

CINESFERA



Carried out by:

- Aleksander Trujillo Prokhorenko
- Hugo Jiménez Muñoz

Index

1. *Resumen*
2. *Despliegue de la aplicación*
 - a. *Back-end*
 - b. *Front-end*
3. *Front-end - APP*
4. *Back-end - APP*
5. *Diseño de la base de datos*
6. *Bibliography*

1.Summary

It is a web application developed with Nextjs 14 which is a React framework, Laravel a Php framework, to deploy it we have used an environment created in Devilbox which is based on Dockers..

The application consists of a platform where you can see trailers and reviews in real time of movies and series that are released worldwide, for this we have used an external api rest TMDB, for the management of users and store their data we have developed our own api rest.

We have also implemented the use of cookies in the application to manage the use of JWT token for login and check if the user is logged in or not.

For the development of the application as a version controller we have used Bibucket, a working environment of the Jira ecosystem, specialised in the creation of projects, based on Git.

.

2. Deployment


a. Back-end

In order to deploy the application you need to:

- Docker desktop (Windows)
- Ubuntu (Microsoft Store)
- Visual Studio Code
 - WSL plugin

To deploy the back once ubuntu is installed, we enter through terminal, and follow the installation steps of [Devilbox](#). Once Devilbox is up and running by logging into the administration panel, we have to allow Docker to connect to Ubuntu via WSL, which we do from the [configuración](#) of Docker itself.

Once the back-end environment is ready, we have to configure our Visual Studio Code to be able to connect to our project and modify the code and execute the necessary commands for Laravel to work correctly.

Once inside visual in the plugins section we have to look for WSL and install it, once installed, an icon will appear in the left bar like this one: , when we click on it, we will be able to connect to our Ubuntu. Then copy this link and clone the repository (it has to be inside /devilbox/data/www/project).

You can also deploy it with laragon or laravel itself with XAMP, no need for devilbox.

Now we have to go to the root path of Devilbox which is: /devilbox and execute the command `./shell.sh` in the Visual Studio terminal, this will open our docker php container and we will have to go to our devilbox project, once in the project we will execute the following commands:

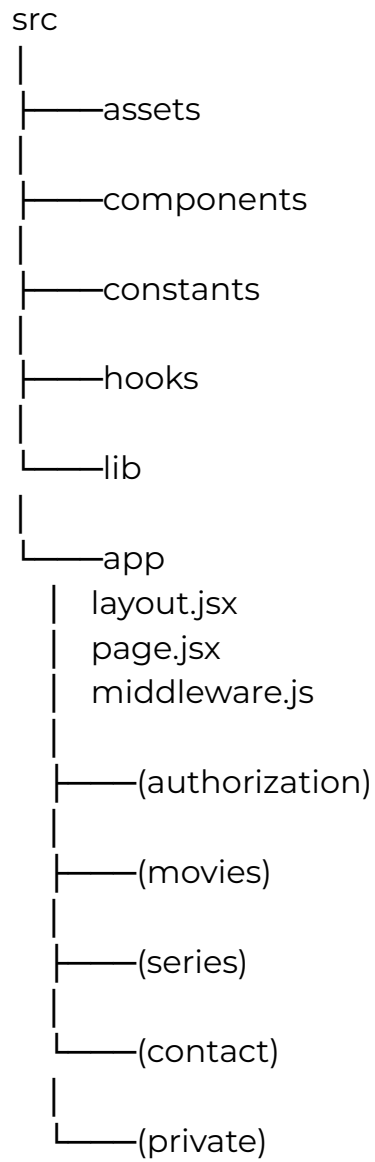
3. composer install
4. php artisan key:generate
5. migrate:
 - a. php artisan migrate
--path=/database/migrations/2024_04_06_111709_create_roles_table.php
 - b. php artisan migrate

c. Front-end

To deploy the front end we need to clone the repository in our windows to the location we want. Once cloned we need to run the following commands:

1. `pnpm i`
2. `pnpm dev`

3.Front-end - APP



1. Public

In this folder we will upload all the static content files, that is to say, all the images, svgs that we have and that we want to be accessible from any side of the project will be here.

