1. INTRODUCCION

Laravel es un framework de código abierto escrito en PHP utilizado para desarrollar aplicaciones y servicios web.

Su filosofía es crear código PHP de forma elegante y sin código espagueti (es un término peyorativo para los programas de computación que tienen una estructura de control de flujo compleja e incomprensible).

Un framework es un entorno de trabajo con código pre-escrito por una comunidad de expertos programadores, proporcionando ventajas a la hora de desarrollar código, ya que evita que tengamos que escribir todo el código desde 0, y ayudando en tareas como el acceso a base de datos, validaciones, etc y generando buenas prácticas a la hora de programar.

Laravel utiliza el patrón MVC que permite la separación de los datos y la lógica de nuestra aplicación de la vista o interfaz que ve el usuario, ordenando el código.

Con Laravel creamos proyectos de forma más rápida, limpia y segura. Laravel utiliza el motor de plantillas Blade.

2. INSTALACION DE LARAVEL 9

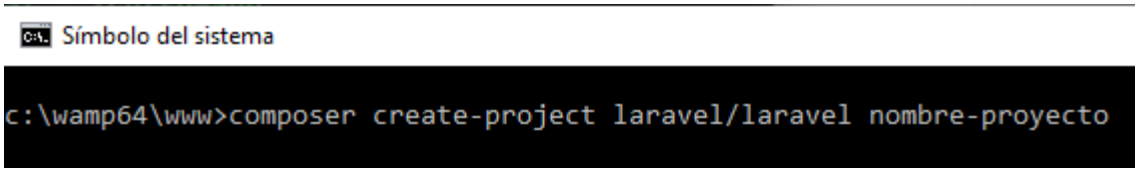
La versión de Laravel que vamos a trabajar es la versión 9. Los requerimientos para usar Laravel 9 son:

- PHP >= 8.0 con las extensiones:
    - php_ctype, curl, php_json, fileinfo
    - si disponemos de wamp, mamp o xamp: mbstring, openssl, pdo_mysql, php_tokenizer

  Aquellas extensiones que necesitemos, las descargaremos de internet (las extensiones son ficheros de extensión .dll) y las copiaremos en la carpeta ext de la versión de PHP que estemos utilizando.
- un servidor MySQL
- el gestor de paquetes Composer. Deberemos instalarlo para la versión de PHP a utilizar.
- Una vez tengamos todo esto instalemos Laravel al crear cada proyecto.

Laravel nos provee además de un **servidor web local para desarrollo** llamado **artisan**, creado en PHP por lo que no necesitaríamos tener instalado otro servidor web. También podemos instalar un servidor web como Apache u otro. El servidor web que utilicemos tiene que tener activado el **módulo rewrite** y la **extensión PDO** para acceso a bases de datos.

Instalación de un proyecto vacío de Laravel empleando la terminal del sistema operativo:



Si tenemos una versión de PHP superior a la 8.0 este comando nos descargará e instará la última versión de Laravel y nos creará nuestro proyecto. Si queremos utilizar wamp como servidor web tendremos que crear el proyecto dentro de c:\wamp64\www.

Para comprobar desde la consola **la versión e PHP** que tengo instalada:

```
c:\wamp64\www\restapi>php -v
PHP 8.1.0 (cli) (built: Nov 23 2021 21:48:28) (ZTS Visual C++ 2019 x64)
Copyright (c) The PHP Group
Zend Engine v4.1.0, Copyright (c) Zend Technologies
```

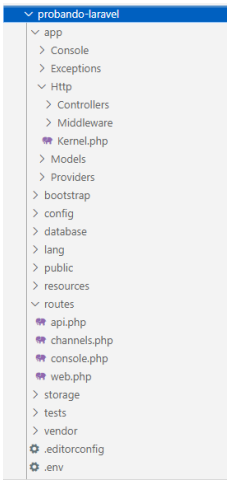
Para saber qué versión de Laravel he instalado:

```
c:\wamp64\www\rolespruebas>php artisan -V
Laravel Framework 9.48.0
```

Para ver todos los comandos disponibles en artisan:

```
c:\wamp64\www\rolespruebas>php artisan list
Laravel Framework 9.48.0
```

Entramos en la carpeta que hemos creado y la abrimos en Visual Studio Code y veremos la estructura de Laravel instalada:



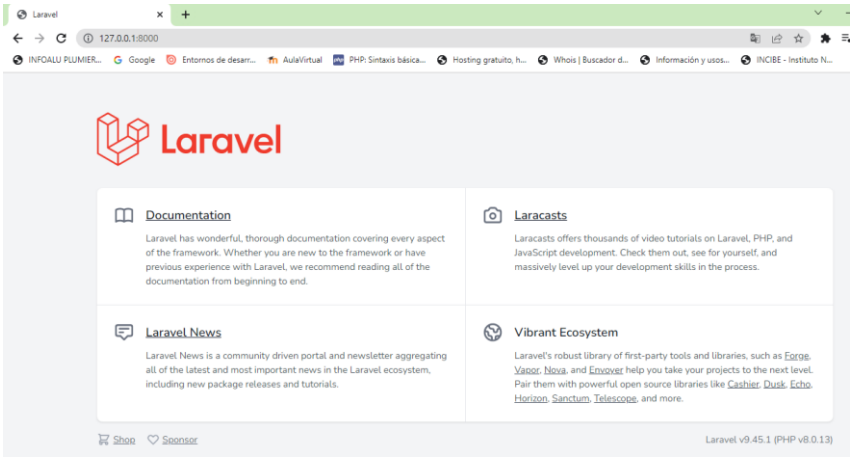
Para levantar el servidor web de Laravel **artisan** ejecutaremos el siguiente comando desde el intérprete de comandos:

```
c:\wamp64\www\probando-laravel>php artisan serve

  INFO  Server running on [http://127.0.0.1:8000].

  Press Ctrl+C to stop the server
```

A partir de este momento podemos ejecutar nuestro proyecto desde el navegador con la URL: http://localhost/probando-laravel/public/
O desde artisan:
http://127.0.0.1:8000



La vista que nos muestra se encuentra en resources/views/welcome.blade.php

3. ESTRUCTURA DE CARPETAS DE LARAVEL

Laravel usa un único punto de acceso a la aplicación llamado **Front Controller.**

Hay varios ficheros ocultos, entre ellos .env, que declara las variables de entorno que utiliza el proyecto.

Carpetas importantes:

- **App** contiene todo el código fuente de la aplicación.
  - App/Http/**Controllers**: controladores del proyecto.
  - App/**Models**: modelos Eloquent del proyecto para interactuar con al BD. Por defecto disponemos del model User para trabajar con usuarios.
- **Bootstrap:** contiene app.php que arranca el framework
- **Config**: ficheros de configuración de la aplicación
  - App.php contiene los parámetros de configuración como la url, zona horaria, idioma, etc
  - Database.php configuración a conexión de base de datos
  - Hashing.php permite la encriptación de datos
  - Logging.php permite almacenar los logs del proyecto
  - Mail.php configuración de emails
  - Queue.php para trabajar con colas de trabajos.
  - Sanctum.php configuración para la nueva autenticación de laravel.
  - Services.php configuración de servicios externos que utilicemos .
  - Sesión.php sesiones que utilicemos en nuestro proyecto.
- **Database**:
  - Migrations: ayudan a estructurar la base de datos. Podemos crear tablas por código. Por ejemplo, por defecto, entre otros, tenemos create_users_table.php
  - Seeders: nos ayuda a rellenar la base de datos con información ficticia.
  - Factories: para generar información en nuestra base de datos
- **public**: es el directorio público del proyecto.

  En public se encuentra el fichero **index.php** o **Front Controller**, **que recibe** todas las peticiones de los usuarios. La redirección es posible gracias al módulo rewrite y al archivo .htaccess que se encargan de ello. Para redirigir peticiones a otra página no creamos un fichero .php, sino que debemos crear una ruta.

  En esta carpeta public debemos colocar nuestros css, ficheros javascript e imágenes, todo aquello que sea accesible de forma pública.

