MARC PERAL CAJIDOS

sa palomera 2023-2024

OVO

DAW Final project

Content

[Introduction 3](#_Toc167912990)

[Functional Requirements 3](#_Toc167912991)

[Login and Register 4](#_Toc167912992)

[Local Disk Libraries 4](#_Toc167912993)

[Movie Metadata from external API 4](#_Toc167912994)

[Custom Video Player 4](#_Toc167912995)

[Multiple Database Support 4](#_Toc167912996)

[Admin Panel 4](#_Toc167912997)

[SyncPlay 4](#_Toc167912998)

[Technical Requirements 5](#_Toc167912999)

[Developer 5](#_Toc167913000)

[User 5](#_Toc167913001)

[Diagrams 6](#_Toc167913002)

[Use case 6](#_Toc167913003)

[Anonymous 6](#_Toc167913004)

[Admin 6](#_Toc167913005)

[User (Authenticated) 6](#_Toc167913006)

[Activity 8](#_Toc167913007)

[Visitor 8](#_Toc167913008)

[User 9](#_Toc167913009)

[Admin 11](#_Toc167913010)

[Interfaces 13](#_Toc167913011)

[Wireframe 13](#_Toc167913012)

[Mockup 13](#_Toc167913013)

[Classes 15](#_Toc167913014)

[Database 17](#_Toc167913015)

[Implementation 18](#_Toc167913016)

[Deployment 18](#_Toc167913017)

[Improvement proposals 18](#_Toc167913018)

[Conclusions 18](#_Toc167913019)

[Webography 18](#_Toc167913020)

[Documentation and help 18](#_Toc167913021)

[Tooling 18](#_Toc167913022)

[Resources 18](#_Toc167913023)

# Introduction

Back in the days where there was no Netflix or any other video streaming, people usually had their own media library. Containing boxes of movies or seasons of a TV show. Today, although the vast majority uses video streaming for a regular basis, some still prefer to own their content, so they still have a physical library.

This project aims to that people that still own their content, and have a physical based library. We can help and improve their media consumption experience by making possible to consume it at any time anywhere as long as there is an internet connection.

As an extra, we also provide features like checking content as “watched” or “favorite”, find movies or shows by an actor or director, having a synced video player to watch your content with your friends and family, and more to come.

# Functional Requirements

The following roadmap specifies the technical and functional requirements for the site. This includes both frontend and backend functionality.

* Responsive frontend (Mobile, Tablet, Desktop)
* Login and Register
* Local Disk Libraries
* Movie Metadata from external API
* Custom Video Player
* Multiple Database Support
* Admin Panel
* Auto installer (web based setup process)
* DLNA (send video to devices)
* SyncPlay

### Login and Register

#### Login

The user has the ability to login using existing credentials (username and password).

#### Register

Users who do not have an account can register using a username and password. Currently any user registration will be accepted, but it is planned to only activate accounts by the admin of the instance.

### Local Disk Libraries

Admin can use admin panel to add searching paths to index movies and make them accessible. For example:

An instance of the server has access to /mnt/movies, but the admin only wants to index some folders. The admin can configure indexing paths to use

* /mnt/movies/christmas
* /mnt/movies/halloween

In that case, only those two folders would be indexed and the system would treat as a single one.

### Movie Metadata from external API

During a content directory scan, it will try to match real content in order to fetch metadata.

Currently the only metadata provider is The Movie Database (TMDB)

### Custom Video Player

A custom video player that plays the content from the site.

### Multiple Database Support

The service has the compatibility with multiple databases. This can be but not only:

* SQLite 3
* MySQL
* PostgreSQL

This is possible due to the GORM package used in the server, that has support for many databases, as stated [here](https://gorm.io/docs/connecting_to_the_database.html).

### Admin Panel

Here a user with privileges, could add or modify libraries.

### SyncPlay

This will enable the custom player to be synced to watch content with other users within the site. For example, SyncPlay on a movie, will sync the video player of those who are in group (one user pausing will pause everyone's player, same for scrubbing, etc...).

Any user can create, join or leave a group.

# Technical Requirements

## Developer

* Backend
  + GO 1.22
    - [Templ](https://templ.guide/quick-start/installation) (Template generator CLI tool)
  + Database (Supported by GORM, could be SQLite, MySQL, PostgreSQL)
  + Sass
  + Node.js
  + FFMpeg installed in PATH
  + Air (Optional, reloads runtime on file change)
* Frontend
  + JQuery
  + Google Icons
  + Sweetalert2
  + Isotope
  + ImagesLoaded

## User

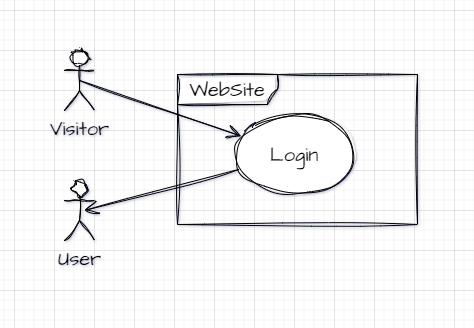
* Binary file (GO program compiled containing the backend and the frontend)
* Database (Supported by GORM, could be SQLite, MySQL, PostgreSQL)
* FFMpeg installed in PATH