2. Src

This is the root folder of the project, inside it we have different folders like:

App which in our application would be the url ('/') everything we put inside app can be routable:

Assets we can find imports of the fonts and their weights as well as the styles separated by modules.



```
1  /**
2   *
3   *  TIPOGRAFIAS BASE DE LA APP
4   *
5   */
6  export const montserrat = Montserrat({
7    subsets: ['latin'],
8    weight: ['200'],
9    style: ['normal']
10 })
11
12 export const kanit = Kanit({
13   subsets: ['latin'],
14   weight: ['200'],
15   style: ['normal']
16 })
17
```

Components in this folder we can find several folders containing the different components used in the application, such as image sliders, movie/series carousel and authors, this is done to obtain a higher code reuse that allows a more stable scaling.

```
1 'use client'
2
3 import { template } from "../Template"
4
5 const Button = ({ label, url = '', isLoading, type='submit', styleType = 'contained', templateType = 'basic', onClick = () => {}, id }) => {
6
7   const LabelTemplate = template[templateType]
8   return (
9     <button type={type} disabled={isLoading} className={styleType} onClick={onClick} id={id}>
10       {isLoading && <span className="loader"></span> }
11       <LabelTemplate label={label} url={url} />
12     </button>
13   )
14 }
15
16 export default Button
```

Constants in this folder we have a file where we find different constants for the operation of the application such as the access token to the movies/series api, the url of our server to which we will make the requests, and the url of the movies/series server api..

```
1 /**
2  *   API URL → BACK-END LARAVEL
3  *
4  *   En caso de cambiar la url del back debes modificar esta URL, tiene que acabar en /api para que funcionen
5  *   todos los end points de la App.
6  *
7  */
8 export const API_URL = "http://tfg-cinesfera-back.dvl.to/api"
9
10
11 /**
12  *   API EXTERNA URL → TMDB
13  *
14  *   1. Url de la api de terceros
15  *   2. Token de acceso a la api.
16  *
17  */
18 export const API_URL_TMDB = "https://api.themoviedb.org/3"
19 export const TOKEN_TMDB = ""
```

Hooks in this folder we abstract all the application logic that will be reused in more than one part of the application, such as setting the context with Zustand, the react-hook-form validation scheme.

Lib in this folder we will find a lot of functionality that we will use mainly in server components, an example would be all the requests that we have, we also find the router of the application that is what makes the nav header to be generated to be able to navigate in the application.

Middleware.js is a general scope file in the application, this means that it is at the same level as the app/ folder, therefore every time a request is made

in the application either to browse or to send data to any of the two apis these requests will be processed first by the middleware file which cannot be given very heavy operations because it can compromise the optimisation of the application.

```
1 export function middleware(request) {
2
3   if(request.nextUrl.pathname === '/profile') {
4     if(hasCookie('user_token', { cookies })){
5       return NextResponse.next();
6     }else{
7       return NextResponse.redirect(new URL('/login', request.url));
8     }
9   }
10
11 }
12
13 // See "Matching Paths" below to learn more
14 export const config = {
15   matcher: [
16     /*
17      * Match all request paths except for the ones starting with:
18      * - api (API routes)
19      * - _next/static (static files)
20      * - _next/image (image optimization files)
21      * - favicon.ico (favicon file)
22      */
23     '/((?!api|_next/static|_next/image|favicon.ico).*)',
24     '/profile'
25   ],
26 }
```

3. App

In this folder you will find all the files that will make the application routable and the different pages of the application..

layout.jsx this file is in charge of giving the general structure to the whole application and that all the pages will share, in this file for example components like the footer, header that are static things that will have all the pages called.

page.jsx this file will be the dynamic content of our application root page, inside each page.jsx is the content that will change throughout the application.

Inside this folder we have several folders whose names are in brackets, this

is the following syntax to be able to group zones without being able to be routable this folder, this is used to give more syntax to the code and a better readability.