  Si creamos un virtual host debemos apuntarlo a esta carpeta y no a la raíz del proyecto. Fuera de esta carpeta la información debe ser protegida.
- **resources**: contiene las vistas donde se encuentra todo el frontal de la aplicación y los ficheros css y javascript no compilados que tras ser compilados acabarán en la carpeta public.
  - **Views**: contiene las vistas del proyecto. Las vistas están implementadas utilizando Blade. Blade es un lenguaje .. Las vistas deben tener un nombre con la extensión blade.php. En las vistas utilizaremos {{ }} en lugar de <?php ?>

- **routes**: contiene el fichero web.php con todas las rutas del proyecto. Una ruta es la dirección que ponemos en el navegador para ir a una parte de la aplicación. El fichero api.php contiene otro tipo de rutas que no contienen sesiones ni cookies.
- **Test**: permiten probar las funcionalidades de la aplicación.
- **Vendor**: contiene los paquetes con las dependencias de terceros que instalemos. No se debe tocar.

4. RUTAS

Las rutas es lo primero que se carga cuando accedemos a una aplicación Laravel.

Laravel utiliza el **patrón Front-Controller,** el cual es un patrón de diseño que se basa en usar un controlador frontal como punto inicial para la gestión de las peticiones. El punto de entrada en nuestra aplicación se encuentra en **public/index.php**. El controlador frontal maneja las peticiones y respuestas de la aplicación mediante el concepto de rutas.

Las **rutas en Laravel** son una de las capas más importante en el Framework, es un sistema de rutas que **se encargan de manejar el flujo de solicitudes y respuestas, desde y hacia el cliente** (hacia el navegador, por ejemplo). Las rutas permiten a Laravel saber en cada momento a qué parte de la aplicación ir.

Las rutas se encuentran en la carpeta routes. En esta carpeta hay dos ficheros de rutas:

- **web.php**: aquí se definen las rutas web (tienen estado)
- **api.php**: aquí se definen las rutas para crear APIs (lo veremos más adelante)

En el fichero routes/web.php iremos añadiendo nuestras rutas de momento. Las rutas definen la dirección URL, el método por el cual se puede ingresar a dicha ruta (GET, POST, etc.) y la acción a realizar.

```php
<?php

use Illuminate\Support\Facades\Route;

/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/

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

La ruta principal del proyecto por defecto es la url / y cargará la vista welcome.blade.php ubicada en resources/views

Esta simple línea de código le dice a Laravel que al realizar una petición GET sobre la URL **/** se ejecute la función anónima que carga la vista welcome.blade.php (las vistas están ubicadas en resources/views).

- **Route** es una interfaz estática que permite definir rutas.
- **get es un verbo HTTP** que acepta dos parámetros, la URL que se llamará desde el navegador y una función anónima que devuelve lo que queremos mostrar.
- **View** es una función helper que sabe dónde están ubicadas nuestras vistas (apunta a resources/views). En caso de que tengamos subcarpetas dentro de la carpeta views, deberemos indicarlas. Por ejemplo:

```php
Route::get('/', function () {
    return view('mi_carpeta.mi_vista');
});
//Equivale a
Route::get('/', function () {
    return view('mi_carpeta/mi_vista');
```

```
});
```

Observa qué pasa si hacemos:

```
Route::get('/', function () {
    // return view('welcome');
    return "Hola, bienvenido a mi sitio web";
});
```

Se retornará la cadena "Hola, bienvenido a mi sitio web"

Si hacemos:

```
Route::get('/', function () {
    return ["Modulo" => "DWES"];
});
```

Se retornará el array en formato json

Las vistas tienen la extensión blade.php, lo que significa que usamos el motor de plantillas BLADE que veremos más adelante.

MÉTODOS EN LAS RUTAS

Route proporciona varios métodos: (verbos Http)

- get: para responder a peticiones GET
- post: para peticiones POST
- view: cuando no se necesita ejecutar nada entre la petición GET y la respuesta. View responde a las peticiones de tipo get y head.
- Otros: head, patch, put, options, delete, match cuando queremos responder a varios métodos indicados y any cuando queremos responder a todos los tipos de peticiones.

Por ejemplo:

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

Carga la vista welcome.blade.php

URL

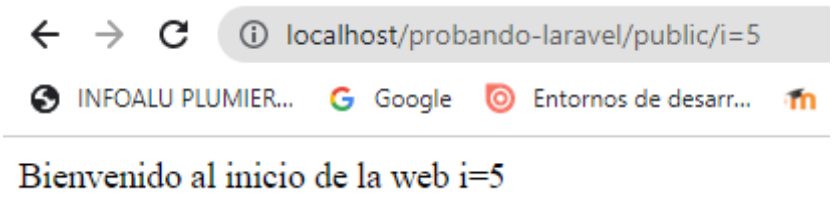
Nombre de la vista

RUTAS CON PARÁMETROS

Para crear rutas más complejas, que requieran de parámetros dinámicos se pueden definir de la siguiente forma:

```
Route::get('/{parametro}', function ($parametro) {
    return 'Bienvenido al inicio de la web '.$parametro;
});
```

Si ejecutamos: http://127.0.0.1:8000/i=5 obtendremos:



Estamos diciendo que la ruta es de tipo GET, por tanto el parámetro se envía en la URL, y para ello utilizaremos llaves {} y el nombre del parámetro que enviaremos. Las llaves nos permiten definir uno o más parámetros dinámicos y pasarlos como argumentos a las funciones a ejecutar.

Se pueden usar tantos parámetros como sean necesarios, solo es importante que estén encerrados entre llaves {} y los nombres pueden ser alfanuméricos pero no está permitido usar el guión - pero sí el subrayado _. Además, importa el orden de los parámetros pasados a la función anónima, pero no los nombres que se les de.

ORDEN EN LAS RUTAS

Cuando un usuario hace una petición HTTP, Laravel busca en los archivos de rutas una definición que coincida con el patrón de la URL según el método HTTP usado y en la primera coincidencia le muestra el resultado al usuario. Por tanto el orden de precedencia de las definiciones de rutas es muy importante. En general debemos colocar al final del fichero las rutas que contengan parámetros.

RUTAS CON RESTRICCIONES

Podemos solucionar los posibles conflictos con el parecido en la URL de distintas rutas de 2 maneras:

- Usando el **método where** para agregar condiciones de expresiones regulares a la ruta:

```
Route::get('aplicacion/{id}/editar', function ($id) {
    return 'Aquí podremos editar el elemento: '.$id;
})->where('id', '[0-9]+');   // o ->where('id', '\d+')
```

- **Ordenando las rutas** de tal manera que las más específicas estén al principio y las más generales al final del archivo de rutas.

RUTAS CON NOMBRE

Es recomendable dar un nombre a las rutas, e invocar a las rutas por dicho nombre usando la función route() en lugar de por la URL. Los nombres se usan para invocar a las rutas desde otros sitios, por ejemplo en las redirecciones (en el controller):

```
Route::get('/aplicacion', function () {
    return 'Bienvenido a esta aplicación';
})->name('nombreRuta');
```

Invocamos a la ruta para una redirección:

```
return redirect()->route('nombreRuta');
```

En una vista en el action de un formulario, en un enlace, etc:

```
 <form action="{{ route('nombreRuta') }} … >
```

Si cambia la URL no afecta al código pues estamos usando el nombre de la ruta.

RUTAS CON PARAMETROS OPCIONALES

Cuando un parámetro no es obligatorio podemos usar el carácter **?** al final del nombre del parámetro para indicar que es opcional y darle un valor por defecto al parámetro cuando lo enviamos a una función:

```
Route::get('/{parametro?}', function ($parametro = null) {
      if ($parametro) {
          return 'Bienvenido al inicio de la web, el parametro es '.$parametro;
      } else {
          return 'Bienvenido al inicio de la web';
      }
});
```

PROTEGER RUTAS

Cuando una ruta debe ser protegida y que solamente un usuario autenticado tenga acceso a ella, utilizamos los middleware, por ejemplo:

```
Route::get('/aplicacion', function () {
    return 'Bienvenido a esta aplicación';
})->name('nombreRuta')->middleware('auth');
```

5. MOTOR DE PLANTILLAS BLADE

Laravel utiliza el **motor de plantillas Blade**, que nos permite escribir HTML estándar que será visualizado tal cual en el navegador, e insertar código PHP.

Para usar Blade las vistas deben terminar su nombre con la **extensión blade.php**, de esta forma Laravel sabe que es un archivo de Blade y lo compila como tal.

Para mostrar datos de variables en Blade utilizamos **{{ $variable }}** , que equivale a <?php echo $variable ?>

Laravel hace el escapado de todos los caracteres especiales de HTML, para evitar ataques inyección de código XSS. [Consultar: https://diego.com.es/ataques-code-injection-en-php **]** Para mostrar código sin escapar la sintaxis a utilizar es: **{!! $cadena_codigo_html !!}**

Para utilizar las llaves dobles en Blade la sintaxis a emplear es: **@{{ variable }}**

Laravel **compila las vistas de Blade a PHP plano**, ya que PHP no reconoce las {{ }}, es decir las vistas que se encuentran en **resources/views** se transforman en PHP plano, y se ubican en la carpeta **storage/framework/views** que nos muestra finalmente.

Para **insertar comentarios** código en Blade se utiliza **{{-- código a comentar --}}**

Estructuras de control en Blade

