

Vamos a desarrollar una pequeña web para la gestión de un museo de obras de arte de tipo pictórico.

Empezaremos por definir las rutas y vistas del sitio y poco a poco en los siguientes ejercicios la iremos completando hasta terminar el sitio web completo.

El objetivo es realizar un sitio web para la gestión del museo, el cual estará protegido mediante usuario y contraseña. Una vez autorizado el acceso, el usuario podrá listar las pinturas, ver información detallada de una de ellas, realizar búsquedas o filtrados y algunas operaciones más de gestión.

### **Instalación de Laravel**

En primer lugar, tenemos que instalar todo lo necesario para poder realizar el sitio web con Laravel. Para esto seguiremos las explicaciones del apartado "Laravel. Instalación" que hemos visto en la teoría.

Una vez instalado crearemos un nuevo proyecto de Laravel llamado **laravel\_museo** y probamos que todo funcione correctamente.

## Definición de las rutas

En este ejercicio vamos a definir las rutas principales que va a tener nuestro sitio web. Para empezar, simplemente indicaremos que las rutas devuelvan una cadena (así podremos comprobar que se han creado correctamente). A continuación, se incluye una tabla con las rutas a definir (todas de tipo GET) y el texto que tienen que mostrar:

Ruta	Texto a mostrar
/	Pantalla principal
<b>pinturas</b>	Listado de pinturas
<b>pinturas /{pintura}</b>	Vista en detalle del pintura {pintura}
<b>pinturas /crear</b>	Añadir una pintura
<b>pinturas /{pintura}/editar</b>	Modificar una pintura

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

## Configuración

En el archivo .env modifica el valor de **APP\_LOCALE** y de **APP\_FALLBACK\_LOCALE** a “es”

Seguimos desarrollando la web para la gestión de un museo. Primero añadiremos los controladores y métodos asociados a cada ruta, y posteriormente también haremos las vistas usando el sistema de plantillas Blade.

## Controladores

Vamos a crear los controladores necesarios para gestionar nuestra aplicación y además actualizaremos el fichero de rutas para que los utilice.

Empezamos por añadir dos controladores que nos van a hacer falta:

- InicioController.php
- PinturaController.php

Para crear **InicioController** hay que utilizar el comando de Artisan que permite crear un controlador vacío (sin métodos).

Para crear **PinturaController** utilizaremos el parámetro para crear un controlador de recursos.

A continuación, vamos a añadir los métodos de estos controladores. En la siguiente tabla resumen podemos ver un listado de los métodos por controlador y las rutas que tendrán asociadas:

Ruta	Name de la ruta	Controlador	Método
/	inicio	InicioController	inicio
<b>pinturas</b>	pinturas.index	PinturaController	index
<b>pinturas/{pintura}</b>	pinturas.show	PinturaController	show
<b>pinturas/crear</b>	pinturas.create	PinturaController	create
<b>pinturas/{pintura}/editar</b>	pinturas.edit	PinturaController	edit

Por último, vamos a cambiar el fichero de rutas routes/web.php para que todas las rutas que teníamos definidas apunten a los nuevos métodos de los controladores, por ejemplo:

```
Route::get("/", [InicioController::class, "inicio"]);
```

El texto que teníamos puesto para cada ruta hay que moverlo al método del controlador correspondiente.

## Vistas

Crearemos las siguientes vistas. Para organizar mejor las vistas las vamos a agrupar en sub-carpetas dentro de la carpeta resources/views siguiendo la siguiente estructura:

Vista	Carpeta	Ruta asociada
<b>home.blade.php</b>	resources/views/	/
<b>index.blade.php</b>	resources/views/pinturas/	pinturas
<b>create.blade.php</b>	resources/views/pinturas/	pinturas/crear
<b>show.blade.php</b>	resources/views/pinturas/	pinturas/{pintura}
<b>edit.blade.php</b>	resources/views/pinturas/	pinturas/{pintura}/editar

Creemos una vista separada para todas las rutas, moviendo el texto de los controladores a ellas.

Por último, vamos a actualizar los métodos de los controladores para que se carguen las vistas que acabamos de crear. Acordaros que para referenciar las vistas que están dentro de carpetas la barra / de separación se transforma en un punto y que, además, como segundo parámetro, podemos pasar datos a la vista. A continuación, se incluyen algunos ejemplos:

```
return view('home');  
return view('pinturas.index');  
return view('pinturas.show', ['pintura'=>$pintura]);
```

Una vez hechos estos cambios ya podemos probarlo en el navegador, el cual debería mostrar el texto que hemos puesto de ejemplo.

## Plantillas para las vistas

### 1.- Layout principal de las vistas con Bootstrap

En primer lugar, vamos a crear el layout base que van a utilizar el resto de vistas del sitio web y además incluiremos la librería Bootstrap para utilizarla como estilo base.

En primer lugar, accedemos a la web <http://getbootstrap.com> y descargamos la librería (*Compiled CSS and JS*). Esto nos bajará un fichero zip comprimido con dos carpetas (js y css) que tenemos que extraer en la carpeta *public/assets/bootstrap* (tendremos que crear las carpetas */assets/bootstrap*).

También nos tenemos que descargar la plantilla para la barra de navegación principal (navbar.blade.php) proporcionada por el profesor y la almacenamos en la carpeta *resources/views/layouts*.