(Authorization)

Within this folder we have to distinguish between the registration page and the login page, each one is responsible for a different thing in the app, as its name suggests, in the registration page we will see a form in which we can register a user in the application and in the login page a form to log in to the application and access private areas of it.

(Contact)

In this folder you will find the contact page of the application where you will find a map, contact information and a form where you can send me directly what you need.

(Movies)

In this folder we can find two parts, one folder movies and another folder [id], the folder with the square brackets are special folders that allow us to create page with data input by parameters, this has been used to show the single of a particular movie and for this we send the id of the movie by parameter.

The other folder 'Movies' is used to route the movies, it contains the file page.jsx in which we can display the content of this without any problem.

(Series)

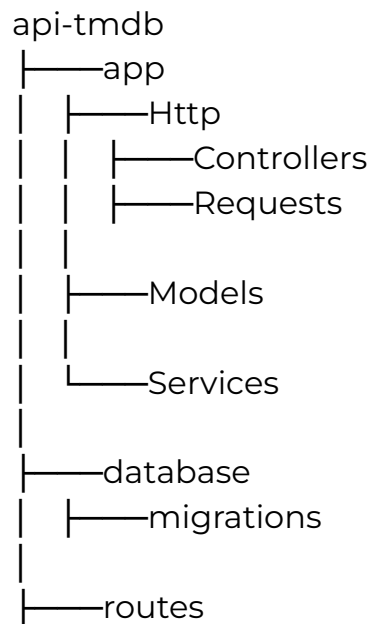
In this folder we can find two parts, one folder series and another folder [id], the folder with the brackets are special folders that allow us to create page with data entry by parameters, this has been used to show the single of a particular series and for this we send the id of the movie by parameter.

The other folder Series is used to route the series, it contains the file page.jsx in which we can visualise the content of this one without any problem.

(Privada)

In this part can only be accessed once the user has logged into the application and there is a cookie stored in the user's browser, previously mentioned that the middleware is the one that receives all requests, as in this same routes that do not want to be accessible to the public are protected by the macher and with a function and through the request received is checked everything necessary to know if you can access this route or not.

4.Back-end - APP



In the App folder we have the logic of our application in it are the controllers, models, custom request and services. Following one of the SOLIDS principles, we have made the controllers have a single responsibility, so we have created the services to abstract the application logic from the controllers that act as intermediaries.

Inside the **Controllers** folder we have organised it in subfolders to divide this and have it more organised:

Auth in this folder we have a controller in charge of registering, logging in and logging out the application, for this it relies on a service that executes the relevant methods for the creation of JWT, in addition to using the methods of the Auth class of laravel to check the login.

Media being an application that manages both movies and series we have created a single controller that manages both, for it has two main functions that are store and destroy to save a 'like' and to make 'dislike' in a media, we treat movies and series as media.

Then we have another function that is in charge of listing which is called index, this receives a series of parameters from a request and gives them to the service so that it returns a list of paged and ordered socks.

```
1 public function __construct(private MediaService $media) {}
2 /**
3  * Display a listing of the resource.
4  */
5 public function index(Request $request)
6 {
7     $params = $request->all();
8     $listOfmedia = $this->media->index($params);
9
10    return response()->json(['message' => 'List of media', 'data' => $listOfmedia, 'status' => 200]);
11 }
```

Watch just as we have with media to like a measure, we have the same to see this later or not depending on what the user indicates, for this we have designed a controller in the same way as media following the same pattern of functions that are responsible for saving a media to see later or not.

Within **middleware**, we have a file in which we control the access to the application, our application being via API we cannot control as such the route to which it accesses the endpoint, but what we can do is check the header of the request.

In the header usually go the cookies and login token, because when a user makes a request to our api we check this token in case you do not have it we make a return of a JSON response to display it on the front, we do this to modify the operation of laravel default that brings.

```
1 /**
2  * Get the path the user should be redirected to when they are not authenticated.
3  *
4  * @param  \Illuminate\Http\Request  $request
5  * @return string|null
6  */
7 protected function redirectTo($request)
8 {
9     if (! $request->expectsJson()) {
10         return response()->json(['error' => 'Unauthenticated'], 401);
11     }
12 }
13 }
```

Within **Request**, we have a folder structure organised in the same way as for the controllers, in which we define our validation rules for the data that

comes to us through the Api, to add another layer of security to our application.