```
@if ($manuales->count() > 0)
   Tenemos manuales que mostrar!
@endif
```

```
@if ($dia == "sabado")
   Es sabado!!
```

```
@elseif ($dia == "domingo")
   Es domingo!!
@else
   No estamos en final de semana
@endif
```

```
@switch($estado)
   @case("a")
      Estado es "a"
      @break
   @case("b")
      El estado es "b"
      @break
   @default
      Es el código que se ejecutará si no era ninguno de los estados anteriores.
@endswitch
```

```
@foreach($certificado as $cert)
    <article class="Cert-card">
       <h3>{{ $cert->nombre }}</h3>
       <div>{{ $cert->descripcion }}</div>
    </article>
@endforeach
```

```
@forelse($certificado as $cert)
    <article class="Cert-card">
       <h3>{{ $cert->nombre }}</h3>
       <div>{{ $cert->descripcion }}</div>
    </article>
@empty
     <div>No hay certificado </div>
@endforelse
```

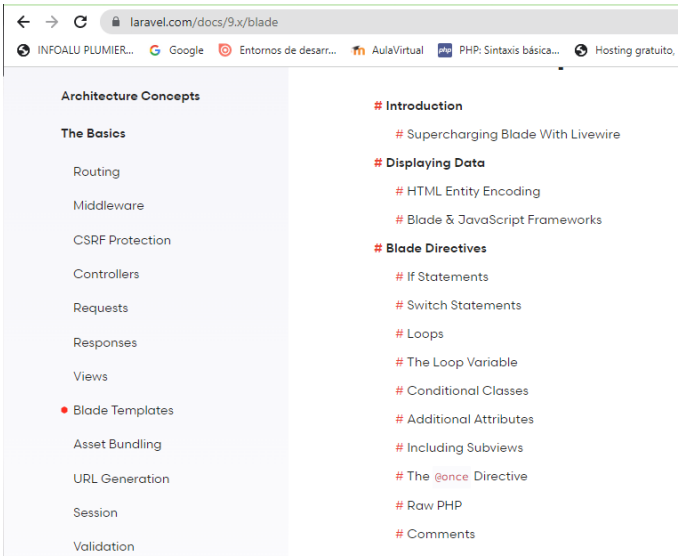
Para ejecutar código PHP dentro del Blade empleamos:

```
@php
   $salida = $venta->obtenerDatos();
   if($salida["estado"] == "canceled") {
  $cancelado = true;
   } else {
  $cancelado = false;
   }
@endphp
```

```
@isset($variable)
  …
@endisset
```

Más adelante veremos cómo trabajar con formularios en Blade.

En la documentación de Laravel encontramos cómo trabajar con Blade:



Consulta en https://laravel.com/docs/9.x/blade#if-statements las estructuras disponibles en Blade.

Vamos a hacer un pequeño ejemplo para ver cómo añadir la navegación en blade:

1. Creamos estas rutas (fichero web.php)

```php
//Ejemplo navegacion
Route::view('/','welcome');
Route::view('/sobremi','viewsobremi')->name('sobremi'); //recomendable
Route::view('/cursos','viewcursos')->name('cursos');
Route::view('/compañeros','viewcompañeros');
```

2. La vista welcome.blade.php tendrá un pequeño menú de navegación:

resources > views > welcome.blade.php > ...

```
1  <!DOCTYPE html>
2  <html lang="{{ str_replace('_', '-', app()->getLocale()) }}">
3      <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="width=device-width, initial-scale=1">
6
7          <title>Página principal</title>
8
9      </head>
10     <body>
11         <ul>
12             <li><a href="/">Inicio</a></li>
13             <li><a href="{{ route('sobremi') }}">Sobre mi</a></li> <!-- recomendable -->
14             <li><a href="/compañeros">Mis compañeros</a></li>
15             <li><a href="/cursos">Módulos que imparto</a></li>
16         </ul>
17
18     </body>
19 </html>
```

Menú navegación

3. Debemos crear una vista para cada ruta.
4. El **menú de navegación** solo está en welcome y queremos que esté presente en las demás vistas. Para no copiar y pegar código (por los errores que se producen si cambia el menú) crearemos una carpeta que contendrá una vista con el menú de navegación nav.blade.php. En las demás vistas tendremos un include de este menú de navegación con directivas de blade.

   Vista nav.blade.php

resources > views > layouts > nav.blade.php > ...

```blade
<ul>
    <li><a href="/">Inicio</a></li>
    <li><a href="{{ route('sobremi') }}">Sobre mi</a></li> <!-- recomendable -->
    <li><a href="/compañeros">Mis compañeros</a></li>
    <li><a href="/cursos">Módulos que imparto</a></li>
</ul>
```

Vista welcome.blade.php

resources > views > welcome.blade.php > ...

```blade
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">

        <title>Página principal</title>

    </head>
    <body>
        @include('menu.nav')
    <h1> Inicio </h1>
    </body>
</html>
```

Resto de vistas:

resources > views > sobremi.blade.php > ...

```blade
<!DOCTYPE html>
<html lang="en">
    <head>
        <title></title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link href="css/style.css" rel="stylesheet">
    </head>
    <body>
    @include('menu.nav')
    <h1>Sobre mi</h1>
    <p> Me llamo Susana Rosa y doy clase en el IES Ribera de los Molinos </p>
    </body>
</html>
```

PLANTILLAS DE BLADE

Otra forma de trabajar con Blade es **utilizar las plantillas** mediante herencia o mediante componentes.

HEREDAR DE PLANTILLAS DE BLADE

Las plantillas las guardaremos en una carpeta llamada **layouts** dentro de views. Por convención, dentro de layouts se suele crear dos plantillas, la plantilla **app.blade.php** que contendrá todo lo que queremos reutilizar. La otra plantilla se suele llamar guest.blade.php y se usa para usuarios no autenticados. Lo veremos más adelante.

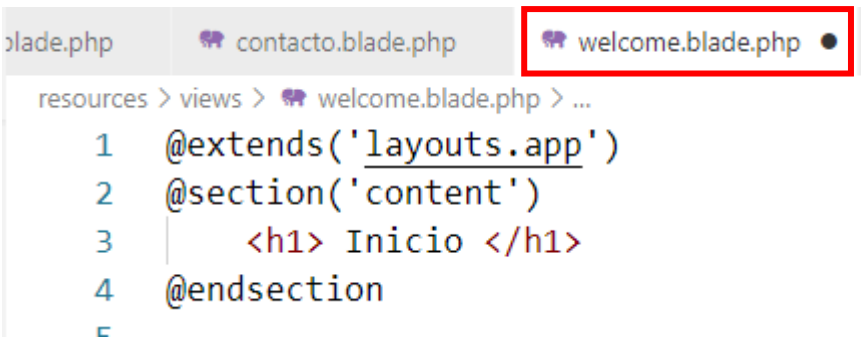
En esta plantilla **app.blade.php** colocaremos:

resources > views > layouts > app.blade.php > ...