A continuación, vamos a crear el layout principal de nuestro sitio:

- Borraremos la vista *welcome.blade.php* ya que no la necesitaremos.
- Creamos el fichero *resources/views/layouts/master.blade.php*.
- Le añadimos como contenido el siguiente:

```
<!doctype html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Plantilla Blade</title>
  <link href="{{ url('/assets/bootstrap/css/bootstrap.min.css') }}" rel="stylesheet" >
</head>
<body>
  <h1>Empezando con Laravel</h1>
  <script src="{{ url('/assets/bootstrap/js/bootstrap.min.js') }}"></script>
</body>
</html>
```

- Añade otra hoja de estilos en */assets/css* llamada *estilo.css* en la que añadas un *padding-top: 4.5rem* a *body*. **Después añade *estilo.css* en el *master.blade.php***
- Dentro de la sección *<body>* del HTML, eliminamos el texto que viene de ejemplo (*<h1>Empezando con Laravel</h1>*) e incluimos la barra de navegación que hemos guardado antes utilizando el siguiente código:
  - ***@include('layouts.navbar')***
- Modifica el *title* añadiendo dentro ***@yield('titulo')***
- A continuación de la barra de navegación añadimos la sección principal donde aparecerá el contenido de la web:

```
<div class="container-lg">
  @yield('contenido')
</div>
```

Con esto ya hemos definido el layout principal. Sin embargo, todavía no podemos probarlo ya que no está asociado a ninguna ruta.

## 2.- Modificar el resto de vistas

Ahora vamos a terminar una primera versión estable de la web. Modificaremos las vistas asociadas a cada ruta para que extiendan del layout que hemos hecho y mostrar (en la sección de contenido del layout) el texto de ejemplo que habíamos definido en ellas.

En general, todas las vistas tendrán un código similar al siguiente (variando únicamente la sección contenido):

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

```
@section('titulo')
```

*Museo*

```
@endsection
```

```
@section('contenido')
```

*Pantalla principal*

```
@endsection
```

Una vez hechos estos cambios ya podemos probarlo en el navegador, el cual debería mostrar en todos los casos la plantilla base con la barra de navegación principal y los estilos de Bootstrap aplicados.

Seguimos desarrollando la web para la gestión de un museo. Ahora completaremos las vistas usando formularios y el sistema de plantillas Blade

### Método InicioController@inicio

En este método de momento solo vamos a hacer una redirección a la acción que muestra el listado de todas las pinturas del museo:

```
//return redirect()->action([PinturaController::class,'index']);  
return redirect()->route('pinturas.index');
```

### Método PinturaController@index

Este método tiene que mostrar un listado de todas las pinturas que tenemos en el museo.

El listado de pinturas lo podéis obtener del fichero *array\_pinturas.php* facilitado por el profesor.

Este array hay que copiarlo como un atributo de la clase PinturaController (más adelante los almacenaremos en la base de datos).

En el método del controlador simplemente tendremos que modificar la generación de la vista para pasarle este array de pinturas completo (`$this->pinturas`).

Y en la vista correspondiente simplemente tendremos que incluir el siguiente trozo de código en su sección content:

```
<div class="row">  
@foreach( $pinturas as $clave => $pintura )  
    <div class="col-sm-6 col-md-4 col-lg-4 col-xl-3 mb-4">  
        <a href="{{ route('pinturas.show', $clave ) }}">  
              
            <h2>  
                {{ $pintura['titulo'] }}  
            </h2>  
        </a>  
    </div>  
@endforeach  
</div>
```

Este código se muestra a modo de ejemplo. Debéis de **modificarlo** por uno vuestro en el que añadáis un estilo propio.

Como se puede ver en el código, en primer lugar, se crea una fila (usando el sistema de rejilla de Bootstrap) y a continuación se realiza un bucle foreach utilizando la notación de Blade para iterar por todos los pinturas.

### Método `PinturaController@show`

Este método se utiliza para mostrar la vista detalle de una pintura.

Hemos de tener en cuenta que el método correspondiente recibe una pintura que (de momento) se refiere a la posición de la pintura en el array.

Por lo tanto, tendremos que coger dicha pintura del array (`$this->pinturas[$pintura]`) y pasárselo a la vista.

En esta vista vamos a crear dos columnas, la primera columna para mostrar la imagen del pintura y la segunda para incluir todos los detalles. A continuación, se incluye un ejemplo la estructura HTML que debería tener esta pantalla:

```
<div class="row">
  <div class="col-md-6">
    {{-- TODO: Imagen de la pintura --}}
  </div>
  <div class="col-md-6">
    {{-- TODO: Datos de la pintura --}}
  </div>
</div>
```

En la columna de la izquierda completamos el TODO para insertar la imagen de la pintura.

En la columna de la derecha se tendrán que mostrar todos los datos de la pintura.

Además, tenemos que incluir dos botones: un botón que nos llevará a editar la pintura y otro para volver al listado de pinturas.

Nota: los botones de momento no tienen que funcionar. En Bootstrap podemos transformar un enlace en un botón, simplemente aplicando la clase "btn" (más info en: <https://getbootstrap.com/docs/5.3/components/buttons/>).