# Diagrams

## Use case

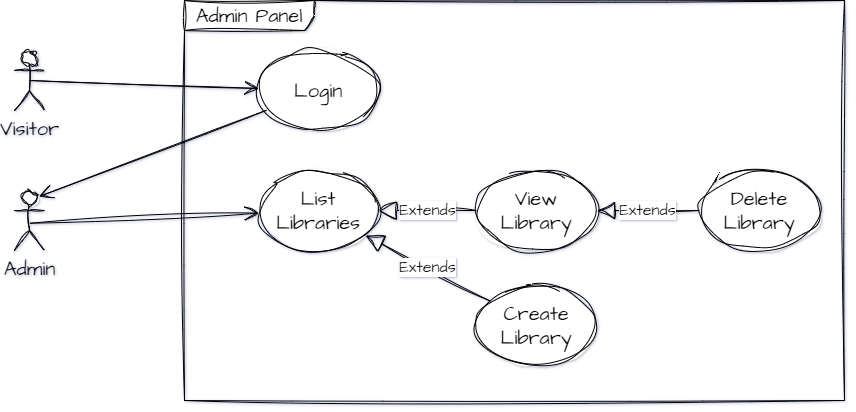
### Anonymous

Anonymous user can only login or register. In context of the app, the only notable action would be login, because that action moves his role into a user for further interactions.



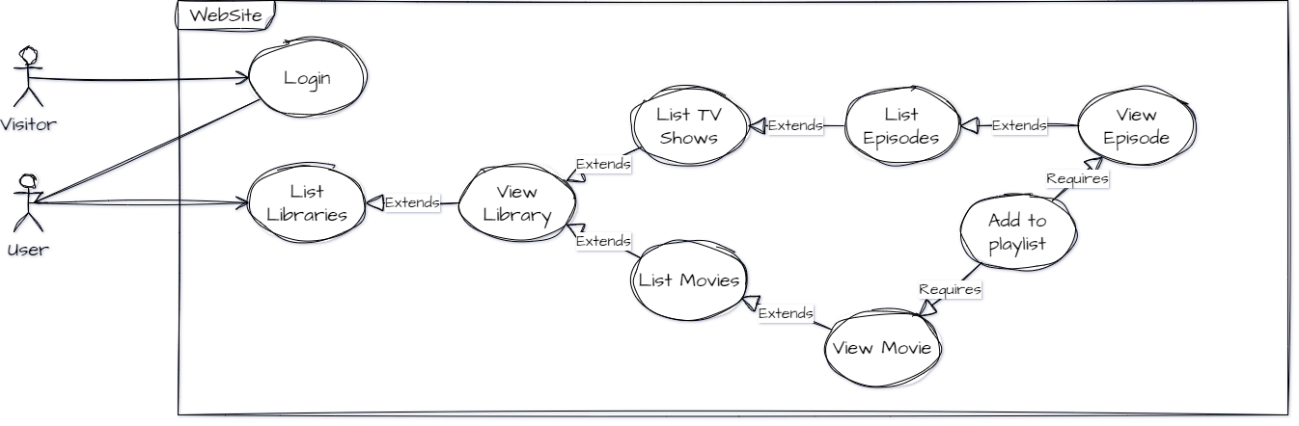
### Admin

An admin in the admin panel, can list, create, modify or delete libraries.



### User (Authenticated)

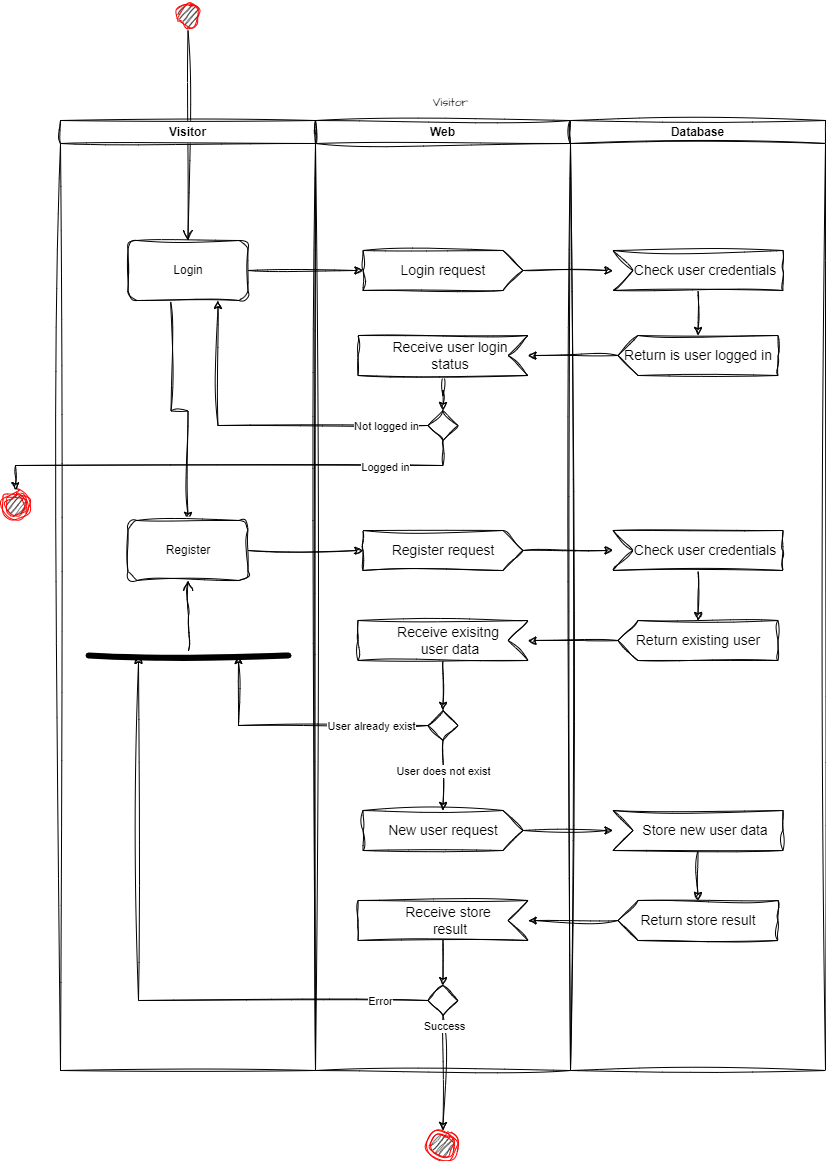
A user can list libraries and items in home page. Within that library, can view a item and play that item.



