldestino');
```

Si únicamente se necesita devolver una vista desde una ruta, se puede utilizar el método Route::view para no tener que definir un controlador solo para devolver una vista. El método view de la clase Route acepta una url como primer parámetro y un nombre de vista como segundo y además, se le puede pasar un array de parámetros si la vista lo requiere.

```
Route::view('/error', 'error');
Route::view('/error', 'error', ['mensaje' => 'web caída']); //si la vista acepta un parámetro
```

Hay que generar una ruta por cada petición web (normalmente una por cada método de las clases de los controladores) en el archivo `routes/web.php` y aunque no es obligatorio, es muy recomendable darle un nombre a cada ruta con el método `->name('nombre');` al final de la ruta, para usarlos en cada petición web en php, con el comando `route('nombreRuta', parametro1, parametro2,…)` en vez de la url, ya que así es posible

cambiar todas las rutas del proyecto de una sola vez, sin modificar el código donde aparecen, sino actualizando la ruta en `routes/web.php`.

Ejemplo de todas las rutas de un CRUD (es altamente recomendable usar los mismos nombres):

- `Route::get('/',HomeController::class); //Si no se especifica un método de la clase, en la ruta raiz del proyecto se llama por defecto al metodo __invoke del controlador HomeController`
- `Route::get('productos',[ProductoController::class,'index'])->name('productos.index');//llamada al método index del controlador ProductoController`
- `Route::get('productos/{producto}',[ProductoController::class,'show'])->name('productos.show'); //llama al método show y le pasa en la url el parámetro {producto} con el id del producto.`
- `Route::get('productos/create',[ProductoController::class,'create'])->name('productos.create');`
- `Route::post('productos/{producto}',[ProductoController::class,'store'])->name('productos.store');`
- `Route::get('productos/{producto}/edit',[ProductoController::class,'edit'])->name('productos.edit');`
- `Route::put('productos/{producto}',[ProductoController::class,'update'])->name('productos.update');`
- `Route::delete('productos/{producto}',[ProductoController::class,'destroy'])->name('productos.destroy');`

IMPORTANTE: Las rutas se van comprobando de arriba a abajo y cuando se encuentra una coincidencia no se sigue comprobando las demás. Dos rutas pueden ser la misma pero con distinto protocolo (get,post,put,delete,…) y se cogerá una u otra según el protocolo usado.

<Video Unidad 5 Controladores> (Hasta el minuto 15:18)

5. CONTROLADORES I

En una aplicación CRUD, por cada tabla/modelo de la BD, se suele crear un controlador en `app/http/Controllers/` que tendrá los métodos necesarios para operar con los datos de dicha tabla. En vez de crear el controlador manualmente en VSC, se creará desde la consola (para abrir la consola de VsCode pulsar Ctrl+ñ) con el siguiente comando:

```
php artisan make:controller nombreController
```

Después de crear el controlador hay que añadir al `routes/web.php` la sentencia `'use App\Http\Controllers\nombreController;'` para importar el controlador en el archivo de rutas y poder asignar las llamadas a los métodos definidos en el controlador en las rutas definidas.

Para asignar la ejecución de un método de un controlador a una ruta se utiliza el nombre del controlador en la definición de la ruta, si no se especifica ningún método del controlador se llamará al método por defecto __invoke(), pero si vamos a llamar a otro método del controlador se establece un array, donde el primer valor es el nombre del controlador y el segundo es el nombre del método llamado:

```
Route::get('/', HomeController::class); //al no especificar un método se llama al método __invoke
Route::get('cursos', [CursoController::class,'index']) //llamada al método index del controlador
```

Para completar el CRUD de una tabla, desde VsCode debemos crear un método para cada una de las opciones en el controlador de dicha tabla, que después de las operaciones pertinentes devolverá al cliente su correspondiente vista asociada.
En un método del controlador después de realizar las operaciones programadas, si queremos hacer un redireccionamiento a otra ruta, en vez de devolver una vista, debemos usar el comando:

```
return redirect()->route('inicio'); // inicio es el nombre que hemos asignado a la ruta destino.
```

<Video Unidad 6 Vistas>(ver solo hasta el minuto 7:50)

6. VISTAS I

Las vistas se crearán desde Visual Studio Code en la carpeta `'resources/views'` con la estructura de la pagina html.

Desde el controlador se devolverá la vista al navegador del cliente, con el comando `return view('nombreVista');` y en el nombre no hace falta indicar la ruta de la carpeta `resources/views` ni la extensión del archivo. El nombre que asignemos a la vista, por convención vamos a hacer que coincida con el método del controlador que devuelve esa vista, junto con el nombre de la tabla con la que opera el controlador, así identificaremos fácilmente la vista devuelta en cada método, por ejemplo `cursos.create` será el nombre que debemos asignar a la vista devuelta por el método `create` del controlador `CursoController`

Para pasar parámetros a la vista se añade un segundo parámetro al método view, que será un array asociativo con los parámetros que se van a pasar `return view('nombreRuta',['curso'=>$curso]);` pero si el nombre del parámetro coincide con el de la variable que indicamos como valor, que es lo más habitual, podemos usar la función compact de la siguiente forma `return view('nombreRuta', compact('curso'));`

7. BASE DE DATOS Y MODELOS DE DATOS

<Video1 Unidad 7 Configurar acceso a BD>

1. Configuración.

Establecer en archivo `'.env'` las credenciales de acceso a la BD. En este archivo están definidas las variables a las cuales hay que asignar los datos de acceso a la BD y estas variables serán usadas en el archivo de configuración `'config/database.php'`

<Video2 Unidad 7 Introducción a migraciones>

<Video3 Unidad 7 Creación con migraciones>

<Video4 Unidad 7 Modificación con migraciones> (Hasta el minuto 8:20)

2. Crear Migraciones.

Crear la BD vacía en phpmyadmin y a continuación una migración (`database/migrations/`) por cada tabla, para que se generen automáticamente y poder llevar un control de versiones de todos los cambios, incluso volver a un estado anterior tanto de los datos como del esquema de la BD. Cada migración tiene un método `up()`, que se ejecuta cuando se lanza la migración (creación de la tabla) y un método `down()` que se ejecuta cuando se deshace la migración (borrado de la tabla). Por defecto se supone que cada tabla debemos generarla con un campo clave 'id' autonúmerico).

- `php artisan make:migration create_nombreTabla_table` (crea automáticamente la estructura de la migración, con nombreTabla en minúsculas y plural)
- `php artisan migrate` -> recorre todas las migraciones ejecutando el método up() de cada migración (el que realiza la modificación creando las tablas)
- `php artisan migrate:fresh` -> borra todas las tablas de una vez (sin ejecutar método down) y las crea de nuevo (se pierden todos los datos)
- `php artisan migrate:refresh` -> borra una a una las tablas usando el metodo down y las crea de nuevo (se utiliza si queremos incluir alguna operación en el método down, que se ejecute con el borrado de la tabla)
- `php artisan migrate:reset` -> borra todas las tablas de la base de datos
- `php artisan migrate:rollback` -> deshace las migraciones hechas en el último batch (borra cualquier cambio hecho en las tablas desde ese momento)
- `php artisan make:migration add_nombreColumna_to_nombreTabla_table` (crea una migración nueva con la estructura para añadir una columna a una tabla)

Una vez creadas las migraciones de cada tabla con la opción `make:migration` del comando `php artisan`, completar la migración con todos los campos de la tabla, desde visual estudio code. Un ejemplo del contenido de una migración para crear una tabla podría ser:

```
public function up(){
   Schema::create('alumnos', function (Blueprint $table) {
      $table->id(); //Entero autoincrementado
      $table->string('matricula',6)->unique(); //varchar de 6 caracteres con índice único
      $table->string('nombre'); //varchar de 255 caracteres
      $table->string('apellidos')->nullable(); //varchar que permite nulos
      $table->integer('edad'); //entero
      $table->timestamp('fecha_nacimiento'); //tipo fecha
      $table->text('intereses'); //texto de más de 255 caracteres
      $table->timestamps(); //columnas fecha de creación y última modificación de cada registro