Esta pantalla finalmente debería tener una apariencia similar a la siguiente:

[Museo](#) [Listado de pinturas](#) [Nueva pintura](#) [Login](#)



## El 3 de mayo en Madrid

Artista: Francisco de Goya

Año: 1814

Periodo: Romanticismo

Una representación del brutal fusilamiento de los rebeldes españoles por las tropas napoleónicas francesas.

Goya captura el horror y la desesperación en las expresiones de los prisioneros, destacando al hombre con la camisa blanca en el centro.

Este cuadro es uno de los grandes manifiestos antibélicos y sigue siendo una referencia en la historia del arte. La crudeza y el realismo de la escena contrastan con el lirismo de otros cuadros de la misma época.

[Editar](#) [Volver a la galería](#)



**PinturaController@create**

Este método devuelve la vista "pinturas.create" para añadir una nueva pintura. Para crear este formulario en la vista correspondiente nos podemos basar en el contenido siguiente. Hay una serie de TODOs que hay que completar.

```
<div class="row">
  <div class="offset-md-3 col-md-6">
    <div class="card">
      <div class="card-header text-center">
        Añadir pintura
      </div>
      <div class="card-body" style="padding:30px">

        {{-- TODO: Abrir el formulario e indicar el método POST --}}
        {{-- TODO: Protección contra CSRF --}}
        <div class="mb-3">
          <label for="titulo">Título</label>
          <input type="text" name="titulo" id="titulo" class="form-control" required>
        </div>

        <div class="mb-3">
          {{-- TODO: Completa el input para la fecha--}}
        </div>

        <div class="mb-3">
          {{-- TODO: Completa el input para el periodo --}}
        </div>

        <div class="mb-3">
          <label for="descripcion">Descripción</label>
          <textarea name="descripcion" id="descripcion" class="form-control"
rows="3"></textarea>
        </div>

        <div class="mb-3">
          {{-- TODO: Completa el input para la imagen --}}
        </div>

        <div class="mb-3 text-center">
          <button type="submit" class="btn btn-success" style="padding:8px 100px;margin-
top:25px;">
            Añadir pintura
          </button>
        </div>
        {{-- TODO: Cerrar formulario --}}
      </div>
    </div>
  </div>
</div>
```

Además, tendrá un botón al final con el texto "Añadir pintura".  
De momento el formulario no funcionará. Más adelante lo terminaremos.

### **Método PinturaController@edit**

Este método permitirá modificar el contenido de un pintura. El formulario de la vista "pinturas.edit" será exactamente igual al de añadir pintura, así que lo podemos copiar y pegar en esta vista y simplemente cambiar el texto del botón de envío por "Modificar pintura".

De momento no tendremos que hacer nada más, más adelante lo completaremos para que se rellene con los datos del pintura a editar.

Seguimos desarrollando la web para la gestión de un museo. Le añadiremos todo lo referente a la gestión de bases de datos.

## 1.- Configuración de la base de datos y migraciones

1. En primer lugar, vamos a configurar correctamente la base de datos. Para esto tenemos que actualizar el fichero **.env** para indicar que vamos a usar una base de datos tipo MySQL llamada "dwes\_laravel\_museo" junto con el nombre de usuario y contraseña de acceso.
2. Luego, creamos la base de datos con codificación utf8mb4\_general\_ci (si no la creamos, artisan la creará por nosotros)
3. Para comprobar que todo se ha configurado correctamente vamos a un terminal en la carpeta de nuestro proyecto y ejecutamos el comando que crea la tabla de migraciones.  
Si todo va bien podremos actualizar desde DBeaver o PhpMyAdmin y comprobar que se ha creado esta tabla dentro de nuestra nueva base de datos.
4. Ahora vamos a crear la tabla que utilizaremos para almacenar las pinturas. Ejecuta el comando de Artisan para crear la migración llamada **create\_pinturas\_table** para la tabla pinturas. Una vez creado edita este fichero para añadir todos los campos necesarios, estos son:

Campo	Tipo
<b>id</b>	Autoincremental
<b>titulo</b>	String (único)
<b>slug</b>	String (único)
<b>artista</b>	String
<b>fecha</b>	Tipo entero (sólo almacenaremos el año)
<b>imagen</b>	String (puede ser null)
<b>periodo</b>	String de 30 caracteres (puede ser null)
<b>descripcion</b>	Texto largo (puede ser null)
<b>timestamps</b>	Timestamps de Eloquent

5. Por último, ejecutaremos el comando de Artisan que añade las nuevas migraciones y comprobaremos en DBeaver o PhpMyAdmin que la tabla se ha creado correctamente con los campos que le hemos indicado.

Seguimos desarrollando la web para la gestión de un museo y configurando la base de datos.

### 1.- Modelo de datos

En este ejercicio vamos a crear el modelo de datos asociado con la tabla pinturas. Para esto usaremos el comando apropiado de Artisan para crear el modelo llamado Pintura.

Una vez creado este fichero lo abriremos y comprobaremos que el nombre de la clase sea el correcto y que herede de la clase Model.

También debemos comprobar si el nombre de la clase en plural coincide con el nombre de la tabla de la base de datos.

Y ya está. No es necesario hacer nada más. El cuerpo de la clase puede estar vacío ({}), ¡todo lo demás se hace automáticamente!

### 2.- Creación de semillas para la base de datos

Sirven para rellenar tablas de la base de datos con datos iniciales.