## Activity

### Visitor

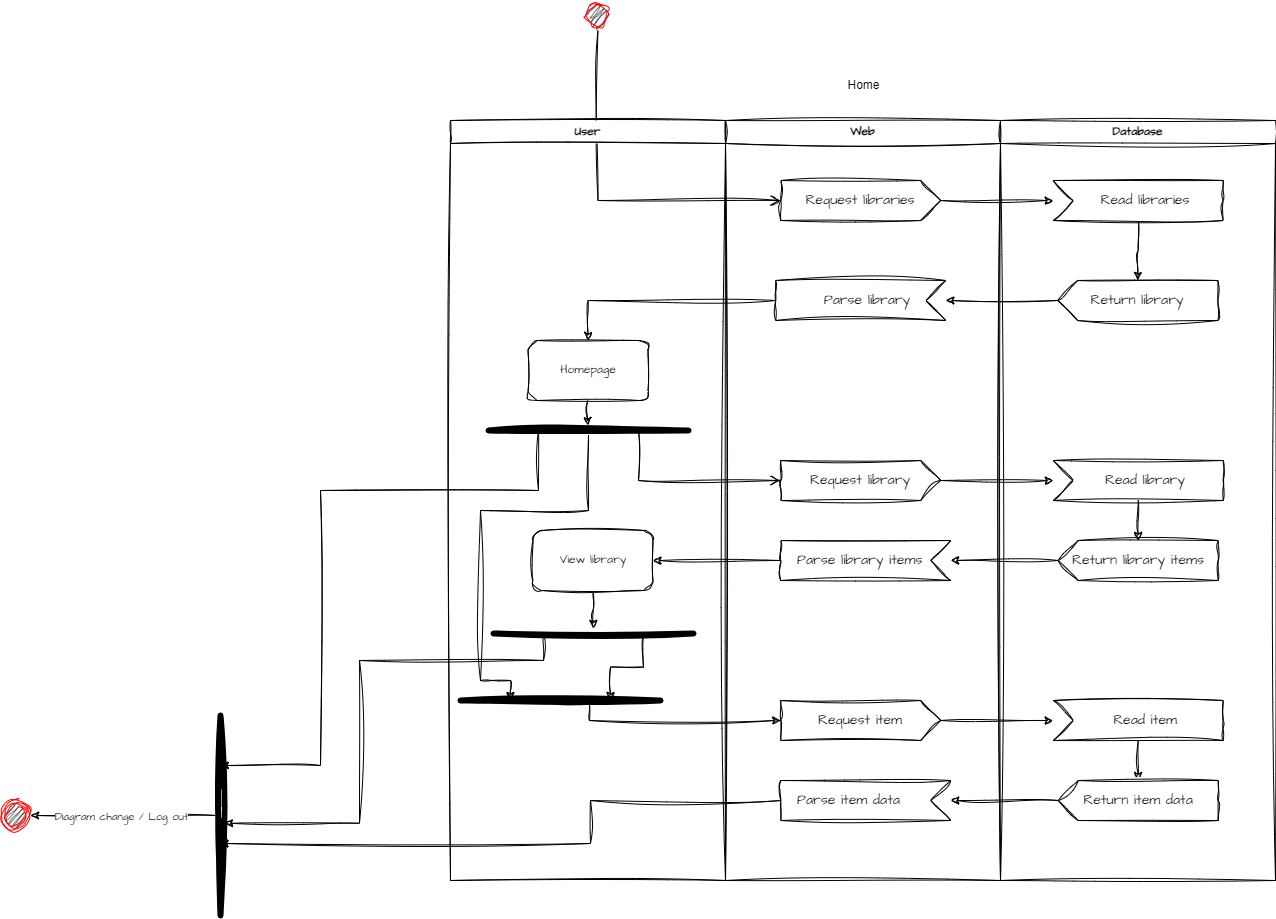
A visitor can only login or register.