```
1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="width=device-width, initial-scale=1">
6
7          <title>Página principal</title>
8
9      </head>
10     <body>
11        @include('menu.nav')
12        @yield('content')
13
14     </body>
15 </html>
```

Nombre de la sección que queremos incluir. Por convección se llama **content**

Para hacer uso de la plantilla utilizamos la directiva @extends y con @section damos nombre a la sección que será incluida donde hayamos colocado la directiva @yield en la plantilla:



Para que cada página tenga su propio título, añadiremos @yield('title') en la plantilla y @section('title','texto titulo') en cada página:

Plantilla app.blade.php

```
resources > views > layouts > app.blade.php > ...
1  <!DOCTYPE html>
2  <html lang="en">
3      <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="wid
6          <title> @yield('title')</title>
7      </head>
8      <body>
9         @include('menu.nav')
10        @yield('content')
11     </body>
12 </html>
```

Vista welcome welcome.blade.php

```
welcome.blade.php ×  sobremi
rces > views > welcome.blade.php > ...
1  @extends('layouts.app')
2
3  @section('title')
4      Página principal
5  @endsection
6
7  @section('content')
8      <h1> Inicio </h1>
9  @endsection
0
```

Vista sobremi sobremi.blade.php

```
p  welcome.blade.php  sobremi.blade.php ●
ources > views > sobremi.blade.php > ...
1  @extends('layouts.app')
2  @section('title')
3      Sobre mi
4  @endsection
5
6  @section('content')
7      <h1> Sobre mi </h1>
8      <p> Me llamo Susana Rosa y
9          doy clase en el IES
10         Ribera de los Molinos
11     </p>
12 @endsection
```

En este caso podemos escribir de dos formas la sección title:

`@section('title')`
`    Sobre mi`
`@endsection`
	`@section('title', Sobre mi)`

También es importante añadir a nuestra plantilla la etiqueta `<meta name="name-description" content="@yield('description','Valor por defecto')" />`, donde el valor por defecto se emplea en caso de que no se defina esta sección. Para sobreescribirla en cada página añadiremos la sección @section('name-description', 'Valor de la descripción')

6. PASAR DATOS A LAS VISTAS

Podemos pasar datos a las vistas a través de las rutas. Por ejemplo, dada la siguiente ruta:

```
Route::view('/cursos','viewcursos', ['modulos'=> ["DWES","ENTORNOS","PROGRAMACIÓN"]])-
>name('cursos');
```

URL | Nombre vista | Datos que pasamos a la vista

Podemos inspeccionar el valor de la variable $**modulos** en la vista "**viewcursos**.blade.php" con la directiva de Blade **@dump**:

```
@extends('layouts.app')
@section('title')
   Cursos que imparto
@endsection
@section('name-descripcion','Cursos de 1DAW')

@section('content')
    <h1> Módulos que imparto</h1>
    @dump($modulos)
    <ul>
    @foreach($modulos as $modulo)
        <li>{{ $modulo }}</li>
    @endforeach
    </ul>
@endsection
```

Este mismo ejemplo con una ruta get sería:

```
Route::get('/cursos', function () {
    $modulos = ["DWES","ENTORNOS","PROGRAMACIÓN"];
    return view('viewcursos',['modulos' => $modulos]);
});
```

Para mantener limpio el archivo de rutas se recomienda utilizar controladores que serán los que contengan la lógica de la aplicación.

7. CONTROLADORES

Un controlador es una clase PHP que recibe la petición del usuario y devuelve una respuesta. Las peticiones las formulamos en el fichero de rutas. Los controladores se ubican en App/Http/Controllers.

Laravel proporciona herramientas a través de comandos para crear controladores. Desde la terminal dentro de nuestro proyecto haremos:

URL

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

Se recomienda utilizar las convenciones de Laravel. El nombre del controlador deberá definirse con la primera letra en mayúscula y con la palabra Controller. Por ejemplo, UserController.php

Para invocar a un controlador desde una ruta get haremos:

```
Route::get('URL', [nameController::class,'accion'])->name('nameRuta');
```

En este caso, el controlador tendrá una función de nombre 'accion' sin parámetros.

```
Route::get('URL/{param}', [nameController::class,'accion'])->name('nameRuta');
```

En este caso, el controlador tendrá una función de nombre 'accion' que recibe un parámetro.

```
Route::post('URL', [nameController::class,'accion'])->name('nameRuta');
```

En este caso, el controlador tendrá una función de nombre 'accion' que recibe un parámetro de tipo Request, el cual contiene los datos procedentes de un formulario normalmente.

No deberemos olvidar importar la clase del contolador en el fichero de rutas.

Ejemplo: sitio web que muestra un listado de videojuegos

1. Creamos un controlador GameController:

```
c:\wamp64\www\probando-laravel>php artisan make:controller GameController

  INFO  Controller [C:\wamp64\www\probando-laravel\app/Http/Controllers/GameController.php] created successfully.
```

2. Creamos 3 rutas, una para listar los juegos, otra para mostrar un formulario para dar de alta nuevos juegos y otra para ver el detalle de un juego:

www > probando-laravel > routes > web.php > ...

```
1  <?php
2
3  use Illuminate\Support\Facades\Route;
4  use App\Http\Controllers\GameController;
5  /*
6  |--------------------------------------------------------------------------
7  | Web Routes
8  |--------------------------------------------------------------------------
9  |
10 | Here is where you can register web routes for your application. These
11 | routes are loaded by the RouteServiceProvider within a group which
12 | contains the "web" middleware group. Now create something great!
13 |
14 */
15
16 Route::get('/', function () {
17     return view('welcome');
18 });
19
20 Route::get('/games', [GameController::class,'index']);
```

Debemos importar los controladores que utilicemos en el fichero de rutas

Ruta

Invoca al Controlador

Función del controlador

Esta simple línea de código le dice a laravel que al realizar una petición GET sobre la ruta **/games** se ejecute el método **index** del controlador **GameController**.

```
Route::get('/games/create', [GameController::class,'create']);

Route::get('/games/{name}/{category?}, [GameController::class,'show']);
```

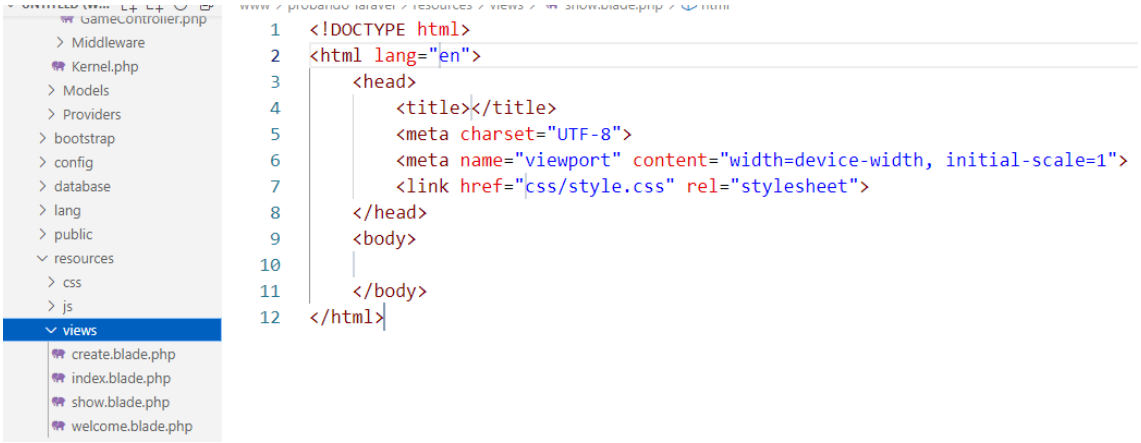
3. Creamos las funciones del controlador:

```
//Funciones del controller
    public function index(){
       return view('index');
    }

    public function create(){
       return view('create');
    }
     //las variables name y cat estarán disponibles en la vista show
    public function show($name_game, $categorie_game= null){
       return view('show', [‘name’=>$name_game, ‘cat’=>$categorie_name]);
    }


//Otra versión de index puede ser:
      public function index(){
        $videojuegos = array('Fifa 22','Mario Kart', 'MineCraft','NBA 22');
        return view('index',['listagames'=>$videojuegos]);
      //dd($videojuegos); //similar a var_dump
       }
```

4. Creamos las 3 vistas:

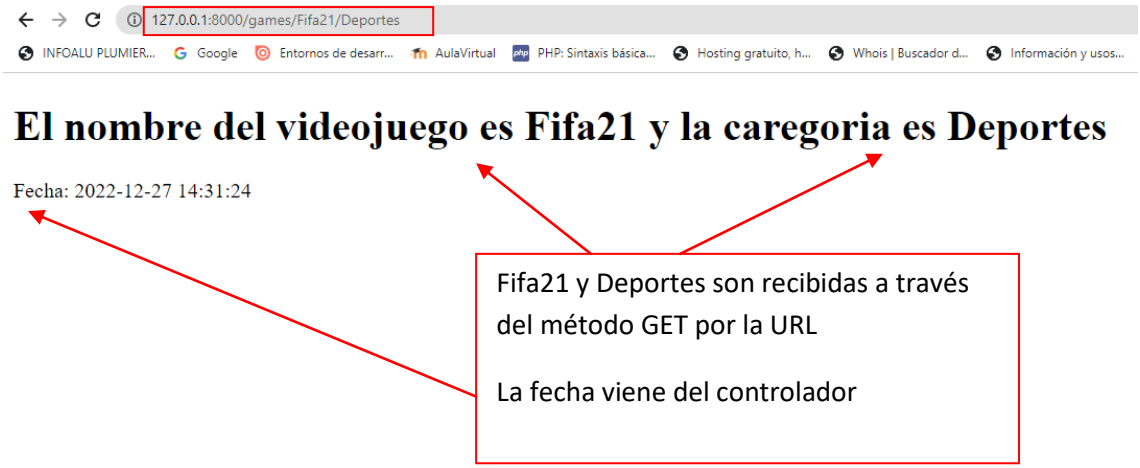


Veamos por ejemplo la vista **show.blade.php**:

www > probando-laravel > resources > views > show.blade.php > ...

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <title>Probando Laravel</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link href="css/style.css" rel="stylesheet">
    </head>
    <body>
        @if ($cat)
            <h1> El nombre del videojuego es {{$name}} y la caregoria es {{$cat}}</h1>
        @else
            <h1> El nombre del videojuego es {{$name}} </h1>
        @endif
    Fecha: {{$date}}
    </body>
</html>
```

