
UNIVERSIDAD
DE GRANADA

# Memoria PE2

FRANCISCO JAVIER ROMERO GARCÍA

18 DE JUNIO DE 2023

Programación Web

GII 3

# 1. Introducción

Para este proyecto de prácticas la he llevado a cabo haciendo uso de contenedores y dos de los frameworks más conocidos y populares para PHP y Javascript, React y Laravel. El resultado se puede observar en http://bahia.ugr.es:30500

# 2. Tecnologías usadas.

A continuación realizaré una breve descripción de las tecnologías usadas en la realización de este proyecto de prácticas:

## - Podman

Podman es una herramienta de administración de contenedores que permite a los usuarios ejecutar y gestionar contenedores de manera eficiente en sistemas Linux. A diferencia de Docker, que utiliza un enfoque cliente-servidor, Podman adopta un enfoque sin demonio, lo que significa que los contenedores se ejecutan en el mismo contexto que el usuario y no requieren privilegios adicionales,


## - Docker-compose

Docker-compose es una herramienta que permite definir y administrar aplicaciones multicontenedor de manera sencilla. Con Docker Compose, los usuarios pueden definir la configuración de múltiples contenedores en un archivo YAML, lo que facilita la definición y el despliegue de aplicaciones complejas que constan de varios servicios interconectados


## - Laravel

Laravel es un popular framework de desarrollo de aplicaciones web basado en PHP, creado con el objetivo de simplificar y agilizar el proceso de desarrollo.

Una de las características distintivas de Laravel es su elegante sintaxis y su enfoque en la legibilidad del código. Utiliza un patrón de diseño MVC (Modelo Vista Controlador) para separar la lógica de negocio, la presentación y el manejo de las solicitudes, lo que facilita la organización y el mantenimiento del código.

Laravel ofrece una amplia gama de características y funcionalidades que simplifican tareas comunes en el desarrollo web, como el enrutamiento de URL, la gestión de las bases de datos, el manejo de formularios, autenticación de usuarios y la generación de vistas. Además de proporcionar una capa de abstracción para interactuar con la base de datos, lo que facilita el trabajo con distintos SGDB.


- **Inertia**

  Inertia es una biblioteca que permite a los desarrolladores crear aplicaciones web de una sola página (SPA) utilizando frameworks como Laravel, Vue.js o React, mientras mantienen la experiencia de desarrollo familiar y productiva del lado del servidor.

  La idea central de Inertia es combinar las ventajas de un SPA, como la interactividad y la respuesta rápida, con la simplicidad y la eficiencia del desarrollo del lado del servidor. En vez de construir una API y una interfaz de usuarios separadas y completamente independientes, Inertia utiliza el enrutamiento del lado del servidor y transmite los componentes de la interfaz del usuario de forma eficiente al cliente , permitiendo una navegación sin interrupciones y una experiencia de usuario fluida.

  Con Inertia, los desarrolladores pueden crear componentes reutilizables en el lado del cliente utilizando bibliotecas como Vue.js o React, y estos componentes se pueden renderizar y actualizar de manera eficiente dentro del contexto de una página generada por el servidor. Esto elimina la necesidad de administrar estados duplicados y simplifica el flujo de desarrollo, al tiempo que proporciona un rendimiento óptimo.


- **React**

React es una biblioteca de Javascript desarrollada por Facebook que se utiliza para construir interfaces de usuario interactivas y reactivas. Es ampliamente utilizada en el desarrollo web moderno y se destaca por su eficiencia y rendimiento.

La principal característica de React es su enfoque en la creación de componentes reutilizables. Los componentes en React son bloques de construcción independientes que encapsulan la lógica y la interfaz de usuario de una parte específica de una aplicación. Estos componentes se pueden combinar y anidar para construir interfaces complejas y dinámicas.

React utiliza un modelo de programación basado en componentes y utiliza una sintaxis especial llamada JSX para definir la estructura de la interfaz de usuario. JSX combina JavaScript y HTML, lo que facilita la escritura de código y la manipulación del DOM de manera declarativa.