### User

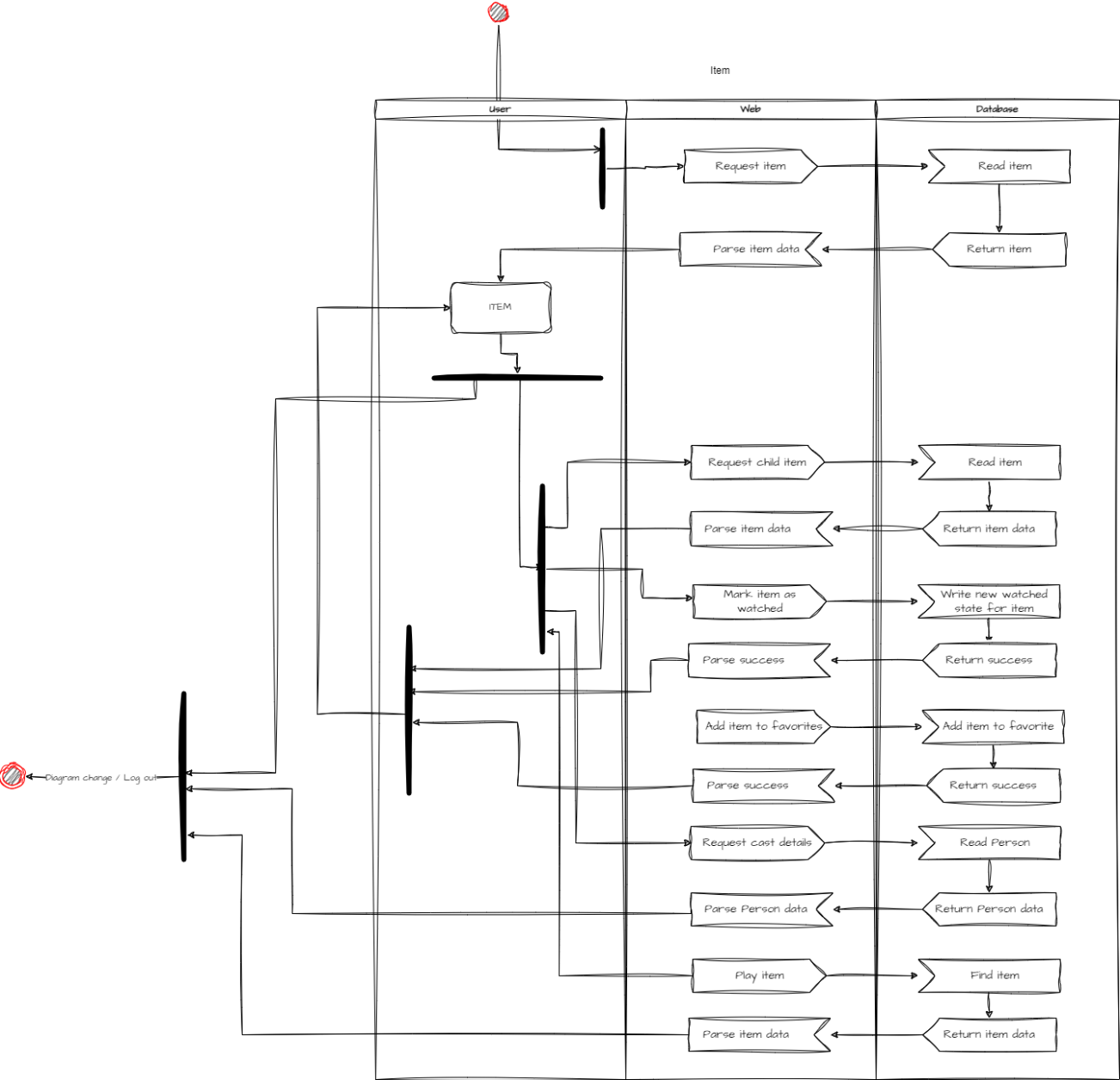
#### Home

User in home can visualize a list of libraries and latest library items.

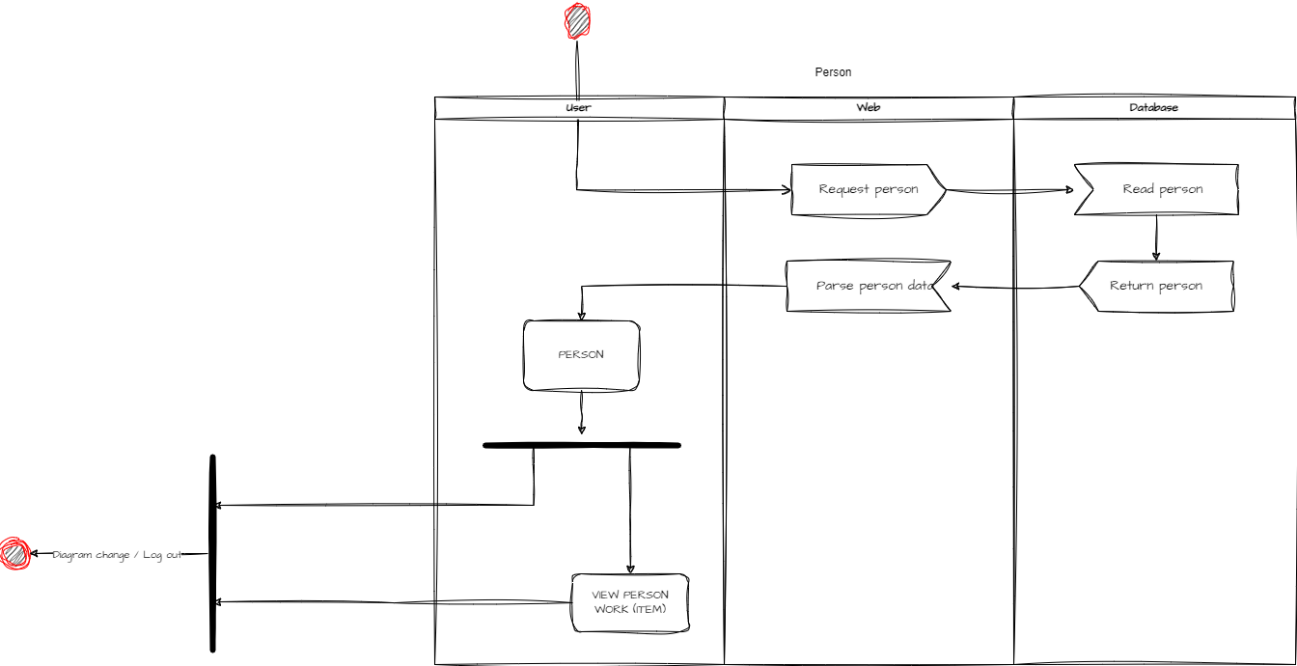


#### Item

Can view an item and the related metadata, mark item as favorite or watched, view a person (cast) profile or play the content.