Por tanto cuando probamos esta ruta:



Ahora, a la vista index vamos a pasar desde el controlador una lista de videojuegos en un array (más adelante estos datos los traeremos de una base de datos).

```
class GameController extends Controller
{
    public function index(){
        $videogames = array('Fifa21','NBA 22','Mario Kart','Super Mario');
        return view('index',['gamesList' => $videogames]);
    }
```

www > probando-laravel > resources > views > index.blade.php > html > body > ul > li

```
1  <!DOCTYPE html>
2  <html lang="es">
3      <head>
4          <title>Probando Laravel</title>
5          <meta charset="UTF-8">
6          <meta name="viewport" content="width=device-width, initial-scale=1">
7          <link href="css/style.css" rel="stylesheet">
8      </head>
9      <body>
10     <h1> Bienvenido a la web que listará los juegos comprados. </h1>
11     @if($gamesList)
12         <h2> Listado de juegos </h2>
13         <ul>
14             @foreach($gamesList as $game)
15                 <li>{{$game}}</li>
16             @endforeach
17         </ul>
18     @else
19         <h3> No hay juegos </h3>
20     @endif
21     </body>
22 </html>
```

```
<h2> Listado de juegos </h2>
<ul>
    @forelse($gamesList as $game)
        <li>{{$game}}</li>
    @empty
        <li> No hay datos </li>
    @endforelse
</ul>
```

En las vistas podemos añadir cualquier elemento HTML5. En el caso de los enlaces para indicar la URL a la que acceder utilizaremos la función **route** a la que indicaremos el nombre de la ruta.

Por ejemplo:

```
<a href="{{ route ('nombreRuta') }}"> Enlace a ... </a>
```

8. ACCESO A BASE DE DATOS EN LARAVEL

Laravel admite los siguientes servidores de bases de datos:

- MariaDB 10.2+
- MySQL 5.7+
- PostgreSQL 9.6+
- SQLite 3.8.8+
- SQL Server 2017+

Para configurar Laravel para trabajar con una base de datos iremos a la carpeta **config**, al fichero **database.php**. Por defecto Laravel viene configurado para trabajar con mysql, tal y como indica la línea:

```
'default' => env('DB_CONNECTION', 'mysql'),
```

La función **env** buscará en el fichero oculto **.env** las variables de entorno del proyecto. La línea anterior busca la variable DB_CONNECTION en el fichero .env y toma su valor. En caso de que dicha variable no estuviese definida en el fichero .env tomaría el valor indicado en database.php.

Es recomendable especificar en el fichero .env las variables de entorno de la aplicación, ya que este fichero es oculto e intransferible. El fichero database.php tomará los valores especificados del fichero .env. De esta forma, podríamos publicar en github el fichero Database.php pero nuestros datos permanecerían protegidos en el fichero .env. En despliegues en producción el fichero .env se crea insitu en el servidor o a través de herramientas de despliegue como Jenkins o Railway.

En el fichero .env se especifican las siguientes variables de entorno, con las que vamos a trabajar:

```
11  DB_CONNECTION=mysql
12  DB_HOST=127.0.0.1
13  DB_PORT=3306
14  DB_DATABASE=laravel //games
15  DB_USERNAME=root
16  DB_PASSWORD=
17
```

Aquí pondremos el nombre de la Base de datos a utilizar

Laravel es capaz de trabajar con varias bases de datos a la vez. En este ejemplo, crearemos una base de datos vacía llamada **games** y la indicaremos en la sección anterior dentro del fichero .env.

Antes de crear la base de datos modificaremos las siguientes líneas de nuestro fichero database.php del proyecto:

```
'mysql' => [
    'driver' => 'mysql',
    'url' => env('DATABASE_URL'),
    'host' => env('DB_HOST', '127.0.0.1'),
    'port' => env('DB_PORT', '3306'),
    'database' => env('DB_DATABASE', 'forge'), //usaremos games
    'username' => env('DB_USERNAME', 'forge'),
    'password' => env('DB_PASSWORD', ''),
    'unix_socket' => env('DB_SOCKET', ''),
    'charset' => 'utf8',
    'collation' => 'utf8_spanish2_ci',
    'prefix' => '',
    'prefix_indexes' => true,
    'strict' => true,
    'engine' => 'InnoDB',
    'options' => extension_loaded('pdo_mysql') ? array_filter([
        PDO::MYSQL_ATTR_SSL_CA => env('MYSQL_ATTR_SSL_CA'),
    ]) : [],
],
```

9. ELOQUENT

Eloquent es el ORM de Laravel. Un **ORM** (Object Relational Mapping) es un modelo de programación que permite mapear las estructuras de una base de datos relacional (SQL Server, Oracle, MySQL, etc.) sobre una estructura lógica de entidades con el objeto de simplificar y acelerar el desarrollo de nuestras aplicaciones**.**

Eloquent hace corresponder cada tabla de la base de datos con una clase modelo de la aplicación, que nos permitirá interactuar con dicha tabla de una forma orientada a objetos y sin escribir código SQL.

La consecuencia más directa que se infiere del párrafo anterior es que, además de “mapear”, los ORMs tienden a “liberarnos” de la escritura o generación manual de código SQL (Structured Query Language) necesario para realizar las consultas y gestionar la persistencia de datos en la base de datos relacional. Esto surge con la idea de obtener un código portable con el que no tengamos la necesidad de usar un lenguaje SQL dentro del código PHP, en nuestro caso.

Eloquent, por tanto, implementa el patrón de arquitectura **Active Record**, que, **hace que cada tabla de la BD se corresponda con un Modelo (clase de PHP) que nos permita interactuar con dicha tabla de BD**. Los campos de cada tabla de la BD se corresponden con los atributos de la correspondiente clase Modelo. Cada instancia de dicha clase o modelo se corresponde con un registro en la tabla y genera persistencia en la base de datos, con lo cual una actualización de los atributos de la clase puede alterar el registro correspondiente en la base de datos.

Eloquent nos facilita la tarea de crear registros, actualizarlos, eliminarlos, etc de forma muy sencilla (CRUD o Create, Read, Update, Delete)

Un **modelo de Eloquent es una clase PHP que extiende de la clase Model** que proporciona funcionalidades para recibir y enviar la información a la base de datos. Los modelos se encuentran en la **carpeta App/Models**.

Usaremos las siguientes convenciones para que Laravel nos facilite el trabajo y nos ahorre líneas de código y tiempo. Para ello:

- Los **nombres de los modelos** se escribirán en **singular**, en contraposición con los nombres de las tablas de la BD que se escriben en plural.
- Los **nombres de los modelos** usan la notación **UpperCamelCase**.
- Uso del lenguaje inglés

Estas convenciones nos ayudan a detectar las tablas automáticamente. Por ejemplo, si tenemos una tabla en la base de datos con la que queremos trabajar que se llama **user_profiles**, vemos que se encuentra con las convenciones para tablas de bases de datos (plural y underscore), entonces el modelo para esta tabla cambiando las convenciones seria: **UserProfile** (singular y UpperCamelCase).

Si la tabla de base de datos necesitamos que tenga un nombre diferente a la clase modelo, dentro de la clase Modelo deberemos indicarlo con la siguiente línea:

```
protected $table = 'nombreTablaBD'
```

Para crear un modelo en Laravel utilizaremos el comando artisan:

```
php artisan make:model Modelo
```

Es muy importante respetar las convenciones para nombres.