Para esto seguiremos los siguientes pasos:

- Crearemos una clase seeder llamada PinturaSeeder
- Desde el método run de DatabaseSeeder.php lo llamaremos de la forma:

```
DB::table('pinturas')->delete();  
$this->call(PinturaSeeder::class);
```

- Movemos el array de pinturas que se facilitaba en los materiales y que habíamos copiado dentro del controlador PinturaController a la clase de semillas (PinturaSeeder.php), guardándolo de la misma forma, como atributo privado de la clase.
- Dentro del método run() de PinturaSeeder realizamos las siguientes acciones:  
Hay que importar: `use Illuminate\Support\Str;`

```
public function run()  
{  
    foreach ($this->pinturas as $pintura)  
    {  
        $p = new Pintura();  
        $p->titulo = $pintura['titulo'];  
        $p->slug = Str::slug($pintura['titulo']);  
        $p->artista = $pintura['artista'];  
        $p->fecha = $pintura['fecha'];  
        $p->imagen = $pintura['imagen'];  
        $p->periodo = $pintura['periodo'];  
        $p->descripcion = $pintura['descripcion'];  
        $p->save();  
    }  
}
```

```
$this->command->info('Tabla pinturas inicializada con datos');  
}
```

- Por último tendremos que ejecutar el comando de Artisan que procesa las semillas (**php artisan db:seed**) y una vez realizado abriremos la base de datos para comprobar que se ha rellenado la tabla

### 3.- Uso de la base de datos

Vamos a actualizar los métodos del controlador PinturaController para que obtengan los datos desde la base de datos. Seguiremos los siguientes pasos:

- Modificar el método **index** para que obtenga toda la lista de pinturas desde la base de datos usando el modelo Pintura y que se lo pase a la vista.
- Modificar el método **show** para que le pase la pintura pasada por parámetro a la vista.
- Modificar el método **edit** para que le pase la pintura pasada por parámetro a la vista.

Ya no necesitaremos más el array de pinturas que habíamos puesto en el controlador, así que lo podemos eliminar.

Ahora tendremos que actualizar las vistas para que en lugar de acceder a los datos del array los obtenga del objeto con la pintura. Para esto cambiaremos en todos los sitios donde hayamos puesto `$pintura['campo']` por `$pintura->campo`.

Además, en la vista pinturas/index.blade.php, en vez de utilizar el índice del array (`$clave`) como identificador para crear el enlace a pinturas/{pinturas}, tendremos que utilizar la pintura:

```
<a href="{{ route('pinturas.show', $pintura ) }}">  
    {{$pintura->titulo}}  
</a>
```

Lo mismo en la vista pinturas/show.blade.php, para generar el enlace de editar pinturas tendremos que añadir llamar a la ruta pinturas.edit pasándole la pintura.

Por último, mostraremos el siglo en números romanos de la pintura. Para ello haremos un método en la clase Pintura llamado `getSiglo()`:

```
public function getSiglo(): string  
{  
    //Devolver el siglo calculado  
}
```

## Tablas relacionadas usando ORM

### 1.- Modificar los id por slug

Previamente habíamos añadido un campo **slug** en la tabla pinturas. El slug es una forma legible y válida para la URL de una publicación o página web. Facilita la búsqueda y el posicionamiento en los buscadores. Se suele sustituir los espacios por guiones y utilizar sólo letras minúsculas.

Modifica el fichero de rutas, las vistas y los controladores necesarios para pasar este parámetro y no el identificador.

Por ejemplo, habría que cambiar la URL [http://127.0.0.1/laravel\\_museo/public/pinturas/1](http://127.0.0.1/laravel_museo/public/pinturas/1) por [http://127.0.0.1/laravel\\_museo/public/pinturas/las-meninas](http://127.0.0.1/laravel_museo/public/pinturas/las-meninas)

Para que las rutas obtengan el slug en vez del id es necesario añadir en la clase Pintura (dentro de la carpeta Models) el siguiente método:

```
public function getRouteKeyName()
{
    return 'slug';
}
```

Acuérdate que los métodos **show** y **edit** de **PinturaController** tendrán que recibir como parámetro la \$pintura a mostrar o editar, por lo que la definición del método en el controlador tendrá que ser como la siguiente:

```
public function show(Pintura $pintura)
{
    return view('pinturas.show', compact('pintura'));
}
```

### 2.- Añadir una relación uno a muchos

Cada pintura tiene un **artista**. Pero un artista puede tener muchas pinturas. Para ello crear una migración **create\_artistas\_table** que tendrá un id, un nombre, slug, biografía (texto largo), fecha de nacimiento y fecha de fallecimiento (opcional).

Ahora habrá que modificar la migración de pintura para que tenga una clave foránea a la tabla artistas (artista\_id).

Vamos a suponer que una pintura es exclusiva de un artista.

El orden de las migraciones es muy importante. Para poder crear una pintura previamente deberá existir su artista. Por tanto, la migración del artista deberá ser anterior a la de la pintura. Para ello, simplemente debemos renombrar el fichero de la migración (poner una fecha anterior a la del artista).

Crear el modelo **Artista**.

Crear una semilla para añadir artistas a partir de un array (**ArtistaSeeder**).

También habrá que modificar el seeder de Pintura para asociar un artista a cada pintura.