- **Tailwind CSS**

Tailwind CSS es un framework de CSS de utilidad altamente personalizable que se centra en la construcción de interfaces de usuario rápidas y eficaces. A diferencia de otros frameworks de CSS que proporcionan estilos predefinidos, Tailwind CSS se basa en clases de utilidad que se aplican directamente en el marcado HTML.

En lugar de escribir estilos de CSS personalizados, los desarrolladores pueden usar las clases de utilidad de Tailwind para aplicar rápidamente estilos específicos. Esto permite un enfoque más rápido y eficiente para el desarrollo de interfaces, ya que no es necesario escribir CSS adicional y las clases se pueden reutilizar fácilmente.


# 3. Realización de proyecto

En esta sección explicaré con detalle cómo llevé a cabo el desarrollo de la aplicación.

## 3.1. Instalación y setup

El hecho de contenerizar se refiere al proceso de adaptar una aplicación y sus componentes de manera que le sea posible ejecutarse en entornos llamados contenedores. Para la realización de esta práctica usaremos Podman y Docker-Compose para contenerizar nuestra aplicación de Laravel en lo siguiente:

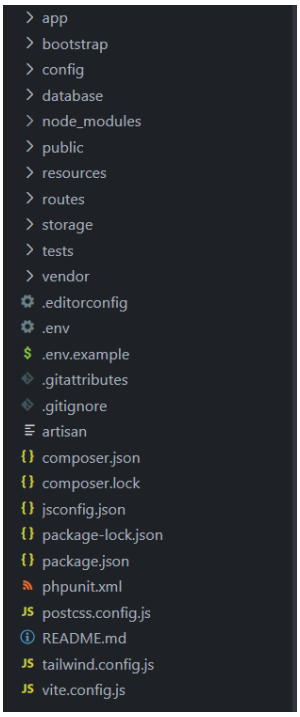
- Un servicio *app* corriendo php:8.2-fpm.
- Un servicio *mysql* corriendo mysql:5.7.
- Y un servicio *nginx* que usará el servicio app para servir el contenido de la aplicación al usuario final.

### 3.1.1. Obtener proyecto de Laravel.

Para empezar podemos obtener un proyecto de prueba de Laravel desde su repositorio de GitHub o si ya contamos con PHP y Composer instalado con la orden

**composer create-project laravel/laravel pe2**

Una vez instalado si accedemos al directorio deberíamos tener la siguiente estructura de directorios:




Lo primero que tenemos que hacer es cambiar la configuración de Laravel, para ello deberemos usar el archivo .env situado en el directorio raíz de nuestro proyecto. En este archivo especificaremos cosas como las configuraciones de conexión a la base de datos, url de la aplicación, etc. Para nuestro proyecto solo vamos a necesitar cambiar la configuración de la base de datos, de la cual crearemos su contenedor más adelante. Para ello debemos cambiar los parámetros DB_CONNECTION, DB_HOST, DB_PORT, DB_DATABASE, DB_USERNAME, DB_PASSWORD, en el que DB_CONNECTION seleccionamos el tipo de base de datos que usaremos (en nuestro caso 'mysql'), DB_HOST el host de la base de datos (en nuestro caso será el nombre del contenedor en el que tenemos mysql), etc.

Una vez hemos configurado el entorno de Laravel continuaremos con la configuración de los contenedores.

### 3.1.2 Configuración del Dockerfile de la aplicación

Aunque para SQL y Nginx voy a usar una imagen oficial, es necesario crear una imagen personalizada. Para ello crearemos un nuevo archivo Dockerfile.

Nuestra imagen estará basada en la imagen oficial de Docker php:8-2-fpm. Y, encima de esta instalaremos algunos módulos extra de PHP , la herramienta de gestión Composer y Node. Además

crearemos un nuevo usuario por si es necesario ejecutar comandos artisan y composer mientras desarrollamos la aplicación, para ello usamos ARG user, y además 'ARG uid' para asegurarnos de que la configuración se asegure de que el usuario dentro del contenedor sea el mismo que el usuario en la máquina local. Así nos aseguramos de que cuando se creen algún archivo en el contenedor también lo haga en local.