In all the custom requests we have designed we have overwritten the default laravel methods to give them more customisation. To do this we have modified the **message()**, **failedValidation()**.

These functions are responsible for displaying error messages every time a validation fails, also in the case of failure instead of allowing laravel to redirect the user as it does by default we modify its functionality to return a message in JSON format so that our front end can interpret it and show it to the user..

```
1 /**
2  * Function with which we modify the default Laravel error message to a custom message
3  */
4 public function message(){
5     return [
6         'email.unique'      => 'Error: The email has already been taken.',
7         'email.email'      => 'Error: Please enter a valid email address.',
8         'password.required' => 'Error: You need to enter a password'
9     ];
10 }
```

```
1 /**
2  * Function with which we handle the one that shows the error messages after the validation fails, this overwrites
3  * to the original method that redirects to the previous route.
4  */
5 protected function failedValidation(Validator $validator){
6     throw new HttpResponseException(response()->json($validator->errors(), 422));
7 }
```

In the **Models** folder we have all the models of our tables, it is the place where apart from the migrations we define our relations so that laravel eloquent can interpret it and allow us to relate things, also the models serve to indicate which fields can be modified and which cannot, also in the case of sensitive data we can indicate that they are hidden in the network as much as possible to avoid that they are visible.

A clear example is the user model containing sensitive data such as passwords:

```
1 class User extends Authenticatable
2 {
3     use HasFactory, Notifiable, HasApiTokens;
4
5     /**
6      * The attributes that are mass assignable.
7      *
8      * @var array<int, string>
9      */
10    protected $fillable = [
11        'display_name',
12        'name',
13        'email',
14        'password',
15        'remember_token',
16        'role_id'
17    ];
18
19    /**
20     * The attributes that should be hidden for serialization.
21     *
22     * @var array<int, string>
23     */
24    protected $hidden = [
25        'password',
26    ];
```

In these models what we are always going to find is the variable `$fillable`, which is the one that allows these fields to be modified and `$hidden`, which is the one that hides some fields.

Another part of the models that is very important is the definition of the relations between them to be able to attach the properties and fill the data with consistency in the database:

```

1  /**
2      *      -----
3      *
4      *      RELATIONSHIPS WITH MODELS
5      *
6      *      -----
7      */
8
9      public function roles () : belongsTo {
10         return $this->belongsTo(Rol::class, 'role_id');
11     }

```

With these relations we indicate what type of relation it is if 1 to N, N to N, depending on the case we will do one or the other and with the table that is the foreign key that we have in this table to relate it.

In this example the relationship is with the roles table, as our users have different types of roles.

In the migrations folder, we have all the files that are executed and create the table structure that we define for the correct functioning of the application, for each table that we create we must have a model to be able to modify its fields and attributes depending on the actions that are done.

Este es un ejemplo de la migración de usuarios:

```

1  Schema::create('users', function (Blueprint $table) {
2      $table->id();
3      $table->string('name');
4      $table->string('display_name')->nullable();
5      $table->string('email')->unique();
6      $table->foreignId('role_id')->constrained('roles');
7      $table->timestamp('email_verified_at')->nullable();
8      $table->string('password');
9      $table->rememberToken();
10     $table->timestamps();
11 });
12

```


In this migration we define the primary key that by default is the property **->id()**, we also define the foreign keys that if we follow the laravel convention with the names we can use constrained('table_name') and the name of the table is optional.