#### Person

From an item, the person profile displays biography

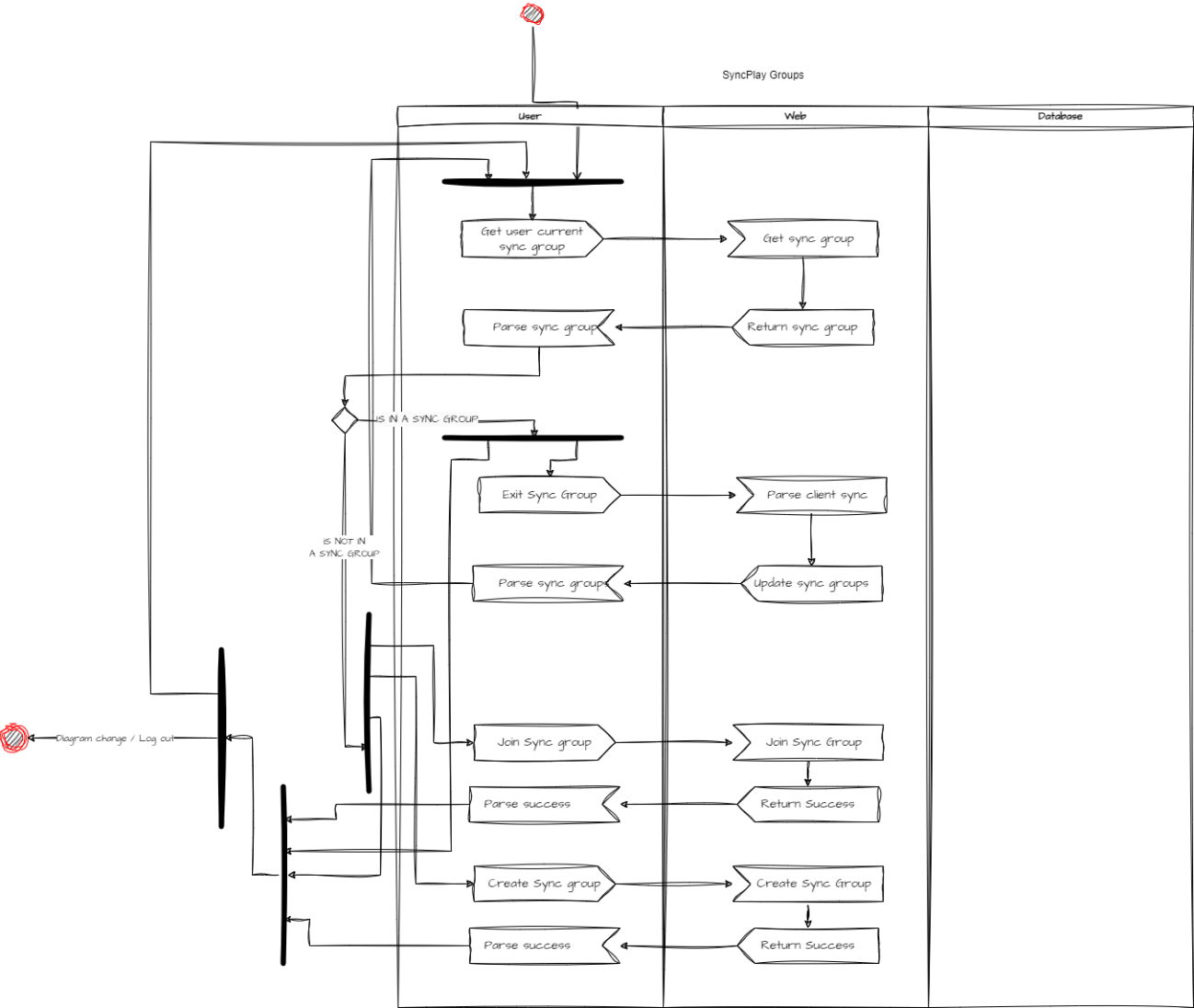
#### Playback

Contained within the video player, this activity is driven by the sync status. If the user is within a sync group, the flow of the activity is different.



#### Syncplay

Part of the Syncplay is related to this diagram. Here a user can create, join or leave a sync group anywhere on the site.