Finalmente el Dockerfile nos quedaría de la siguiente forma:

Dockerfile

```
FROM php:8.2-fpm

ARG user
ARG uid

RUN apt-get update && apt-get install -y \
    git \
    curl \
    libpng-dev \
    libonig-dev \
    libxml2-dev \
    zip \
    unzip

RUN apt-get clean && rm -rf /var/lib/apt/lists/*

RUN docker-php-ext-install pdo_mysql mbstring exif pcntl bcmath gd

COPY --from=composer:latest /usr/bin/composer /usr/bin/composer

RUN curl -fsSL https://deb.nodesource.com/setup_18.x | bash -
RUN apt-get install -y nodejs

RUN chmod +x /home

RUN useradd -G www-data,root -u $uid -d /home/$user $user
RUN mkdir -p /home/$user/.composer && \
    chown -R $user:$user /home/$user
WORKDIR /var/www

USER $user
```

### 3.1.3. Configuración de Nginx.

A continuación configuraremos Nginx, para ello primero crearemos una carpeta llamada nginx en el directorio raíz del proyecto y dentro de este un fichero default.conf en el cual añadiremos los siguiente:

```
server{
    listen 80;
    index index.php index.html;
    server_name localhost;
    error_log /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;
    root /var/www/html/public;

    charset utf-8;

    location / {
        try_files $uri $uri/ /index.php?$query_string;
        gzip_static on;
    }

    location = /favicon.ico { access_log off; log_not_found off; }
    location = /robots.txt  { access_log off; log_not_found off; }

    location ~ \.php$ {
        try_files $uri =404;
        fastcgi_split_path_info ^(.+\.php)(/.+)$;
        fastcgi_pass app:9000;
        fastcgi_index index.php;
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
    }
}
```

De esta forma configuraremos Nginx para que escuche al puerto 80 y use el index.php como índice default. También establecerá el directorio raíz a /var/www/public, y configurar Nginx para que use el servicio ‘app’ en el puerto 9000 para procesar archivos php.

### 3.1.4. Creación del entorno multi-contenedor con Docker Compose

Docker-Compose permite crear entornos multi-contenedor para aplicaciones. Para configurar nuestros servicios crearemos un docker-compose.yml y lo situaremos en el directorio raíz de nuestra aplicación.

Definiremos tres servicios: app, mysql y  nginx.

El servicio app creará una imagen basada en el Dockerfile que creamos anteriormente. Este contenedor correrá un servidor php-fpm para analizar código php y pasarle los resultados al servicio nginx, que correrá en un contenedor aparte. Y el servidor mysql define un contenedor que corre un servidor MySQL 5.7. Todos nuestros servicios compartirán una red puente llamada ‘networkapp’.

- **El servicio app.**

El servicio app contendrá un contenedor llamado app , y como se ha dicho anteriormente se basará en la imagen creada anteriormente en el Dockerfile. La configuración que debemos usar en este es la siguiente:

- build: con esta configuración indicamos que construya una imagen local para el servicio, usando el Dockerfile indicado. Los argumentos user y uid que requerimos en el Dockerfile se le aportan también.
- working_dir: en este apartado indicamos que el directorio por defecto para el servicio será /var/www/html

- volumes: creará un volumen compartido que sincronizará los contenidos del directorio actual con el directorio /var/www/html de dentro del contenedor.
- networks: indica a la red a la que se conectará el servicio.

```yaml
app:
  build:
    args:
      user: laraveluser
      uid: 1000
    context: .
    dockerfile: ./Dockerfile
  container_name: app
  working_dir: /var/www/html
  volumes:
    - .:/var/www/html
  networks:
    - networkapp
```

- **El servicio nginx.**

El servicio nginx usará una imagen prehecha: nginx:stable-alpine. Crea un contenedor llamado nginx, y usa la definición de puertos para redireccionar el puerto **30500** de bahía al puerto 80 del contenedor.