      //opcion 1 para definir una clave ajena
      $table->foreignId('grupo_id')->constrained('grupos'); //clave ajena de tabla grupos

      //opción 2 para definir una clave ajena
      $table->unsignedBigInteger('grupo_id'); //columna de clave ajena
      $table->foreign('grupo_id')->references('id')->on('grupos'); //restricción de clave ajena
});}
```

Si se añaden claves foráneas, mejor hacerlo en migraciones independientes que modifiquen la tabla ya creada, para que evitar error si no se ejecutan en el orden correcto según las dependencias de una tabla con otra.

<Video5 Unidad 7 Creación de modelos de datos>

3. Crear Modelos de datos

Los modelos de datos en Laravel tienen definido el código para realizar las operaciones comunes sobre las tablas, por lo que no hay que generar dicho código, aunque podemos ampliar el modelo con nuevos métodos que realicen otras funcionalidades no definidas. El nombre del modelo debe empezar con mayúscula y terminar en singular para enlazar automáticamente con la tabla de la bd, la cual debe tener su nombre en minúsculas y plural sobre el idioma inglés. Si queremos usar otra nomenclatura distinta, debemos incluir la sentencia `protected $table = "nombreTabla";` en la clase del modelo, para establecer la correspondencia entre el modelo y la tabla de la bd.

Una vez creadas las tablas en la BD a partir de las migraciones, crear un modelo en `app/Models/` por cada tabla desde la consola:

- `php artisan make:model nombreModelo //crea el modelo en app/Models`
- Añadir la propiedad `protected $guarded=[];` dentro de la clase del modelo para poder asignar de forma masiva (con 1 sola sentencia) todos los datos recibidos de un formulario a la BD.
- `php artisan tinker ->` Al ejecutar este comando en la consola, entramos a la shell de tinker para poder usar comandos eloquent orm de manera interactiva con la bd (exit para salir), ejemplo de uso del modelo 'Alumno' en tinker:
    - `use App/Models/Alumno; //necesario para hacer uso del modelo`
    - `$alumno = new Alumno; //crea una variable objeto de tipo Alumno`
    - `$alumno->matricula = '123abc'; //asignar un valor a un atributo del objeto`
    - `$alumno; muestra el contenido de todos los atributos de la instancia alumno`
    - `$alumno->save(); //insert o actualiza si existe, los datos del objeto en la BD`

<Video6 Unidad 7 Generar Seeder>

<Video7 Unidad 7 Generar Factory>

4. Llenar las tablas con datos de prueba

Tambien es posible añadir datos a las tablas de manera automática, generados por laravel con factory. Crear un factory (se creará en la ruta '`database/factories/`') por cada tabla, desde la consola con el comando `php artisan make:factory nombreModeloFactory --model=nombreModelo` , especificando el nombre del modelo enlazado con la tabla. Editar el factory creado completando la sentencia return con los tipos de datos que queremos generar para cada campo de la tabla. Por ejemplo:

```
return[
 'matricula'=>$this->faker->unique()->bothify('###-??'), //tres numeros, un guión y dos letras
 'nombre'=>$this->faker->name, //nombre aleatorio en inglés
 'direccion'=>$this->faker->address, //dirección aleatoria en inglés
 'email'=>$this->faker->unique()->safeEmail, //correo electrónico sin repeticiones
 'edad'=>$this->faker->randomNumber(2,true), //número con 2 digitos, false serían 2 dígitos máximo
 'sexo'=>$this->faker->randomElement('Masculino','Femenino'), //aleatorio entre los elementos indicados
 'curso'=>$this->faker->numberBetween(1,4).'ESO'.$this->faker->randomElement(['A','B','C']) //Ej: 2ESOC
 'descripcion'=>$this->faker->paragraph(), //Texto con varias líneas
]
```

Documentación librería Faker:
https://codersfree.com/blog/documentacion-de-la-libreria-faker-php-traducida-al-espanol

En el seeder '`databases/seeders/DatabaseSeeder.php`' por cada tabla a rellenar, añadir 2 líneas:

- Al principio-> '`use App\Models\nombreModelo;`'

- Dentro del método run()-> '`nombreModelo::factory(numRegistros)->create();`'

Con el comando de consola '`php artisan db:seed`' o también añadiendo '`--seed`' al final de cualquier comando de migración se insertarán en las tablas todos los datos generados aleatoriamente.

<Video Unidad 8 Consultas en Elocuent ORM>

8. ELOCUENT ORM,

Veamos los principales comandos de Elocuent ORM que nos van a permitir manipular y consultar los datos de la BD (se pueden probar en tinker, ejecutar en la consola '`php artisan tinker`'):

MANIPULACIÓN DE DATOS

- Para insertar un registro hay que crear el objeto del modelo, dar valor a sus propiedades y ejecutar el comando `$variableObjeto->save();` para reflejar el cambio en la bd.
- Para actualizar un registro solo hay que recuperar con el id, el objeto a modificar y después de actualizar, ejecutar el mismo comando anterior `$variableObjeto->save();`
- Para eliminar un registro solo hay que crear o recuperar el objeto con el id y ejecutar el comando `$variableObjeto->delete();` También se puede usar el método estático `Articulo::destroy(1);` para eliminar el artículo con `id=1`.

CONSULTAS DE DATOS

```
use App\Models\Articulo; //debemos usar el modelo de la tabla (en este caso Articulo)
$articulos = Articulo::all(); //método all() devuelve un array con todos los objetos (cada registro de la tabla)
$articulos = Articulo::get();//método get() devuelve un array con los objetos recuperados por la consulta
$articulo = Articulo::first();//método first() devuelve el primer objeto recuperado en la consulta, no un array
$articulos = Articulo::where('titulo', 'Voluptas.')->get(); //método where() filtra los artículos con titulo igual a 'Voluptas.' y es necesario el método get para crear el array de objetos
$articulos = Articulo::orderby('id','desc')->get(); //método orderby() ordena los artículos de la consulta por id descendente
$articulos = Articulo::select('id','titulo')->get(); // método select() selecciona solo los campos id y titulo
```

- `$articulos = Articulo::take(3)->get(); //metodo take() coge solo los 3 primeros registros de la consulta`
- `$articulo = Articulo::select('titulo')->find('5'); //método find() busca por el id=5 sin usar where y devuelve un objeto`
- `$articulos = Articulo::where('id', '>', '20')->get(); //condición relacional en método where()`
- `$articulos = Articulo::where('titulo', 'like', '%amor%')->get()// cualquier título con la palabra amor dentro de su cadena.`
- `$articulos = Articulo::where('titulo', 'like', '%amor%')->delete()// borra los registros seleccionados en la consulta.`
- `$articulos = Articulo::select('articulos.titulo', 'categorias.nombre')->join('articulos', 'categoria.id', '=', 'articulos.categoria_id')->get(); //mostrar campos de 2 tablas con join //Si quisiéramos incluir también categorías sin artículos, usar leftjoin en vez de join`

<Video Unidad 9 Plantillas Blade en vistas> (a partir del minuto 7:50)
<Video Unidad 9 Métodos CRUD >
<Video Unidad 9 Modificar datos desde un formulario >

9. CONTROLADORES II – Métodos CRUD.

Los nombre usados normalmente para estos métodos son `index()` para mostrar el listado, `show()` para mostrar los datos de un registro, `create()` para mostrar el formulario de alta de un nuevo registro, `store()` para almacenar el registro en la bd, `edit()` para mostrar el formulario de actualización de un registro, `update()` para grabar los cambios de la actualización en la bd, `destroy()` para eliminar un registro de la bd. Si el parámetro que recibe el método es un objeto que representa un registro de la tabla anteponer al nombre del método el nombre del modelo para indicar que se recibe un objeto Ej: `function show(Producto $producto){}` ya que si no se antepone el modelo lo que se recibe solo es el Id, no el objeto completo

- **Método index()** En la ruta web principal se llamará al método index() del controlador que lista los registros de la tabla, y cada registro podrá ser consultado en otra página con todos sus datos y un botón para modificarlo o borrarlo. También suele haber en la página principal un enlace para insertar un registro nuevo.