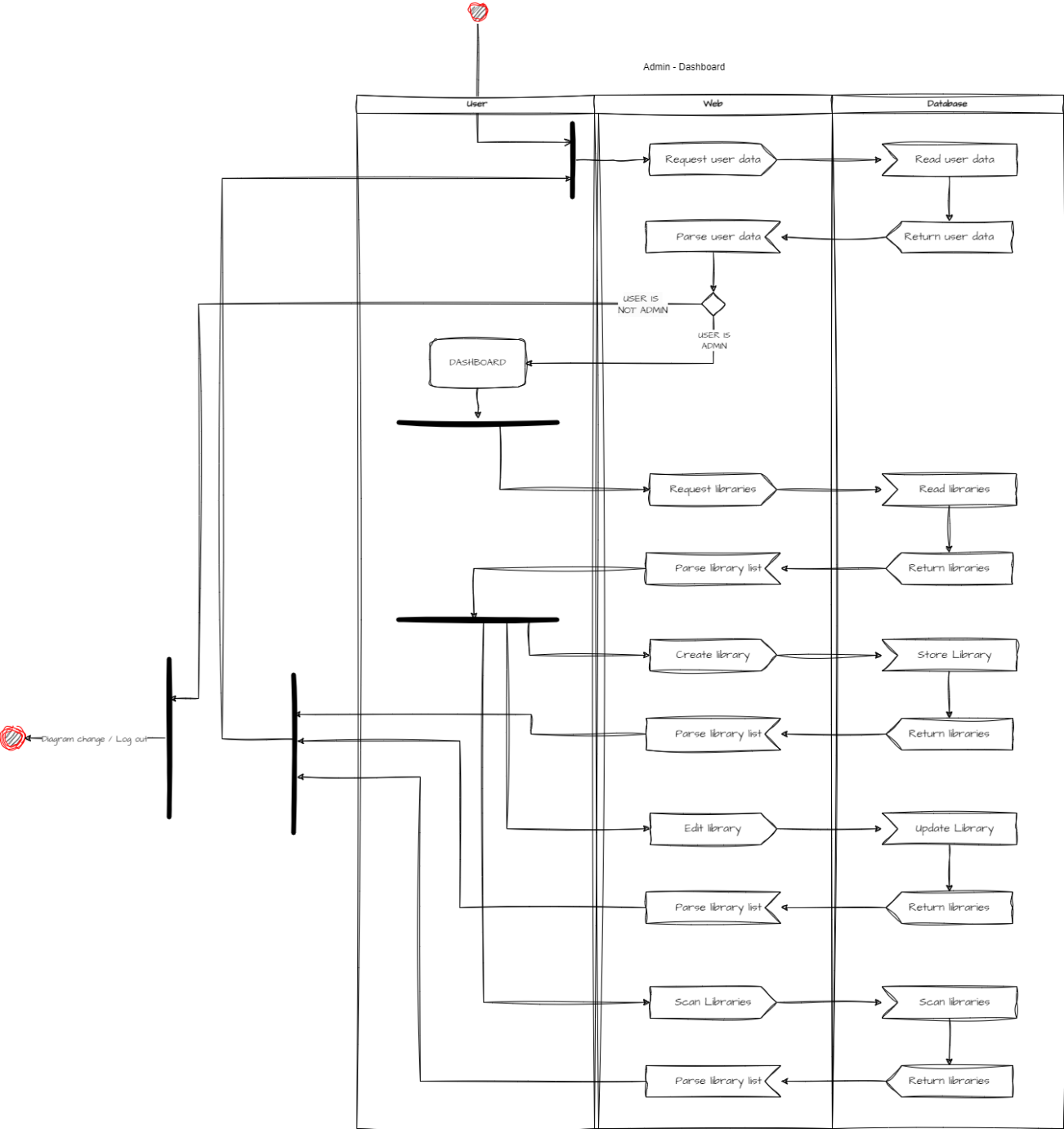


### Admin

The admin activity can be the following.

A user can access the admin panel if the role is admin.

Within the dashboard, the admin can request the current libraries, and modify or delete existing ones. Can also create new ones. Finally, the admin can scan the listed libraries to discover new, removed or modified content and update the database and metadata.



## Interfaces

### Wireframe

The project wireframe is available on a [Figma file](https://www.figma.com/file/aCbIxzpaxcp6AcK81DXtMA/OVO?type=design&node-id=1%3A6&mode=design&t=XCCFpA34dZMLbMfa-1)