Además, en el apartado volumes crearemos dos volúmenes compartidos. El primero sincronizará los contenidos como el del servicio app, de esta manera cuando realicemos cambios en local estos se reflejarán en la aplicación servida por Nginx dentro del contenedor. Y el segundo volumen se encargará de que nuestro archivo de configuración de Nginx se copie a la carpeta de configuración de Nginx del contenedor.

```yaml
nginx:
  image: nginx:stable-alpine
  container_name: nginx
  ports:
    - 30500:80
  volumes:
    - .:/var/www/html
    - ./nginx:/etc/nginx/conf.d
  networks:
    - networkapp
```

- **El servicio mysql.**

El servicio mysql usará una imagen de MySQL 5.7 prehecha. Como Docker Compose carga automáticamente las variables del archivo .env que se encuentre en el mismo directorio que el docker-compose.yml, por lo que obtendrá la configuración de la base de datos desde el archivo .env que hemos definido anteriormente.

```yaml
mysql:
  image: mysql:5.7
  container_name: mysql
  restart: always
  environment:
    MYSQL_ROOT_PASSWORD: '${DB_PASSWORD}'
    MYSQL_ROOT_HOST: '${DB_HOST}'
    MYSQL_DATABASE: '${DB_DATABASE}'
    MYSQL_USER: '${DB_USERNAME}'
    MYSQL_PASSWORD: '${DB_PASSWORD}'
  ports:
    - 30002:3306
  volumes:
    - dbdata:/var/lib/mysql
  networks:
    - networkapp
```

### 3.1.5. Ejecutando la aplicación con Docker Compose

Una vez definido todos los archivos necesarios, usaremos los comandos de docker-compose para construir la imagen de nuestra aplicación y correr los servicios que hemos especificado.

Para construir la aplicación ejecutaremos:

**docker-compose build app**

Una vez que haya terminado podemos correr el entorno en segundo plano con:

**docker-compose up -d**

Este comando correrá los 3 contenedores que hemos especificado anteriormente, si deseas obtener información acerca de los servicios con:

**docker-compose ps**

veremos un resultado como el siguiente:

```
['podman', '--version', '']
using podman version: 4.5.0
podman ps -a --filter label=io.podman.compose.project=pe2
CONTAINER ID  IMAGE                                COMMAND               CREATED      STATUS      PORTS                     NAMES
7339fb6d9233  docker.io/library/nginx:stable-alpine  nginx -g daemon o...  3 weeks ago  Up 3 weeks  0.0.0.0:30500->80/tcp     nginx
30bad4985f76  docker.io/library/mysql:5.7           mysqld                3 weeks ago  Up 3 weeks  0.0.0.0:30002->3306/tcp   mysql
c7b1474ff953  localhost/pe2_app:latest             php-fpm               3 weeks ago  Up 3 weeks                            app
exit code: 0
```

El entorno ya estaría construido y corriendo, ahora si necesitamos ejecutar cualquier comando sobre los contenedores deberemos usar el siguiente comando:

**docker-compose exec ‘nombre contenedor’ ‘comando’**

### 3.1.6. Últimos pasos instalación

Por último solo quedaría instalar Laravel-Breeze ( una implementación simple con todas las características de autenticación) y React para ello ejecutamos los siguientes dos comandos:

**docker-compose exec app composer require laravel/breeze --dev**

**docker-compose exec app php artisan breeze:install react**

Y por último deberíamos correr las migraciones de la base de datos para poblar la base de datos con las tablas por defecto de Laravel y de Breeze.

## 3.2. Desarrollo de la aplicación en Laravel.

Para la explicación de cómo se realiza el desarrollo de la aplicación, me centraré en la explicación de cómo se crean las películas y todo lo que las envuelve, ya que el proceso es el mismo para el resto.