```
public function index(){
  //$productos= Producto::all(); //obtenerlos todos
  //$productos= Producto::orderBy('fecha','desc')->get(); //todos ordenados por fecha
  $productos=Producto::orderby('updated_at','desc')->Paginate(5);//paginado de 5 en 5
  return view('productosIndex',compact('productos'));
}
```

  La función compact() permite definir un parámetro y su valor con una variable con el mismo nombre que el parámetro, en la sentencia anterior sin usar esta función, habría que definir un array de parámetros `return view('productosIndex',['productos'=>$productos]);`

- **Método show()** El método show devolverá una vista que muestre todos los datos del registro, a modo de consulta, por lo que hay que pasar a la vista el objeto con dichos datos, y normalmente desde esta vista tendremos enlaces para eliminar o actualizar el registro.

```
public function show(Producto $producto){
  //$producto=Producto::find($producto);//si recibimos el id, obtener el objeto con find()
  return view('productosShow',compact('producto'));
}
```

- **Método create()** El método para crear un nuevo registro devolverá una vista con un formulario en blanco con todos los campos de la tabla, y que mandará los datos al método store() por `method="post"`

(crear la ruta con post), y además es necesario incluir la directiva `@csrf` dentro del formulario de la vista.

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

- **Método store()** Para crear un registro en la tabla a partir de los datos de un formulario, crear en el controlador un método `store(Request $request){}` que recibe un objeto `$request` con todos los campos incluidos en el formulario, y que procesará la inserción del registro en la tabla de manera tradicional:

```
$variableObjeto=new nombreModelo();
$variableObjeto->propiedad1=$request->parametroForm1;
. . .
$variableObjeto->save();
```

O de manera masiva, con mucho menos código, para lo cual primero hay que añadir la propiedad `protected $guarded=[];` dentro de la clase del modelo, con los campos de la tabla que queramos que no se asignen de manera masiva, si no queremos proteger ningún campo se deja el array vacío [], y de esta manera se podrán asignar todos los campos recibidos del formulario al objeto en una sola sentencia, incluso quedando ya grabado en la BD sin tener que ejecutar el método save()

```
public function store(Request $request){
  $producto=Producto::create($request->all()); //inserta registro en BD con datos recibidos
  return redirect()->route('productos.show',$producto); //redirección automática a otra ruta
}
```

- **Método edit()** El método para editar un registro devolverá una vista con un formulario, cuyos campos estarán rellenos con todos los datos del registro, por lo que hay que pasar a la vista el objeto con dichos datos.

```
public function edit(Producto $producto)
  {
        return view('productosEdit',compact('producto'));
  }
```

- **Método update()** Para actualizar un registro desde un formulario se recomienda usar el método put (crear la ruta con put), para lo cual en el formulario html indicar el método post, pero además añadir la directiva `@csrf` y la directiva `@method('put')` dentro del formulario. Para guardar los datos recibidos del formulario en la BD de manera masiva usar la sentencia `$variableObjeto->update($request->all());`

```
public function update(Producto $producto, Request $request){
  $producto->update($request->all());//actualiza objeto en BD con los datos del formulario
  return redirect()->route('productos.show',$producto);
}
```

- **Método destroy()** Para eliminar un registro de la tabla hay que llamar al método destroy del controlador a través de un botón de un formulario y se recomienda usar el método delete (crear la ruta con delete) para lo cual en html indicar el método post, pero además añadir la directiva `@csrf` y la directiva `@method('delete')` dentro del formulario. Para borrar el objeto recibido de la BD, ejecutar `$variableObjeto->delete();` en el método destroy() del controlador