*Fijarse en el orden de ejecución de los seeders*

Hacer un rollback de las migraciones, volver a migrar y ejecutar el seed (también se puede utilizar el comando **php artisan migrate:fresh --seed** para realizarlo todo de una vez)

Añadir el método **artista** en la clase Pintura

```
public function artista()
{
    return $this->belongsTo(Artista::class);
}
```

Y añadir el método pinturas en la clase Artista

```
public function pinturas()
{
    return $this->hasMany(Pintura::class);
}
```

Ahora, cada artista pasado a una vista Blade tendrá sus pinturas sin falta de hacer ninguna consulta en la base de datos.

También cada pintura tendrá su objeto Artista asociado.

### 3.- Modificar las vistas para mostrar correctamente el nombre del artista

Modifica la vista pinturas/index.blade.php para mostrar el nombre del artista




## Las Meninas

1656 - Diego Velázquez

**Periodo:** Barroco

Modifica la vista pinturas/show.blade.php para mostrar una relación de pinturas del mismo artista.

[Museo](#) [Listado de pinturas](#) [Nueva pintura](#) [Login](#)



## La Maja Desnuda

Artista: Francisco de Goya

Año: 1800 (siglo XVIII)


Periodo: Romanticismo

Retrato de una mujer desnuda, considerada controvertida en su época por la falta de simbolismo mitológico o histórico. La modelo mira directamente al espectador, lo que era poco común en representaciones similares.


La obra tiene una pareja, "La Maja Vestida", que representa a la misma mujer, pero vestida, y ambos cuadros se encuentran en el Museo del Prado. La técnica de Goya en ambas piezas muestra su maestría en la representación de la piel y las texturas.'

[✍ Editar](#) [◀ Volver a la galería](#)

Pinturas del mismo artista:



El 3 de mayo en Madrid



Saturno devorando a su hijo

4.- Modificar los formularios de añadir y editar una nueva pintura para que muestre un select en los artistas

Título

El 3 de mayo en Madrid

Artista

Peter Paul Rubens

Diego Velázquez

Francisco de Goya

Juan de Flandes

Peter Paul Rubens

Rogier van der Weyden

El Bosco

Fra Angélico

Alberto Durero

Rafael

El Greco

Imagen

Seleccionar archivo

Ningún archivo seleccionado

Editar pintura



## 5.- Añadir un formulario para añadir un nuevo artista

[Nueva pintura](#) [Nuevo artista](#)

Crear un nuevo artista

Nombre

Biografía

Fecha de nacimiento

Fecha de fallecimiento

Añadir artista

## 6.- EXTRA. Realizar la página para mostrar el detalle de un artista

En ella se podrá ver los detalles del artista, así como sus pinturas.



Curso 2024-2025  
Profesor: Iván Lorenzo



Laravel

DWES

DAW2

*Hoja07\_MVC\_09*

---

Seguimos desarrollando la web para la gestión de un museo realizando el procesamiento de formularios.

### 1.- Añadir y editar pinturas

En primer lugar, vamos a añadir las rutas que nos van a hacer falta para recoger los datos al enviar los formularios. Para esto editamos el fichero de rutas y añadimos dos rutas:

- Una ruta de tipo POST para la url pinturas que apuntará al método store del controlador PinturaController.
- Y otra ruta tipo PUT para la url pinturas/{pintura} que apuntará al método update del controlador PinturaController.

Editamos la vista pinturas/create.blade.php para que el action del formulario nos lleve a `{{ route('pinturas.store') }}`

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

- Añadimos en el action del formulario `{{ route('pinturas.update', $pintura) }}`
- Tenemos que modificar todos los inputs para que como valor del campo ponga el valor correspondiente de la pintura. Por ejemplo, en el primer input tendríamos que añadir `value="{{ $pintura->titulo }}"`. Realiza lo mismo para el resto de los campos. El único campo distinto será el de descripción ya que el input es tipo textarea, en este caso el valor lo tendremos que poner directamente entre la etiqueta de apertura y la de cierre.
- Añade debajo de `@csrf` el método del formulario (será put): `@method('put')`

Por último, tenemos que actualizar el controlador PinturaController con los dos nuevos métodos. En ambos casos tenemos que usar la inyección de dependencias para añadir la clase Request como parámetro de entrada.

Además, para cada método haremos:

- En el método **store** creamos una nueva instancia del modelo Pintura, asignamos el valor de todos los campos de entrada y los guardamos. Por último, después de guardar, hacemos una redirección a la ruta pinturas.show.

Crear un almacenamiento en el fichero **config/filesystems.php** llamado pinturas:

```
'pinturas' => [  
    'driver' => 'local',  
    'root' => public_path('imagenes'),  
    'visibility' => 'public',  
],
```

Para guardar la imagen, desde el método store del controlador haremos:

```
$pintura->imagen = $request->imagen->store("", 'pinturas');
```

- En el método **update** recibe la pintura por parámetro y actualizamos sus campos y los guardamos. Por último, realizamos una redirección a la pantalla con la vista detalle de la pintura editada.

## 2.- Añadir un nuevo artista

En primer lugar crearemos el método **store** en **ArtistaController**. Lo completaremos de modo similar al que hemos realizado con las pinturas.

Creamos la ruta **artistas** de tipo POST, que apunte al método del controlador creado.

Por último, modificamos el formulario para que el action nos lleve a `{{ route('artistas.store') }}`

Modificamos la web para la gestión del museo para añadir una relación muchos a muchos

### 1.- Añadir una relación muchos a muchos

Cada pintura puede estar en varias exposiciones. A su vez una exposición puede tener muchas pinturas.