### 3.2.1. Modelos, migraciones y controladores.

Para permitir que los administradores puedan crear/modificar/borrar películas y el público en general puedan verlas necesitaremos crear modelos, migraciones, y controladores. A continuación voy a explicar en qué consisten estas:

- Los **modelos** ofrecen una poderosa y agradable interfaz para que puedas interactuar con las tablas de la base de datos.
- Las **migraciones** nos permiten crear y modificar fácilmente las tablas de la base de datos. Estas nos aseguran de que la misma estructura de base de datos existe en cualquiera en la que nuestra aplicación corra.
- Los **controladores** son los responsables de procesar las peticiones hechas por la aplicación y devolver una respuesta.

Como he dicho anteriormente prácticamente cada una de las características que añadas a una aplicación involucra los modelos/migraciones/controladores.
Para crear estas tres de una vez podemos ejecutar el siguiente comando en nuestro contenedor app, (en el caso de las películas):

**docker-compose exec app php artisan make:model -mrc Movie**

Con este comando crearemos los siguiente archivos:

- app/Models/Movie.php - El modelo.
- databases/migrations/<marca_de_tiempo>_create_movies_table.php - La migración que creará la tabla en la base de datos.
- app/Http/Controller/MovieController.php - El controlador HTTP que recibe las peticiones y devuelve las respuestas.

### 3.2.2. Routing.

También es necesario crear “URLs” para nuestros controladores. Podemos lograr esto añadiendo “rutas”, las cuales son administradas desde el directorio ‘routes’ del proyecto. En el caso de las películas como vamos a usar el controlador, podemos usar **Route::resource()**, lo cual creará las rutas con una estructura de URL convencional.

Podemos definir las siguientes rutas: **index** (mostrar las películas), **create** (mostrar formulario de creación), **store** (almacenar un nuevo objeto con los datos obtenidos en '**create**'), **show** (muestra un objeto en concreto), **edit** (formulario para editar un objeto en concreto), **update** (modifica el objeto con los datos obtenidos en **'edit'**) y **destroy** (borrar un objeto de la base de datos permanentemente). Estos refiriéndose a los diferentes métodos de nuestro controlador.

Además podremos usar Middlewares para filtrar las peticiones, que explicaremos en el siguiente apartado.

Para el ejemplo de las películas:

```php
Route::resource('movies', MovieController::class)
    ->only('index','create', 'store', 'edit', 'update', 'destroy')
    ->middleware('admin', 'verified', 'auth');
```

```php
Route::get('/', function () {
    return Inertia::render('Index', [
        'regular' => Movie::where('status', 'regular')
            ->with(['photos' => fn ($query) => $query->where('type', 'poster')->select('movie_id', 'src')])
            ->limit(3)
            ->get(),
        'premiere' => Movie::where('status', 'premiere')
            ->with(['photos' => fn ($query) => $query->where('type', 'poster')->select('movie_id', 'src')])
            ->limit(3)
            ->get(),
        'soon' => Movie::where('status', 'soon')
            ->with(['photos' => fn ($query) => $query->where('type', 'poster')->select('movie_id', 'src')])
            ->limit(3)
            ->get(),
        'popular' => Movie::orderBy('score', 'desc')
            ->with(['photos' => fn ($query) => $query->where('type', 'poster')->select('movie_id', 'src')])
            ->limit(3)
            ->get(),
    ]);
});
```

En esta última podemos observar que también es posible crear los métodos indicando de qué tipo de verbo HTTP se basa junto con la 'URL' y lo que se desea devolver.

### 3.2.3. Middleware.

Anteriormente hemos mencionado y observado en las capturas el Middleware, y esta es una parte esencial del routing, ya que es un mecanismo muy conveniente para inspeccionar y filtrar peticiones HTTP. Para nuestra aplicación necesitamos que sólo los usuarios que son administradores sean capaces de crear/modificar/borrar películas y que el resto de usuarios no pueden ni acceder a los formularios ni intentar realizar llamadas a la url de modificación de la BD. Para ello he creado un Middleware llamado admin que se asegure de que solo sean los administradores los que realicen dichas operaciones.