```
public function destroy(Producto $producto){
  $producto->delete();
  return redirect()->route('productos.index');
}
```

- **REDIRECCIONAMIENTOS**
  - A otra ruta: `return redirect('home/dashboard');`
  - A otra ruta con nombre: `return redirect()->route('nombreRuta', parametro);`

- A un método: `return redirect()->action([Controlador::class, 'index']);`
  O con parámetros: `return redirect()->action([Controlador::class,'profile'],['id'=>1]);`
- A dominio externo: `return redirect()->away('https://www.google.com');`
- A la página anterior `return redirect()->back();`

10. VISTAS II – Motor de plantillas Blade

1. Las vistas contienen el HTML que se sirve por cualquier aplicación y separa la lógica del controlador/aplicación de la lógica de presentación. Las vistas en Laravel están basadas en el motor de plantillas Blade, por lo que se crean con la extensión `'.blade.php'` Para devolver una vista al cliente desde un controlador, se usa la función view de la siguiente forma `return view('nombreVista', ['parametro' => 'valor']);` el nombre de la vista se indica sin la extensión y a partir de la carpeta `views/`
   Hay que crear las vistas que devuelvan cada uno de los controladores, generando archivos `metodoControlador.blade.php` (se aconseja formar nombre uniendo el nombre de la tabla y el método que la utiliza, sin carácter de unión, por ejemplo `productosIndex.blade.php`) en la carpeta `resources/views` Los parámetros pasados a la vista, se utilizan en el código html de la vista en la forma `{{$nombreParametro}}` o dentro de un script `<php? $nombreParametro ?>`
2. Herencia de plantillas. Se puede optimizar y reutilizar el código creando plantillas con la estructura común de las vistas en la carpeta `'resources/views/layouts'` con extensión `.blade.php`. En estas plantillas se dejan los huecos rellenables con `@yield('nombreAsignado')`. En las vistas donde se haga uso de las plantillas, solo hay que añadir al principio `@extends ('layouts.plantilla')` y definir los huecos de la plantilla con los valores, de la siguiente forma `@section('nombreAsignado', valor)` si es un valor simple, pero si el hueco a rellenar es un código más extenso se puede hacer en la forma `@section('nombreAsignado') … código a insertar en el hueco … @endsection`
3. Para aplicar estilo definido en archivos css a las vistas, ubicar los archivos css dentro de la carpeta `'public/css'` y para definir el link en la vista, usar la función asset de la siguiente forma: `<link rel="stylesheet" href="{{asset('css/estilos.css')}}">` donde la ruta pasada a la función asset es definida a partir de la carpeta public.
4. También se pueden crear componentes anónimos Blade en el proyecto (también hay componentes con nombre que no estudiaremos), para reutilizar bloques de código en distintas vistas añadiendo dichos componentes:

- Crear dentro de `'resources/views/components/nombreComponente.blade.php'` el archivo que será el componente con el contenido html que formará parte de la vista donde se incluya.
- Para incluir el componente solo hay que colocar la etiqueta `<x-nombreComponente />` en el lugar donde se debe ubicar dentro de la vista
- Componentes dinámicos con información recibida al colocar el componente en la vista:
  - Con parámetros: hay que colocar la etiqueta `<x-nombreComponente nombreParametro1='valor' nombreParametro2='valor' />` en la vista y en el componente hay que añadir al principio la sentencia `@props(['nombreParametro1'=>'valorPorDefecto', 'nombreParametro2'=>'valorPorDefecto'])` y ya se podrán colocar los parámetros en cualquier parte del componente con `{{nombreParametro1}}` que serán sustituidos por el valor recibido o por su valor por defecto si no se recibe el parámetro.
  - Con slots (bloques de texto), para incluir el componente en la vista hay que colocar la etiqueta `<x-nombreComponente name='nombreSlot'> texto del bloque </x-nombreComponente>` y en el componente se debe colocar la etiqueta `{{nombreSlot}}` en el lugar que queramos colocar el bloque de texto indicado en la vista. (los slots y los parámetros se pueden usar conjuntamente)

5. Estructuras de control Blade. Aunque podemos abrir un script de php y usar las estructuras propias del lenguaje, podemos usar estructuras que nos aporta Blade para no tener que imprimir los resultados con echo, sino escribirlos directamente en html.

- Condicional