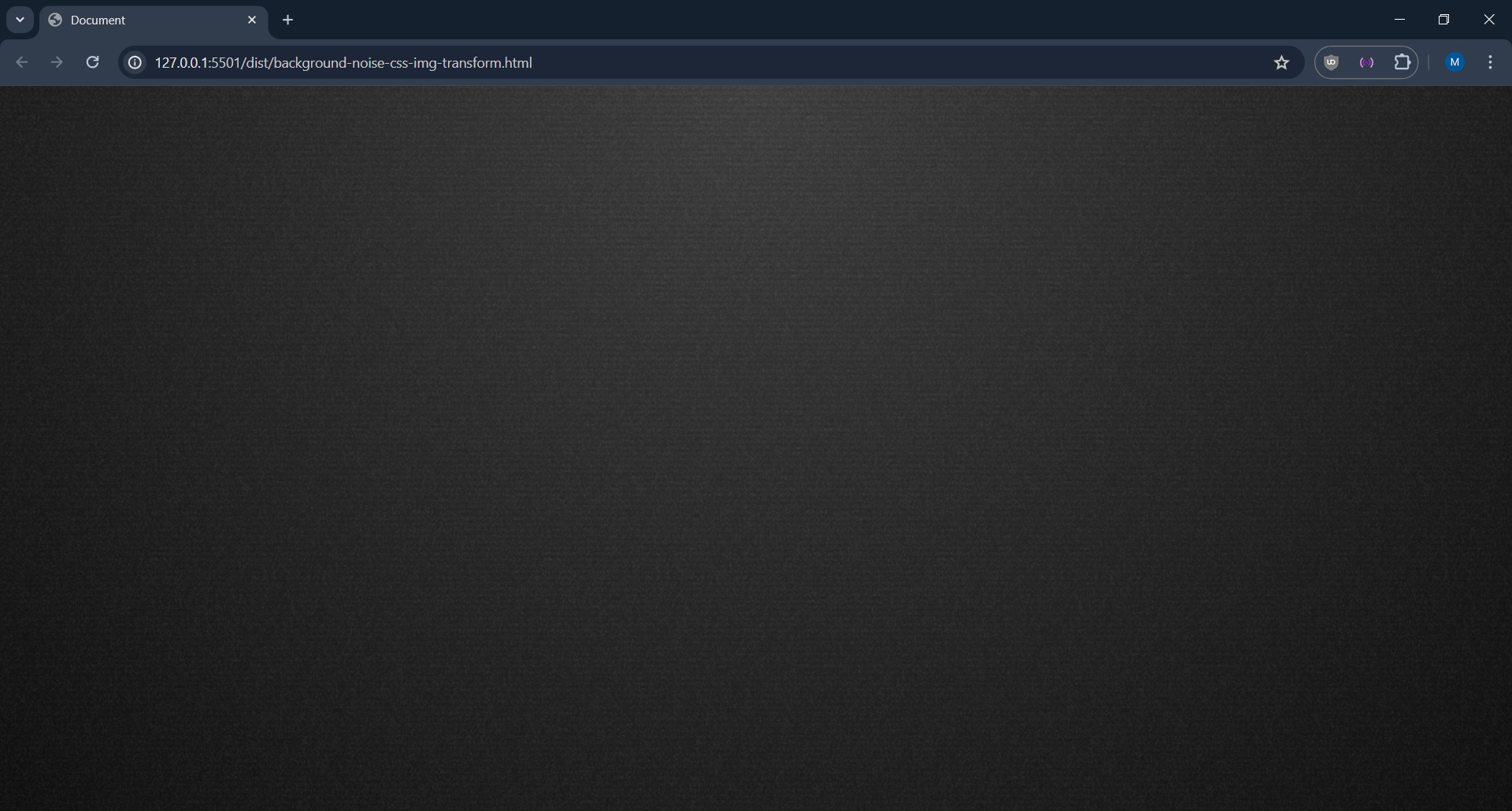
### Mockup

The mockup was a copy of the wireframe with some extras. But the real test was made in HTML&CSS&JS on a testing project in order to visualize better the effects instead of making assumptions on Figma.

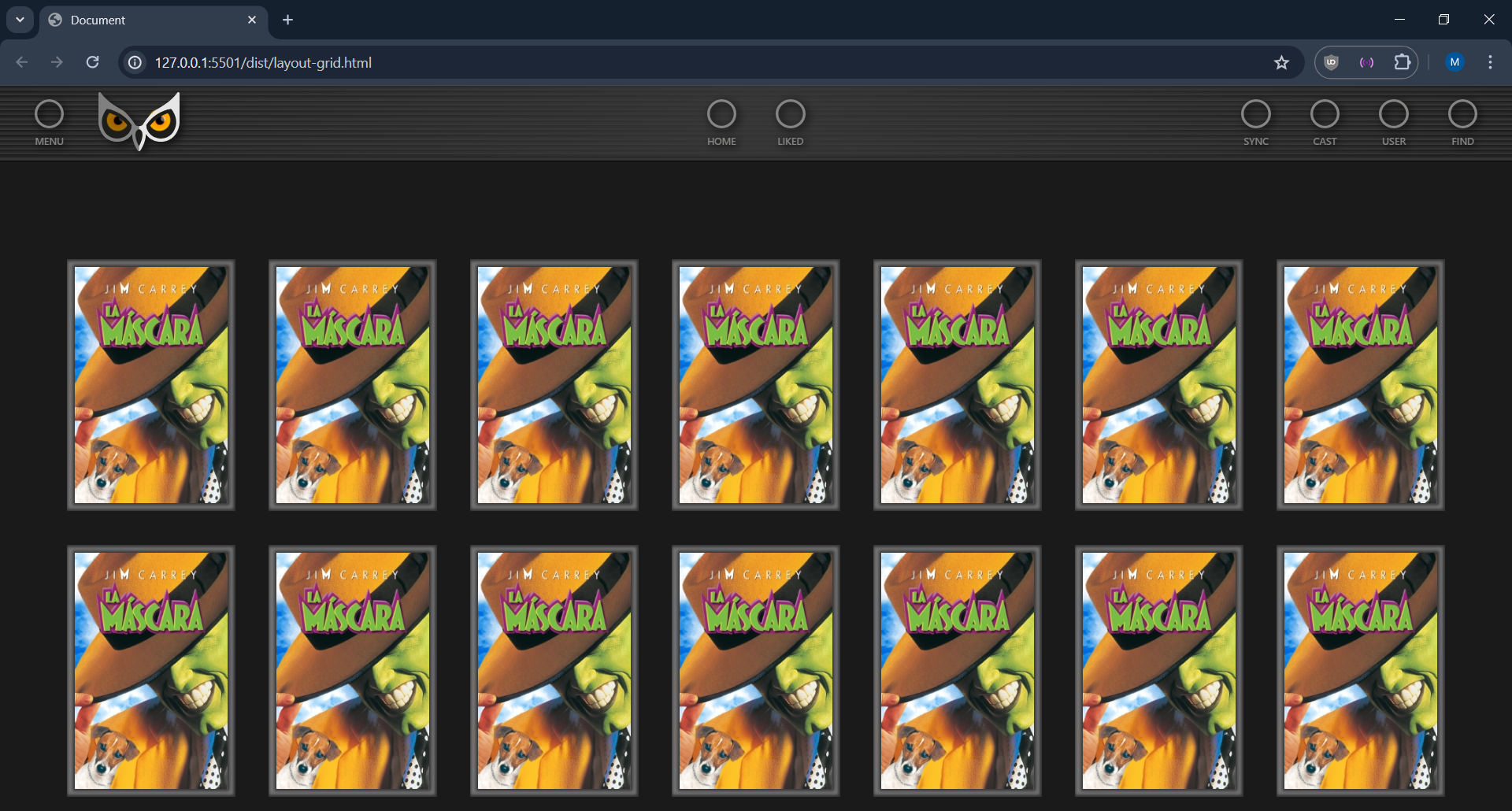
The Figma file can be found [here](https://www.figma.com/design/IfNPP6hXkPsHfpmMQlNgFZ/OVO-Mockup?node-id=0-1&t=7NnfQtD5Pj3zKpn2-1)