Para crear un Middleware podemos ejecutar el siguiente comando:

**docker-compose exec app php artisan make:middleware IsAdmin**

Este comando creará una clase en el directorio app/Http/Middleware

```php
<?php

namespace App\Http\Middleware;

use Closure;
use Illuminate\Http\Request;
use Symfony\Component\HttpFoundation\Response;

class IsAdmin
{
    /**
     * Handle an incoming request.
     *
     * @param  \Closure(\Illuminate\Http\Request): (\Symfony\Component\HttpFoundation\Response)  $next
     */
    public function handle(Request $request, Closure $next): Response
    {
        return (auth()->check() && auth()->user()->is_admin) ? $next($request) : abort(403, 'Access denied');
    }
}
```

Como observamos en la captura, comprobamos de que el usuario está identificado (no es invitado) y que el usuario consta con el campo de is_admin a true, si estas dos condiciones se cumple permite la petición en otro caso devuelve un código '403 Access denied'.

Finalmente para realizar uso de este Middleware simplemente debemos añadirlo en el apartado middleware de la ruta

```php
Route::resource('movies', MovieController::class)
    ->only('index','create', 'store', 'edit', 'update', 'destroy')
    ->middleware('admin', 'verified', 'auth');
```

### 3.2.4. Uso de controladores, modelos y migraciones.

Para explicar el uso de dichas herramientas mostraré cómo realicé la creación y guardado de las películas.

#### 3.2.4.1. Migración.

La creación de las migraciones son muy sencillas, cuando abres el archivo de migración creado anteriormente con el comando, en el método up() en Schema::create() se añaden los parámetros de la siguiente forma:

**$table→data_type('nombre')→opciones**

Resultando en lo siguiente:

```php
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('movies', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->longText('sinopsis');
            $table->string('director');
            $table->string('cast');
            $table->string('genre');
            $table->integer('duration');
            $table->integer('score');
            $table->enum('status', ['regular', 'premiere', 'soon']);
            $table->timestamps();
        });
    }
```

```php
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('movie_photos', function (Blueprint $table) {
            $table->id();
            $table->foreignId('movie_id')->constrained()->cascadeOnDelete();
            $table->string('src');
            $table->enum('type', ['poster', 'showcase']);
            $table->timestamps();
        });
    }
```

Una vez terminados los cambios tenemos que migrar para ello volvemos a:

**docker-compose exec app php artisan migrate**

### 3.2.4.2. Modelo

En el modelo nos encargaremos de definir las relaciones entre clases. Poniendo como ejemplo a nuestra clase Movie, sabemos que esta cuenta con varias imágenes que sólo pertenecen a una película, al igual que los comentarios (relación uno a muchos). Entonces para ello tenemos que definir una relación HasMany en las películas, tal que:

```php
public function photos(): HasMany
{
    return $this->hasMany(MoviePhoto::class);
}

public function comments(): HasMany
{
    return $this->hasMany(Comment::class);
}
```

Además en los modelos también podemos garantizar una protección de asignación masiva, ya que muchas veces si le pasamos a nuestro modelo todo los datos sería muy peligroso, ya que la gente podría modificar campos importantes como 'is_admin'. Para ello podemos activar dicha protección en nuestro modelo marcando los atributos "seguros" como "fillable" de la siguiente manera:

```php
class Movie extends Model
{
    use HasFactory;

    protected $fillable = [
        'title',
        'sinopsis',
        'director',
        'cast',
        'genre',
        'duration',
        'score',
        'status',
    ];
```

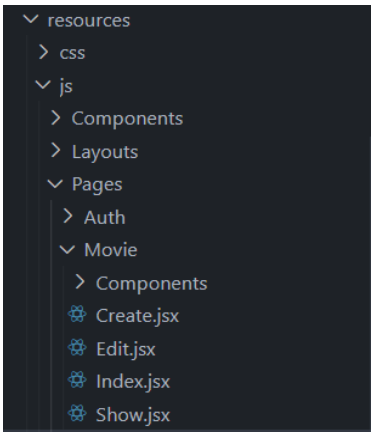
De esta forma solo los parámetros especificados son con los que se puede trabajar.