También podemos crear el modelo y su migración a la vez:

```
php artisan make:model Modelo –m
```

OBTENER TODOS LOS DATOS DE LA TABLA

En el controlador obtendremos todos los registros de una tabla mediante Eloquent, es decir, empleando el correspondiente modelo y empleando el método all() o get() de eloquent:

```
$videogames = Videogame::get(); //array con los registros de la tabla videogames
```

OBTENER UN REGISTRO DE LA TABLA

Para obtener un determinado registro por su id utilizamos el método find() de Eloquent:

```
$game = Videogame::find($id)
```

Es posible, indicar como segundo argumento del método find() un arreglo con los campos de la tabla que se quieren obtener de cada registro, con esto optimizamos el método find de Eloquent:

```
$game = Videogame::find($id, ['id', 'name']);
```

El método `findOrFail($id)` devuelve un registro indicado por su id o una exception que reportará un error 404.

OTROS MÉTODOS DE ELOQUENT

Es posible crear consultas con el método where, por ejemplo:

```
$games = Videogame::where('category_id', 1)->orderBy('name')->get();
```

Otros métodos son:

```
$post = Post::whereTitle('Laravel Eloquent')->first();

$category = Category::whereName('Laravel Eloquent')->first();

$post = Post::orderBy('created_at', 'DESC')->get();

$post = Post::whereYear('created_at', 2023)->first();

$post = Post::whereMonth('created_at', 1)->first();

$post = Post::whereDay('created_at', 1)->first();

$post = Post::whereTime('created_at', '00:00:01')->first();

$post = Post::whereDate('created_at', '2023-01-01')->first();
```

Puedes consultar otros métodos en: https://laravel.com/docs/9.x/eloquent

10. MIGRACIONES

Las migraciones permiten crear la estructura de la base de datos a través de la programación en Laravel. Con las migraciones creamosy modificamos las tablas y sus columnas en la base de datos, a través de ficheros de migración. Por tanto, las migraciones permiten llevar un control sobre los cambios realizados en el esquema de la base de datos y permiten automatizar la instalación, facilitando el posterior despliegue. Cada clase modelo de la aplicación debería tener su correspondiente migración.

Los ficheros de migración se encuentran en la carpeta **database/migrations** y cada uno de ellos tiene un nombre, y una marca de tiempo que permite a Laravel determinar el orden de las migraciones. Laravel usará el nombre de la migración para determinar el nombre de la tabla de la base de datos, y si la migración creará una nueva tabla o una actualización de la misma.

Un fichero migración es una clase con dos métodos **up** y **down**. Con el método up agregaremos nuevas tablas o campos a la base de datos y con down podemos revertir las operaciones realizadas con el método up.

Documentación: https://laravel.com/docs/9.x/migrations (ver los tipos de columna que podemos usar)

En primer lugar, crearemos la migración con el siguiente comando ejecutado en la terminal del S.O:

```
php artisan make:migration create_nombreTabla_table
```

En segundo lugar, podemos ver que Laravel ha creado un fichero de clase con el nombre de la migración con los métodos up y down. Modificaremos el fichero up añadiendo los campos de la tabla de BD.

En tercer lugar, ejecutaremos la migración con el comando:

```
php artisan migrate
```

NOTA: para que laravel haga tareas de forma automática como crear las tablas debemos seguir las convenciones en cuento a nombres de Laravel.

Nuestro proyecto, trae migraciones creadas por defecto para la tabla users de la aplicación. En la función up() vemos que se va a crear la tabla users con las columnas id (autoincremental y único), name (string), email (string y único), email_verified_at (timestamp con fecha y hora y puede ser null), password, token (tokens de

autenticación), y timestamps (dos columnas de tipo timestamps, created_at para indicar cuándo se creó el registro y updated_at para indicar cuándo fue modificado por última vez).

```php
public function up()
{
    Schema::create('users', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('email')->unique();
        $table->timestamp('email_verified_at')->nullable();
        $table->string('password');
        $table->rememberToken();
        $table->timestamps();
    });
}
```

Como ejemplo, vamos a crear una **nueva migración** utilizando el comando siguiente dentro de nuestro proyecto. En el ejemplo vamos a crear una migración para la tabla categorías de nuestra aplicación para juegos:

```
c:\wamp64\www\probando-laravel>php artisan make:migration create_categories_table

  INFO  Migration [2022_12_28_072943_create_categories_table] created successfully.
```

En la carpeta **database/migrations** se habrá creado un fichero de nombre **create_categories_table** con las funciones up y down:

```php
public function up()
{
    Schema::create('categories', function (Blueprint $table) {
        $table->id();
        $table->timestamps();
    });
}
```

Modificamos la función up para añadir el 'nombre' y 'descripción' de la categoría.

```php
public function up()
{
    Schema::create('categories', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('description');
        $table->timestamps();
    });
}
```

Una vez creada la migración debemos ejecutarla:

```
c:\wamp64\www\probando-laravel>php artisan migrate

  INFO  Preparing database.

Creating migration table ..................................................... 46ms DONE

  INFO  Running migrations.

2014_10_12_000000_create_users_table ......................................... 66ms DONE
2014_10_12_100000_create_password_resets_table ............................... 54ms DONE
2019_08_19_000000_create_failed_jobs_table ................................... 56ms DONE
2019_12_14_000001_create_personal_access_tokens_table ........................ 88ms DONE
2022_12_28_072943_create_categories_table .................................... 39ms DONE
```

Como resultado, se ejecutan todas las migraciones que tengamos, una por cada fichero de la carpeta migrations. Si nos dirigimos a nuestro SGBD podremos observar que se han creado las tablas indicadas en los ficheros de migraciones.

NOTA:

Usaremos text('name') para crear un campo de tipo TEXT cuyo tamaño máximo establecido no editable es de 65535 caracteres. Un campo de tipo TEXT no puede formar parte de un índice.

Usaremos string('name') para crear un campo de tipo VARCHAR cuyo tamaño máximo es editable entre 1 y 65535 caracteres. Un campo de tipo VARCHAR puede forma parte de un índice.

MODIFICAR TABLAS DE BASE DE DATOS DESDE MIGRACIONES

Es posible modificar la estructura de una tabla de la base de datos para añadir/eliminar campos sin pérdida de datos, es decir, aun cuando tengamos registros en la tabla, para ello creamos una nueva migración con el comando:

```
php artisan make:migration  add_campo_nombreTabla_table  --table = nombreTabla
```

Como ejemplo, vamos a añadir el campo active a la tabla categories:

```
php artisan make:migration add_active_column_to_categories_table --table=categories
```

En el fichero de migración creado añadimos el campo, quedando:

```php
public function up()
{
    Schema::table('categories', function (Blueprint $table) {
        //
        $table->boolean('active')->default(true);
    });
}
```

Finalmente ejecutamos la migración con el comando: `php artisan migrate`

Y comprobamos que se ha añadido el campo indicado en la tabla en la base de datos.

Para usar el ORM debemos **crear los modelos de datos** de la aplicación. Para crear un modelo **eloquent** de datos utilizamos el comando:

```
php artisan make:model nombreModelo
```

Cada clase modelo que creemos extiende o hereda de la clase Illuminate\Database\Eloquent\Model. Es posible crear un modelo y su correspondiente migración con el comando:

```
php artisan make:model nombreModelo --migration
```

Otros comandos interesantes: https://laravel.com/docs/9.x/eloquent

Tinker

Laravel proporciona una herramienta para trabajar con los modelos llamada **Tinker**. Tinker, es una shell interactiva que viene incluida como dependencia en la instalación de laravel, que nos permite interactuar con toda la aplicación a través de la línea de comandos. Con Tinker podemos trabajar con objetos PHP e interactuar con la BD. Para **ingresar en tinker** ejecutamos el comando:

```
c:\wamp64\www\probando-laravel>php artisan tinker
Psy Shell v0.11.10 (PHP 8.0.13 — cli) by Justin Hileman
> _
```

En el siguiente ejemplo, indicamos a Tinker el modelo de datos que vamos a instanciar, creamos un objeto, modificamos sus atributos (que se corresponden con los campos de la tabla con la que está asociada) y guardamos los datos.

```
c:\wamp64\www\probando-laravel>php artisan tinker
Psy Shell v0.11.10 (PHP 8.0.13 — cli) by Justin Hileman
> use App\Models\Category;
> $category = new Category();
= App\Models\Category {#3686}

> $category->name = "Deportes";
= "Deportes"

> $category->description = "Categoria de deportes";
= "Categoria de deportes"

> $category->active = true;
= true

> $category->save();
= true

>
```

name, description y active son los campos de la tabla categories de la BD

Tras ejecutar el método sabe() se guarda un nuevo registro en la tabla categories.