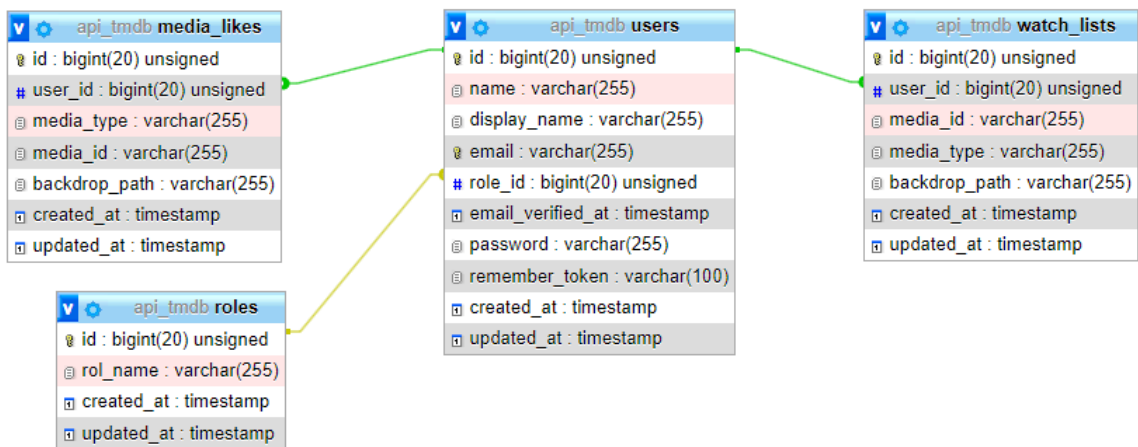
The routes folder is the heart of our application because thanks to it we can make visible all the endpoints of the application and tell it which controller manages them as well as protect the routes with the middleware to avoid that they can make requests without having authorization token.

For the endpoints to work these routes have to be defined in the api.php file so that laravel-api detects them, otherwise they would be web routes that would be entered in case we are using laravel with views.

```
1  /**
2  * Group of routes that manage the Login/Register system
3  */
4  Route::prefix('auth')->group(function () {
5      Route::post('/register', [AuthController::class, 'register']);
6      Route::post('/login', [AuthController::class, 'login']);
7      Route::middleware('auth:sanctum')->group(function () {
8          Route::get('/logout', [AuthController::class, 'logout']);
9      });
10 });
11
12 /**
13 * Todas las rutas protegidas ahí que añadir las dentro del middleware.
14 */
15 Route::middleware('auth:sanctum')->group(function(){
16
17     /** ----- MEDIA ROUTE ----- */
18     Route::get('media', [MediaLikeController::class, 'index']);
19     Route::get('media/{media_id}/user/{user_id}', [MediaLikeController::class, 'show']);
20     Route::post('media', [MediaLikeController::class, 'store']);
21     Route::delete('media', [MediaLikeController::class, 'destroy']);
22
23     /** ----- WATCH LIST ROUTE ----- */
24     Route::get('watch', [WatchListController::class, 'index']);
25     Route::get('watch/{media_id}/user/{user_id}', [WatchListController::class, 'show']);
26     Route::post('watch', [WatchListController::class, 'store']);
27     Route::delete('watch', [WatchListController::class, 'destroy']);
28
29 });
```

The routes that manage the login system have been prefixed to differentiate them from the normal routes and to have a better syntax in our api.

5. Diseño de la base de datos



We have designed the database so that each user is unique in the application in order to maintain data integrity and so that it cannot be mixed with another user's data.

All the tables are related to the `user_id` to know which media was liked or not, and the same happens with watchlist.

In the case of the roles, as the user can only have one role as Registrar or Unregister, we don't need a pivot table.

6. Bibliografía

- Laravel. (s.f.). Laravel - The PHP Framework for Web Artisans. <https://laravel.com/>
- Docs. (s.f.). Next.js. <https://nextjs.org/docs>
 - Devilbox.org. (s.f.). <http://devilbox.org/>
 - Docker: Accelerated Container Application Development. (2024, 20 mayo). Docker. <https://www.docker.com/>
 - The Movie Database. (s.f.). Getting started. The Movie Database (TMDB) <https://developer.themoviedb.org/reference/intro/getting-started>
 - Zanic, A. (s. f.). cookies-next: Getting, setting and removing cookies on both client and server with next.js. GitHub. <https://github.com/andreizanic/cookies-next>