### 3.2.4.3. Controlador.

Como he descrito anteriormente, en el controlador nos encargamos de devolver una respuesta a las peticiones HTTP, y podemos diferenciarlos en dos tipos, devolver vistas o trabajar con la clase a la que pertenece el controlador ya sea crear, modificar o borrar.
En el caso de las vistas ya que usamos Inertia con React, lo que devolvemos en el método es una función para que se renderice nuestro componente de React:

```php
/**
 * Display the specified resource.
 */
public function show(Movie $movie)
{
    return Inertia::render('Movie/Show', [
        'movie'=> $movie,
        'poster'=> $movie->photos()->where('type', 'poster')->get(['src'])->first(),
        'showcases' => $movie->photos()->where('type', 'showcase')->get(['src']),
        'comments' => $movie->comments()->with('user')->orderBy('created_at', 'desc')->get(),
    ]);
}
```

En la imagen podemos observar que esta función sirve para mostrar una película en concreto. Mirando detenidamente la función render contiene dos parámetros, el primero se encarga de especificar dónde se encuentra el componente que debemos renderizar (en este caso en la carpeta Movie el componente Show, y esto siempre dentro de resources/js/Pages)




Y el segundo parámetro son la información necesaria para la página, que se le pasan en forma de 'props', lo cual simplifica mucho las cosas, y es una de los fuertes de Inertia:

```jsx
export default function Show({auth, movie, poster, showcases, comments}){
    return (
        <>
            {auth.user ? (
                <AuthenticatedLayout
                user={auth.user}
                header={<h2 className="font-semibold text-xl text-gray-800 leading-tight">{movie.title.toUpperCase()}</h2>}
                />
            ) : (
                <NonAutheticatedLayout/>
            )}
            <main className='min-h-screen bg-gray-100'>
                <div id='info-wrapper' className="max-w-5xl mx-auto pt-4 px-4 lg:px-8 lg:pt-8 flex flex-wrap">
                    <div className="basis-1/4 ">
                        <MoviePosterInfo numComments={comments.length} movie={movie} poster={poster}/>
                    </div>
                    <div className="basis-[calc(67%-26px)] ml-12 mt-3 p-4 rounded-md h-fit bg-white">
                        <div id='movie-sinopsis' className=" mb-3">
                            <p className=" text-sky-950 font-semibold">Sinopsis</p>
                            <p>{movie.sinopsis}</p>
                        </div>
```

Por último la manera de trabajar con las clases es muy sencilla y amigable con el desarrollador:

```php
public function store(Request $request): RedirectResponse
{
    $validated = $request->validate([
        'title' => 'required|string|max:50|unique:movies',
        'sinopsis' => 'required|string|max:500',
        'director' => 'required|string|max:30',
        'cast' => 'required|string|max:60',
        'genre' => 'required|string|max:30',
        'duration' => 'required|integer',
        'score' => 'required|integer|min:0|max:10',
        'status' => 'required|string|in:regular,premiere,soon',
        'poster' => 'required|string|max:50',
        'showcases' => 'required|string|max:250',
    ]);

    $movie = Movie::create([
        'title' => $validated['title'],
        'sinopsis' => $validated['sinopsis'],
        'director' => $validated['director'],
        'cast' => $validated['cast'],
        'genre' => $validated['genre'],
        'duration' => $validated['duration'],
        'score' => $validated['score'],
        'status' => $validated['status'],
    ]);

    $movie->photos()->create([
        'src' => $validated['poster'],
        'type' => 'poster',
    ]);

    $showcases = explode(' - ', $validated['showcases']);

    foreach($showcases as $showcase) {
        $movie->photos()->create([
            'src' => $showcase,
            'type' => 'showcase',
        ]);
    }

    return redirect('/');
}
```

Como observamos en el ejemplo de la imagen, el cual corresponde a la función de guardado de una nueva película guardada, consiste en 3 pasos:

- Primero se validan los datos, si es requerido, que tipo de dato ha de ser, y otras restricciones.
- A continuación se crean los nuevos objetos de las clases usando el método create del modelo, añadiendoles así a la base de datos.
- Redireccionar.