```
@if (condicion1)
    Codigo1;
@elseif (condicion2)
    Codigo2;
@else
    Codigo3;
@endif

@switch($i)
    @case(1)
        Codigo1;
        @break
    @case(2)
        Codigo2;
        @break
    @default
        Codigo3;
@endswitch
```

- Bucles

```
@for ($i = 0; $i < 10; $i++)
    <h3>Estamos en la vuelta {{ $i }}</h3>
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

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

- Bloque de código php

```
@php
    Código php;
@endphp
```

11. PAGINACIÓN DE REGISTROS.

Paginar el listado de los registros obtenidos en una consulta: Usar el método `paginate(numero)` o `simplePaginate(numero)` en vez de `all()` para obtener la cantidad de registros correspondientes a la página solicitada. Para acceder a una página en concreto habría que acceder con el parámetro en la url `'?page=2'` para acceder a los registros de la página 2, pero esto lo gestiona laravel automáticamente si añadimos el siguiente componente en el lugar de la vista que nos interese, y así obtenemos los botones de navegación: `{{$variableCursor->links()}}` pero para que se vea correctamente en el navegador web o añadimos las librerías de tailwindcss o si preferimos usar Bootstrap debemos incluir las librerías cdn en la cabecera y también añadir en el archivo `App\Providers\AppServiceProvider` la sentencia `Paginator::useBootstrap();` dentro del método `boot()` y sustituir `use Illuminate\Contracts\Pagination\Paginator;` por `use Illuminate\Pagination\Paginator;` (si no estuviera la sentencia en vez de sustituir, simplemente añadir) y así se verá correctamente usando Bootstrap.

12. VALIDACIÓN DE FORMULARIOS.

Para validar los datos de un formulario en el método del controlador que recibe los datos incluir la sentencia `$request->validate(['nombreParametro1'=>'required|max:10|min:5|…', …]);` con cada campo-

restricción que haya que comprobar, al comienzo del método. Si alguna regla no se cumple volverá a la vista sin procesar los datos.
Para mostrar los errores en el formulario, hay que incluir junto a cada campo la directiva `@error('nombreParametro') {{message}} @enderror` y para cambiar los mensajes que están en inglés hay que modificarlo en `/resources/lang/en/validation.php` aunque se pueden descargar los ficheros (`validation.php, pagination.php, auth.php, paginations.php`) en español en la carpeta `/resources/lang/es/`, traducidos por la comunidad y especificar el idioma de la aplicación en español cambiando a `locale=>'es'`, en el fichero `config/app.php` Si además del mensaje quiero cambiar el nombre del parámetro en el mensaje, añadir al final de validation.php
`'attributes'=>['nombreParametro1'=>'nombre1Elegido', …]`
Y por último para que no se pierdan los datos en los campos cumplimentados correctamente, al validar un fallo en otro campo, hay que añadir en el atributo value de los campos del formulario con `value="{{old('nombreParametro', 'valorOriginal')}}"` donde nombreParametro hace referencia al parametro editado correctamente y valorOriginal al valor que debe tener por defecto al cargar la página, por si no se ha cumplido la validación, aunque si no existe ningún valor original por defecto no hay que especificar este valor.

13. MENÚ DE NAVEGACIÓN.

Crear un menú de navegación del sitio web donde indique en que sección me encuentro en todo momento. Para ello crear un partial (una parte común en varias páginas) que podamos incluir en las plantillas, en la carpeta `resources/views/layouts/partials/header.blade.php`, con el contenido `<header><nav>menú de navegación sitio web</nav></header>` y para incluirlo en la plantilla usar la directiva `@include('layouts.partials.header')` donde la ruta es a partir de la carpeta views y en el nombre del partial no hay que indicar la extensión. Dentro del header debemos colocar los enlaces del menú a las rutas del sitio web `<a href="{{route('home')}}">Home</a>` y para mostrar resaltado la opción de menú donde nos encontramos hay que indicar que el enlace pertenece a una clase con el estilo de resaltado `class="{{request()->routeIs('home*') ? 'active' : ''}}"` donde se pregunta con el método routeIs() de la clase request si nos encontramos en una ruta que comience por home (* es el carácter comodín para abarcar varías rutas de la misma opción del menú) para establecer active como valor del atributo class o vacío en caso contrario.