Para ello crear un modelo **Exposicion**. Debemos indicar en el modelo que la tabla de la base de datos se llamará exposiciones.

Crear la migración **create\_exposiciones\_table**.

Tendrá un campo id, un nombre, slug, descripcion, fecha\_inicio y fecha\_fin

Después habrá que crear la tabla intermedia que relacionará pinturas y exposiciones. Para ello crea la migración **create\_exposicion\_pintura\_table**. Sólo tendrán 2 campos: `exposicion_id` y `pintura_id`. Ambos campos serán claves foráneas. También hay que añadir una clave primaria que será la unión de ambos campos.

Seguidamente tendremos que añadir un método en los modelos (Pintura y Exposicion).

Al ser una relación muchos a muchos tendremos que llamar al método `belongsToMany`.

En la clase Pintura añadiremos el siguiente método:

```
//Relación muchos a muchos
public function exposiciones()
{
    return $this->belongsToMany(Exposicion::class);
}
```

En la clase Exposicion:

```
//Relación muchos a muchos
public function pinturas()
{
    return $this->belongsToMany(Pintura::class);
}
```

Lo siguiente que haremos será crear un Factory que cree datos de prueba de exposiciones. Para ello teclearemos **php artisan make:factory ExposicionFactory**

También configuraremos la aplicación para que los factories se generen en español. Para ello modificamos el valor de **APP\_FAKER\_LOCALE** a 'es\_ES' dentro del fichero `.env`

En el método `definition` podremos hacer algo así:

```
public function definition()
{
    $nombre = $this->faker->sentence(3);
```

```
return [  
    'nombre' => $nombre,  
    'slug' => Str::slug($nombre),  
    'descripcion' => $this->faker->paragraph(4),  
    'fecha_inicio' => $this->faker->dateTimeBetween('-2 years', 'now'),  
    'fecha_fin' => $this->faker->dateTimeBetween('now', '+2 years'),  
];  
}
```

Luego añadiremos la siguiente línea al inicio del método run de la clase DatabaseSeeder:

```
Exposicion::factory(20)->create();
```

Eso hará que se creen 20 exposiciones.

Sólo faltaría añadir datos a la tabla “pivot” exposicion\_pintura. Para ello en el método run (justo después de llamar al método save) del seeder de **Pintura** añadimos lo siguiente:

```
$p->save();  
  
$p->exposiciones()->attach([  
    Exposicion::all()->skip(0)->take(10)->random()->id,  
    Exposicion::all()->skip(10)->take(10)->random()->id  
]);
```

Esto hará que cada pintura tenga 2 exposiciones. La primera será una entre los ids 1 y 10 y la segunda entre los ids 11 y 20.

Probamos todo ejecutando las migraciones: **php artisan migrate:fresh --seed**

Para finalizar **modifica la vista show** de pinturas para mostrar el nombre de las exposiciones de cada una.

## Exposiciones:

Ipsa non sed qui impedit. (Desde **2023-08-02** al **2025-05-12**)

Ut ratione ad fugiat. (Desde **2024-07-09** al **2025-11-11**)

[✎ Editar](#)[< Volver a la galería](#)

```
@if($pintura->exposiciones->count() > 0)
    <h4 class="mt-4">Exposiciones:</h4>
    <div class="row">
        <ul>
            @foreach($pintura->exposiciones as $exposicion)
                <li>{{ $exposicion->nombre }} (Desde <strong>{{ $exposicion->fecha_inicio }}</strong> al <strong>{{ $exposicion->fecha_fin }}</strong></li>
            @endforeach
        </ul>
    </div>
@endif
```

## 2.- Borrar una pintura

Añadir un formulario con únicamente un botón en la vista de detalle de la pintura. Se deberá utilizar `@method('DELETE')`. El action del formulario deberá llevar a la ruta `pinturas.destroy` que crearemos a continuación



Pinturas del mismo artista:



## El 3 de mayo en Madrid

Artista: Francisco de Goya

Año: 1814 (siglo XIX)

Periodo: Romanticismo

Una representación del brutal fusilamiento de los rebeldes españoles por las francesas.

Goya captura el horror y la desesperación en las expresiones de los prisioneros, especialmente el hombre con la camisa blanca en el centro.

Este cuadro es uno de los grandes manifiestos antibélicos y sigue siendo una historia del arte. La crudeza y el realismo de la escena contrastan con el lirismo de la misma época.

### Exposiciones:

Cargar exposición

Editar

Eliminar

Volver a la galería

Crear una ruta de tipo **delete** que apunte al método **destroy** de **PinturaController**. En él habrá que borrar la pintura pasada y redirigir a la página principal enviando un mensaje para mostrarlo en ella.

Si obtenemos un error al no tener el borrado en cascada establecido en la migración de la tabla intermedia de la relación muchos a muchos: "Integrity constraint violation: 1451 Cannot delete or update a parent row: a foreign key constraint fails..." debemos hacer un detach:

```
try
{
    $pintura->exposiciones()->detach();
    $pintura->delete();
    return redirect()->route('pinturas.index')->with('mensaje', 'Pintura eliminada correctamente');
}
catch (\Illuminate\Database\QueryException $ex)
{
    return redirect()->route('pinturas.index')->with('mensaje', 'Fallo al eliminar la pintura: ' . $ex);
}
```

Museo   Listado de pinturas   Nueva pintura   Nuevo artista

Pintura eliminada correctamente





Finalizamos la web para la gestión de un museo realizando la autenticación de usuarios

### 1.- Migración de la tabla usuarios y creación de vistas y rutas

Debemos comprobar que cuando añadimos las migraciones se haya creado la tabla users en nuestra base de datos.