A través del objeto $category podemos modificar los datos de un registro. Si hacemos $category->name = “Acción” se modifica el campo name del registro asociado al objeto $category. También podemos buscar registros (por id), eliminarlos, etc:

```
> $category3 = Category::find(2);
= App\Models\Category {#4642
    id: 2,
    name: "Acción",
    description: "Categoria de Acción",
    active: 1,
    created_at: "2022-12-28 11:52:43",
    updated_at: "2022-12-28 11:52:59",
  }

> $category3->delete();
= true
```

11. SEEDERS

Los Seeders permiten inicializar las tablas de la BD para configurar el estado inicial de las tablas para un proyecto y/o crear datos de prueba. De esta forma podemos crear datos de prueba para trabajar durante el desarrollo del proyecto o configurar el estado inicial de las tablas de la BD. Las Seeders se almacenan dentro de la carpeta **database/seeders**.

Para crear un Seeder utilizamos el comando:

```
php artisan make:seeder nombreSeeder
```

Como ejemplo, vamos a crear la seeder para la tabla categories (vaciamos la tabla antes de nada).

```
c:\wamp64\www\probando-laravel>php artisan make:seeder CategoriesTableSeeder

  INFO  Seeder [C:\wamp64\www\probando-laravel\database/seeders/CategoriesTableSeeder.php] created successfully.
```

El siguiente paso es completar el fichero de Seeder que acaba de crearse en database/seeders con las categorías que queramos crear. No se te olvide incluir el fichero del modelo a utilizar.

```php
use Illuminate\Database\Seeder;
use App\Models\Category;

class CategoriesTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        $category1 = new Category; //usa el modelo Category de Models
        $category1->name = "Deportes";
        $category1->description = "Categoría basada en deportes como fútbol, baloncesto, tenis
        $category1->active= true;
        $category1->save();

        $category2 = new Category;
        $category2->name = "Acción";
        $category2->description = "Categoría basada en juegos de acción";
        $category2->active= true;
        $category2->save();
```

Añade el modelo

En el fichero **DatabaseSeeder.php** añadiremos una llamada a nuestra Seeder dentro del método run:

```php
$this->call([CategoriesTableSeeder::class, ...]);
```

Si tenemos varias Seeders, las añadimos separadas por comas

```php
namespace Database\Seeders;

// use Illuminate\Database\Console\Seeds\WithoutModelEvents;
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        $this->call([CategoriesTableSeeder::class]);
    }
}
```

El siguiente paso es correr una migración completa, borrando todo y creando de nuevo las tablas y ejecutando las seeders para añadir los datos:

```
php artisan migrate:fresh  --seed
```

Recargamos la BD y comprobamos que la tabla categories tiene los datos indicados en la seeder.

12. FACTORIES

Las factorías nos permiten rellenar la BD con datos de ejemplo de forma masiva (datos aleatorios no reales).

Las factorías se crean con el comando:

```
php artisan make:factory nombreFactory --model=nombreModel
```

Para utilizar las factorías nos vamos a ayudar de otra herramienta incorporada en Laravel 9, llamada **Faker**, que permite generar datos aleatorios falsos.

Como ejemplo, crearemos una factoría para la tabla videogames:

```
c:\wamp64\www\probando-laravel>php artisan make:factory VideogameFactory --model=Videogame

  INFO  Factory [C:\wamp64\www\probando-laravel\database/factories/VideogameFactory.php] created successfully.
```

Las factorías se encuentran en la carpeta **database/factories**, cada una dentro de un fichero.

```php
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

/**
 * @extends \Illuminate\Database\Eloquent\Factories\Factory<\App\Models\Videogame>
 */
class VideogameFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array<string, mixed>
     */
    public function definition()
    {
        return [
            'name' => $this->faker->name(),
            'category_id' => $this->faker->randomElement([1,2,3])
        ];
    }
}
```

name() genera nombres de persona

Tenemos en BD las categorías 1, 2, 3

A continuación para correr las factorías que creemos debemos añadir su invocación dentro del fichero DatabaseSeeder.php que se encuentra dentro de la carpeta database/seeders:

```php
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        //Invocacion de las Seeders
        $this->call([CategoriesTableSeeder::class , VideogamesTableSeeder::class]);

        //Invocacion de las factorias
        \App\Models\Videogame::factory(100)->create();

        // \App\Models\User::factory(10)->create();

        // \App\Models\User::factory()->create([
        //     'name' => 'Test User',
        //     'email' => 'test@example.com',
        // ]);
    }
}
```

Crear 100 registros videogames

Finalmente, ejecutamos las migraciones:

```
php artisan migrate:fresh --seed
```

13. FORMULARIOS EN BLADE E INSERTAR DATOS EN LA BD

El motor de plantillas Blade nos permite trabajar con formularios. En los formularios de Laravel siempre hay que enviar un token **(@csrf)** para evitar problemas de seguridad y ataques CSRF (Cross Site Request Forgery o Falsificación de Petición en Sitios Cruzados). Si no lo utilizamos se rechazará la invocación del formulario.

Consultar: https://aprendible.com/series/laravel-tips/lecciones/que-significa-el-error-tokenmismatchexception-y-de-que-nos-protege

El token @csrf crea un campo oculto en el formulario con un token que Laravel utilizará para verificar la procedendecia de los datos del formulario. Por defecto, el tiempo de vida de este token es de 2 horas.

En el campo **action** del formulario añadiremos la ruta que indica qué acción ejecutar al enviar los datos del formulario, por tanto, la ruta indicada en action deberá ser una ruta de **tipo POST** que crearemos en el fichero de rutas (web.php).

Los datos del formulario serán enviados dentro del objeto Request (petición). Por tanto el método indicado en en la ruta post debe recibir como parámetro un objeto de tipo Request. Observa que artisan incluye en los controllers la clase Request.

Se recomienda que los campos de los formularios tengan el mismo nombre que los campos de las tablas de BD. Veamos cómo trabajar con formularios con ejemplos.

VALIDAR FORMULARIOS

Para validar los campos que vienen de un formulario añadiremos código de validación en las funciones del controlador. Por ejemplo:

```
public function nombrefuncion(Request $request){
    //Los datos procendentes del formulario se reciben en $request
    $request-> validate([
      'nombreCampo1' => 'required | min:5 | max:255 | unique:nombreTabla',
      'nombreCampo2' => 'required | integer | exists:nombreTabla,campoTabla'
      ]);
      $game = new Videogame; //Instanciamos el ORM
      $game->name = $request->name;
      $game->category_id = $request->category_id;
      $game->active = 1;
      $game->save();

      return redirect()->route('games');
 }
```

'**nombreCampo'** es el nombre de los campos del formulario de la vista

En la documentación de Laravel puedes consultar todas las validaciones posibles que existen (https://laravel.com/docs/9.x/validationhttps://laravel.com/docs/9.x/validation). El método **validate** devuelve un **mensaje de error** a la vista cuando falla la validación sobre el campo indicado. Para mostrar el mensaje de validación, añadimos a la vista en cuestión el siguiente código:

```
@error ('nombreCampo1')

    {{ $message }}
@enderror
```

'nombreCampo1' es el nombre del campo validado

Evitamos volver a rellenar todos los campos de un formulario cuando alguno de ellos no pasa la validación añadiendo:

```
<input type="text" name="nombreCampo" value="{{old('nombreCampo')}}">
```

14. MENSAJES DE SESION EN LARAVEL

Para almacenar datos en la sesión y que estén disponibles para la siguiente solicitud podemos utilizar el método flash. De esta manera podemos crear mensajes de sesión:

```
session()->flash('message', 'Videogame created successfully');
```

Para mostrar la variable de sesión en la vista:

```
    @if(session('message'))
        <div>{{ session('message') }} </div>
    @endif
```

https://laravel.com/docs/9.x/session#interacting-with-the-session

15. TRADUCCIÓN DE MENSAJES DE LARAVEL

Para traducir los nombres de los mensajes que muestra Laravel, modificaremos la directiva 'locale' del fichero de configuración config/app.php.

```
  'locale' => 'es',
```

Esta directiva apunta a una carpeta (en, es, etc) dentro de la carpeta lang del proyecto. Debemos crear por tanto la carpeta es, en cuyo interior crearemos el fichero **validation.php** que tendrá el siguiente contenido para los mensajes de validación:

```
return [

    'required' => 'El campo :attribute es obligatorio',
    'attributes' => [
                      'name' => 'nombre',
                      'title' => 'titulo',
                    ...
                    ],

];
```

Una opción mejor es descargar de github los ficheros traducidos, para ello:

- Accedemos a la url github.com/Laraveles/spanish
- Entramos en la carpeta resources/lang
- Entramos en la carpeta es
- Copiamos los 4 ficheros en la carpeta lang/es de nuestro proyecto local

16. RUTAS RESOURCES

Anteriormente, hemos creado las rutas para hacer el CRUD de videojuegos. Solamente para un CRUD hemos necesitado 7 rutas diferentes. En aplicaciones mayores con varios CRUD, el número de rutas se multiplica exponencialmente.