14. Componente Jetstream

Para obtener inicio de sesión, registro, etc… para lo cual es necesario tener instalado nodeJS

Para crear un proyecto nuevo con Jetstream ejecutar los siguientes comandos:

- `laravel new nombreProyecto --jet (elegir livewire cuando pregunte para usar blade)`
- `npm install && npm run dev (es necesario tener instalado nodejs)`
- `php artisan migrate (para crear las tablas en la bd)`
- `En config/jetstream.php se pueden configurar uso de grupos y de foto en el perfil.`

Para instalarlo en un proyecto existente ejecutar los siguientes comandos:

- `composer require laravel/jetstream`
- `php artisan jetstream:install livewire –teams (para elegir el entorno livewire)`
- `npm install && npm run dev (es necesario tener instalado nodejs)`
- `php artisan migrate (para crear las tablas necesarias en la bd)`

Con lo anterior ya tenemos habilitados en la página principal creada por defecto, los botones para iniciar sesión y registrarse y todo el código asociado, incluso configuración del perfil de usuario.

15. Control acceso a páginas con MIDDLEWARE

Para filtrar el acceso a una ruta con alguna condición distinta al inicio de sesión con jetstream se utilizan middlewares, primero hay que crearlo en la carpeta `app/http/middlewares` con el comando `php artisan make:middleware nombeMiddleware` También hay que registrar el middleware en el archivo `app/http/Kernel.php` incluyendo la sentencia para registrarlo
`'aliasMiddleware'=>\App\Http\Middleware\nombreMiddleware::class`
Para asociar el middleware al método que se tiene que filtrar, se puede hacer de 2 formas distintas:

- Añadir a la ruta definida en `routes/web.php` que se quiere filtrar, el método middleware()
  `Route::get('ruta', [HomeController::class, 'metodo'])->middleware('aliasMiddleware');` si hubiera que cumplir más de un middleware indicarlos en un array `(['alias1', 'alias2',..])`.
- Añadir un método constructor en el controlador asociando los middlewares a los métodos

```
public function __construct(){
        $this->middleware('auth'); //asociado a todos los métodos (sesión iniciada)
        $this->middleware('log')->only('index'); //asociado solo al método index
        $this->middleware('subscribed')->except('store'); // a todos menos store
}
```

Por último, tenemos que poner la condición de filtrado en el método handle del middleware que hemos creado

```
if($request->parametro > valor){ //condición de filtrado
    return $next($request);
}else{
    return redirect('nombreRuta'); //ruta donde dirigir si no cumple filtrado
}
```

16. Crear un dominio virtual en local para acceso más cómodo desde el navegador

Editar como administrador el archivo

```
C:\Windows\System32\drivers\etc\hosts
```

Añadir la línea

```
120.0.0.1       nombreDominio
```

Guardar y cerrar. Editar el archivo

```
C:\xampp\apache\conf\extra\httpd-vhosts.conf
```

Añadir una única vez el siguiente bloque

```
NameVirtualHost *
<VirtualHost *>
        DocumentRoot "C:\xampp\htdocs"
        ServerName localhost
</VirtualHost>
```

Y añadir el siguiente bloque por cada dominio a incluir

```
<VirtualHost *>
        DocumentRoot "C:\xampp\htdocs\LARAVEL\carrito\public"
        ServerName carrito
        <Directory "C:\xampp\htdocs\LARAVEL\carrito\public">
                Options All
                AllowOverride All
                Require all granted
        </Directory>
</VirtualHost>
```