### 2.- Seeders de usuarios

Vamos a rellenar la tabla con datos iniciales.

Para esto editamos el fichero de semillas situado en database/seeds/**DatabaseSeeder.php** y añadiremos el siguiente código:

```
DB::table('users')->delete();
User::factory(5)->create();

User::factory()->create([
    'name' => 'tunombre',
    'email' => 'tucorreo@educastur.es',
    'password' => bcrypt("tupassword")
]);
```

Recuerda que para guardar el password es necesario encriptarlo manualmente usando el método **bcrypt**.

Por último, tendremos que ejecutar el comando de Artisan que procesa las semillas.

Una vez realizado esto comprobamos en la base de datos que se han añadido los usuarios a la tabla users.

### 3.- Sistema de autenticación

En primer lugar, haremos una **copia de nuestro fichero routes/web.php** (puede que se borre en la instalación de Breeze)

Seguidamente, instalaremos Laravel Breeze

A continuación, edita el fichero routes/web.php y realiza las siguientes acciones:

- Borra la ruta generada:

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

- Añade todas tus rutas anteriores de la copia que has realizado.
- Añade un *middleware* que aplique el filtro **auth** para proteger las rutas de crear y editar pinturas. Protege también la creación de nuevos artistas.
- Revisa mediante el comando de Artisan **php artisan route:list** las nuevas rutas y que el filtro auth se aplique correctamente.

Realizar además las siguientes operaciones:

- Modificar el método **store** del controlador **Auth/AuthenticatedRouteController**. Comentar la línea `return redirect()->intended(route('dashboard', absolute: false));` y cambiarla por `return redirect()->intended('/')`; Cuando se realice login redirigirá a la ruta /.
- “Traducir” los datos de la aplicación para que se muestre en castellano. Esto se puede hacer “a mano”, pero también se puede instalar las traducciones en español que trae por defecto Laravel. Para ello hacemos lo siguiente:
  - `composer require laravel/lang`
  - `php artisan laravel:install-lang`

Esto creará la carpeta `resources` → `lang` → es con ficheros con las traducciones

- Cambia el parámetro `APP_LOCALE` del fichero `.env` a `'es'`

Comprueba en este punto que el sistema de autenticación funciona correctamente: no te permite entrar a las rutas protegidas si no estás autenticado, puedes acceder con los usuarios definidos en el fichero de semillas y funciona el botón de cerrar sesión.

#### 4.- Paginación de las pinturas

Por último, vamos a mostrar el listado de pinturas paginado. Mostraremos las pinturas de 8 en 8.

Para ello haremos lo siguiente:

- En el método `index` de `PinturaController` cambiaremos el modo de recuperar las pinturas de la base de datos: `$pinturas = Pintura::paginate(8);`
- Por último, en la vista `pinturas/index.blade.php` mostraremos un enlace para que nos cree el paginador: `{{ $pinturas->links('pagination::bootstrap-5') }}`




Si quisiésemos modificar las vistas de la paginación podemos hacer:

**php artisan vendor:publish --tag=laravel-pagination**

Esto nos copiará las vistas a nuestra carpeta resources/views/vendor/pagination

Podemos modificar el fichero bootstrap-5.blade.php y cambiar los textos necesarios para mostrarlos en castellano



Mostrando del **1** al **8** de **13** resultados

En esta actividad añadiremos roles a nuestros usuarios.

### 1. - Creación de usuarios

En primer, lugar, vamos a crear los siguientes usuarios:

Usuario	Password
<a href="mailto:gestor@educastur.es">gestor@educastur.es</a>	gestor
<a href="mailto:admin@educastur.es">admin@educastur.es</a>	admin

Lo podemos hacer desde el seeder de usuarios como en el ejercicio anterior. Recuerda ejecutar el comando de Artisan que procesa las semillas.

### 2.- Instalación de librería laravel-permission

Realiza la instalación de la librería laravel-permission de Spatie y su configuración tal y como se ha descrito en los apuntes.

### 3.- Creación de roles

En el seeder de usuarios vamos a crear 2 roles: administrador y gestor. Asignarlos respectivamente a los usuarios admin y gestor creados anteriormente.

Hay que añadir el Trait HasRoles en la clase User:

```
use HasRoles;
```

Vuelve a ejecutar las semillas de datos.

Si da algún error en la creación de los roles podemos intentar limpiar la caché:

```
php artisan cache:forget spatie.permission.cache
```

### 4.- Protección de las rutas

Por último, vamos a proteger las siguientes rutas:

- La creación de una pintura únicamente podrá ser realizada por los administradores

- La edición de una pintura podrá ser realizado tanto por administradores como por gestor, pero no por usuarios sin roles

Para poder usar el middleware tenemos que definirlo en el fichero bootstrap/app.php:

```
->withMiddleware(function (Middleware $middleware) {  
    $middleware->alias([  
        'role' => \Spatie\Permission\Middleware\RoleMiddleware::class  
    ]);  
});
```

Para añadir varios middlewares podemos hacer lo siguiente:

```
->middleware(['auth','role:administrador']);
```

Y si queremos que se pueda acceder con algún rol utilizamos el carácter |

```
->middleware(['auth', 'role:administrador|gestor']);
```

Si intentamos acceder a una página sin el rol necesario nos saldrá un error como el siguiente:

403 | USER DOES NOT HAVE THE RIGHT ROLES.