El comando siguiente nos muestra la lista de rutas con las que estamos trabajando:

```
php artisan r : list
```

Las rutas resources ofrecen una manera limpia y automatizada para la creación de las rutas para implementar un CRUD. Se crean con el siguiente comando, el cual creará un controlador con todas las funciones necesarias para implementar un CRUD, empleando la nomenclatura de Laravel:

```
php artisan make:controller  nombreControlador –r
```

En el fichero de rutas, web.php, añadiremos la siguiente línea, indicando que queremos crear una ruta de tipo resource (una sola línea en lugar de 7):

```
Route::resource('games', nameController::class);
```

Donde games sería el nombre de la URL. Si volvemos a consultar las rutas veremos que Laravel ha generado las rutas necesarias para este CRUD.

El nombre de cada ruta será por ejemplo:

- games.index
- games.store
- games. create
- games.show
- games.update
- games.destroy

No olvidar incluir en el fichero de rutas el nombre de cada controlador a utilizar.

17. PROTEGER RUTAS DE USUARIOS NO AUTENTICADOS

Es posible restringir el acceso a determinadas rutas, dependiendo de si el usuario se ha autenticado o no, utilizando middleware.

Un método para proteger rutas consiste en determinar el acceso a las mismas en el fichero de rutas. Por ejemplo:

```
Route::get('/games/create', [GameController::class, 'create'])->name('create')->middleware('auth');
```

En este caso solamente podremos acceder a la ruta create si estamos autenticados, en caso contrario, la aplicación nos redirigirá a la ruta login.

Si consultamos la clase Kernel.php ubicada en App\Http, la variable middleware 'auth' nos lleva a la clase App\Http\Middleware\Authenticate donde podemos ver o modificar la ruta a la que se nos redirigirá en caso de no estar autenticados. Por tanto, nuestro proyecto deberá contar con la ruta login.

Otra forma de proteger las rutas de nuestra aplicación es hacerlo desde el controlador relacionado con la ruta mediante un constructor en el que añadimos una de las siguientes líneas:

```
$this->middleware('auth'); //protege todas las rutas del controlador
```

```
$this->middleware('auth' , [ 'except' => ['index']]);//Protege todas excepto la del método index
```

```
$this->middleware('auth' , [ 'only' => ['create','store']]);//protege solamente las rutas create y store
```

18. LARAVEL BREEZE

Laravel Breeze es un paquete que nos proporciona todo lo necesario para tener funcionando **un sistema de autenticación ya implementado** en nuestro proyecto. Esto incluye el login y registro, pero también otras necesidades básicas de las aplicaciones como recuperar claves olvidadas, la verificación del email, etc. Laravel Breeze está enfocado para trabajar con Blade. Al instalar Breeze se instalaran las plantillas de Blade, Tailwindcss, etc. Para trabajar con Breeze se recomienda consultar el manual de Laravel y seguir los pasos para su instalación:

Para instalar Breeze sobre un proyecto nuevo hacemos:

```
composer require laravel/breeze  --dev
```

```
php artisan breeze:install
```

Antes de ejecutar el comando migrate deberemos configurar el proyecto para usar la base de datos con la que se vaya a trabajar. A continuación ejecutamos:

```
php artisan migrate
```

Los dos siguientes comando se recomienda que los ejectutes en otra terminal dentro del proyecto con el que estamos trabajando:

```
npm install
```

```
npm run dev
```

El comando npm run dev lo dejaremos corriendo para que las vistas que vayamos modificando o diseñando se recarguen automáticamente.

Si probamos nuestro proyecto veremos que ya tenemos funcionando la autenticación. Probar el registro y autenticación. Ya podemos integrar esta funcionalidad en cualquier aplicación.

FLOWBITE

Flowbite es una biblioteca de código abierto de componentes de Tailwind Css (https://flowbite.com)

En la página web de Flowbite, en Docs / Laravel nos indica cómo instalarlo:

```
npm install –D tailwindcss postcss autoprefixer flowbite
```

En el fichero **tailwind.config.js** de nuestro proyecto añadimos:

```
JS tailwind.config.js > ...
1   const defaultTheme = require('tailwindcss/defaultTheme');
2
3   /** @type {import('tailwindcss').Config} */
4   module.exports = {
5       content: [
6           './vendor/laravel/framework/src/Illuminate/Pagination/resources/views/*.blade.ph
7           './storage/framework/views/*.php',
8           './resources/views/**/*.blade.php',
9           "./node_modules/flowbite/**/*.js",
10      ],
11
12      theme: {
13          extend: {
14              fontFamily: {
15                  sans: ['Nunito', ...defaultTheme.fontFamily.sans],
16              },
17          },
18      },
19
20      plugins: [require('@tailwindcss/forms'),
21      require('flowbite/plugin')
22      ],
23
```

Como Breeze ya utiliza Tailwind no es necesario hacer nada más, pero sí hemos de añadir en la plantilla que utilicemos, o en nuestra vista:

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

Y el CDN de Flowbite:

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/flowbite/1.6.3/flowbite.min.js"></script>
```

19. ROLES Y PERMISOS

Para trabajar con roles utilizamos el paquete Spatie de Laravel. En la documentación se indica cómo instalar este paquete. Los siguientes comandos se ejecutan sobre un proyecto laravel recién creado:

```
composer require spatie/laravel-permission
```

```
php artisan vendor:publish --provider="Spatie\Permission\PermissionServiceProvider"
```

```
php artisan optimize:clear
```

```
php artisan migrate
```

Editamos el modelo **User**.php y añadimos las líneas en rojo:

```
use Illuminate\Foundation\Auth\User as Authenticatable;
use Spatie\Permission\Traits\HasRoles;

class User extends Authenticatable
```

```
{
    use HasRoles;

    // ...
}
```

Los roles se crearán mediante una **migración**, en cuyo método up() crearemos roles con la función create del modelo Role:

```
$unRol = Role::create([‘name’ => ‘nombre_rol’]);
```

Asociamos este rol a un usuario de la BD:

```
$unUsuario = User::find(id);
$unUsuario->assignRole($unRol);
```

En el fichero de migración añadiremos las clases:

```
use Spatie\Permission\Models\Permission;
use App\Models\User;
use Spatie\Permission\Models\Role;
```

En las vistas hacemos uso de los roles creados mediante la directiva de Blade:

```
@role(‘nombre_rol’)
…
@endrole
```

También es posible crear roles nuevos con el comando:

```
php artisan permission:create-role nombre_rol
```

Con el comando siguiente podemos consultar los roles de que disponemos:

```
php artisan permission:show
```

20. SUBIR FICHEROS AL SERVIDOR

Para subir ficheros al servidor utilizamos el Facade Storage:

```
use Illuminate\Support\Facades\Storage;
…
if($request->hasFile(‘image’){
    …
    Storage::putFile(‘public’,$request->file(‘image’));
}
```

Para eliminar el fichero del servidor:

```
Storage::delete($url);
```

Los ficheros se almacenan en la carpeta /storage de nuestro proyecto. Para crear enlaces simbolicos a los ficheros subidos en la carpeta /public ejecutamos el comando:

```
php artisan storage:link
```

21. BREEZE

Laravel Breeze es un paquete que proporciona la funcionalidad e interfaz necesaria para proporcionar un sistema de autenticación a una aplicación desarrollada con Laravel. Se intala con los siguientes pasos:

```
composer require laravel/breeze –dev
```

```
php artisan breeze:install
```

```
php artisan migrate
```

```
npm install
```

```
npm run dev
```