Otro museo nos ha solicitado ayuda para realizar una aplicación web y poder gestionar información de los pintores, cuadros y exposiciones. Crear un proyecto Laravel llamado **laravel\_pintores**

Del pintor nos interesa guardar un id, su nombre, slug, país y fecha de nacimiento. Un cuadro tendrá un id, un nombre, una imagen y si está disponible o no (booleano). Además, un cuadro será únicamente de un pintor.

Por otra parte, habrá exposiciones. Un cuadro puede estar en muchas exposiciones y una exposición puede tener muchos cuadros. Una exposición únicamente tiene un id, un nombre y un slug.

Se deberán crear las siguientes rutas:

- `/:` deberá redirigir a pintores
- **pintores:** se mostrará un listado con todos los pintores de la pinacoteca y el número de cuadros de cada uno.

Museos Todos los pintores Nuevo cuadro		
Nombre	País	Cuadros
<a href="#">Pablo Picasso</a>	España	3
<a href="#">Vincent van Gogh</a>	Países Bajos	2
<a href="#">Salvador Dalí</a>	España	1
<a href="#">Diego Velázquez</a>	España	2
<a href="#">Francisco de Goya</a>	España	2


- **pintores/{slug}:** se mostrará el pintor (nombre y país de nacimiento). En esta vista también se mostrarán todos los cuadros pertenecientes al pintor (nombre, fotografía, las exposiciones en la que está y un enlace para modificar su disponibilidad)

Museos
Todos los pintores
Nuevo cuadro

## Pablo Picasso

País: España


Cuadros

Guernica


Exposición de Retratos

Exposición de Paisajes


Marcar como No disponible

Las señoritas de Avignon


Exposición de Retratos

Exposición de Paisajes

Marcar como Disponible

La mujer que llora


Exposición de Arte Moderno

Exposición de Retratos

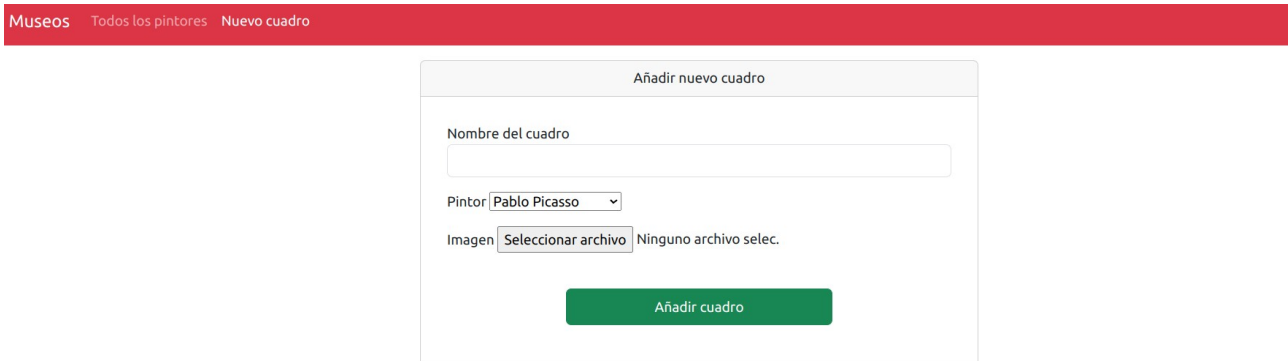
Marcar como No disponible

El enlace anterior deberá mostrar el texto “Marcar como no disponible” en caso que el cuadro esté disponible (y mostrarlo en rojo).

Si el cuadro no está disponible podremos ponerlo disponible.

El enlace del botón será la ruta `cuadros/cambiarEstado/{cuadro}`

- **cuadros/cambiarEstado/{cuadro}**: si el cuadro está disponible deberá ponerlo no disponible y viceversa. Luego deberá redirigir a la página del pintor del cuadro.
- **cuadros/crear**: se mostrará una vista que contenga un formulario con el que crear un cuadro. En esta vista se deberá mostrar un listado de todos los pintores (select), para seleccionar aquel de cuya autoría es el cuadro.



- El resto de las rutas necesarias para que funcione la aplicación (si es que hiciese falta alguna más). NO es necesario implementar el mecanismo de autenticación de Laravel.

Crear los siguientes controladores y los métodos que necesites en ellos:

- **PintoresController**
- **CuadrosController**

Crear los modelos **Pintor**, **Cuadro** y **Exposicion**

Se deberá crear configurar y crear una base de datos llamada **dwes\_laravel\_pintores**. Además, habrá que crear cuatro **migraciones** para crear las tablas.

Crear un **seeder** para añadir datos de prueba. Crea un **factory** para crear 2 pintores ficticios.

Crear 3 **vistas** utilizando un layout genérico y el navbar proporcionado por el profesor en la práctica anterior:

- `pintores/index`
- `pintores/mostrar`
- `cuadros/crear`