Here are some screenshots.

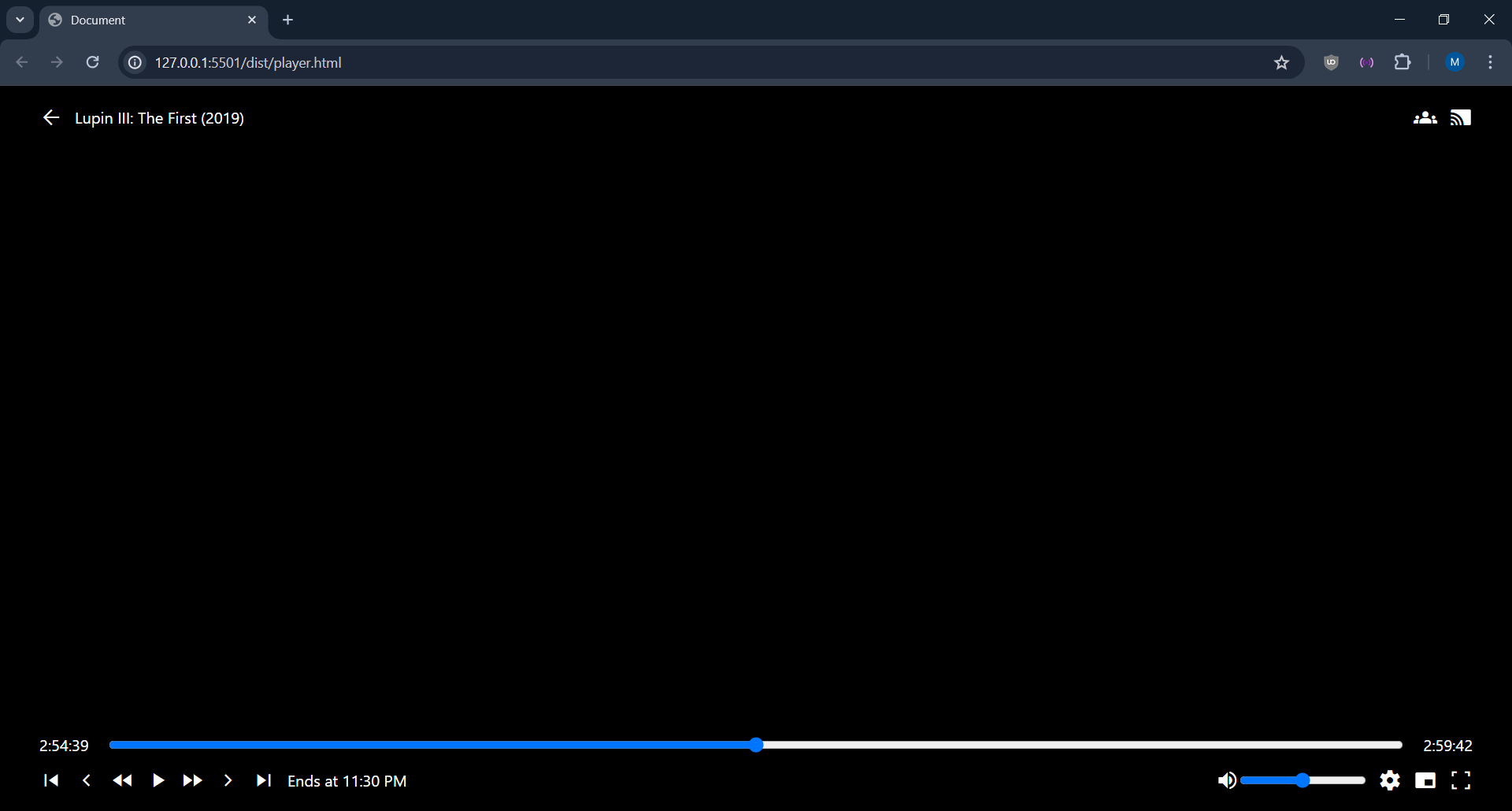
#### Background of the page plus noise plus the scanlines filter



#### Library Page



#### Custom Video player



## Classes

The class diagram is too big to be placed in this document. Can be found in the documentation [repository](https://github.com/mperalsapa/projecte-final-daw).

On this diagram, you can see each module and inside each “class”. The architecture of the project is based on MVC, where a module controls the view, another the model, and another the controller.

Over this, we have a main package that assembles all modules.

The architecture could be explained in the following definitions:

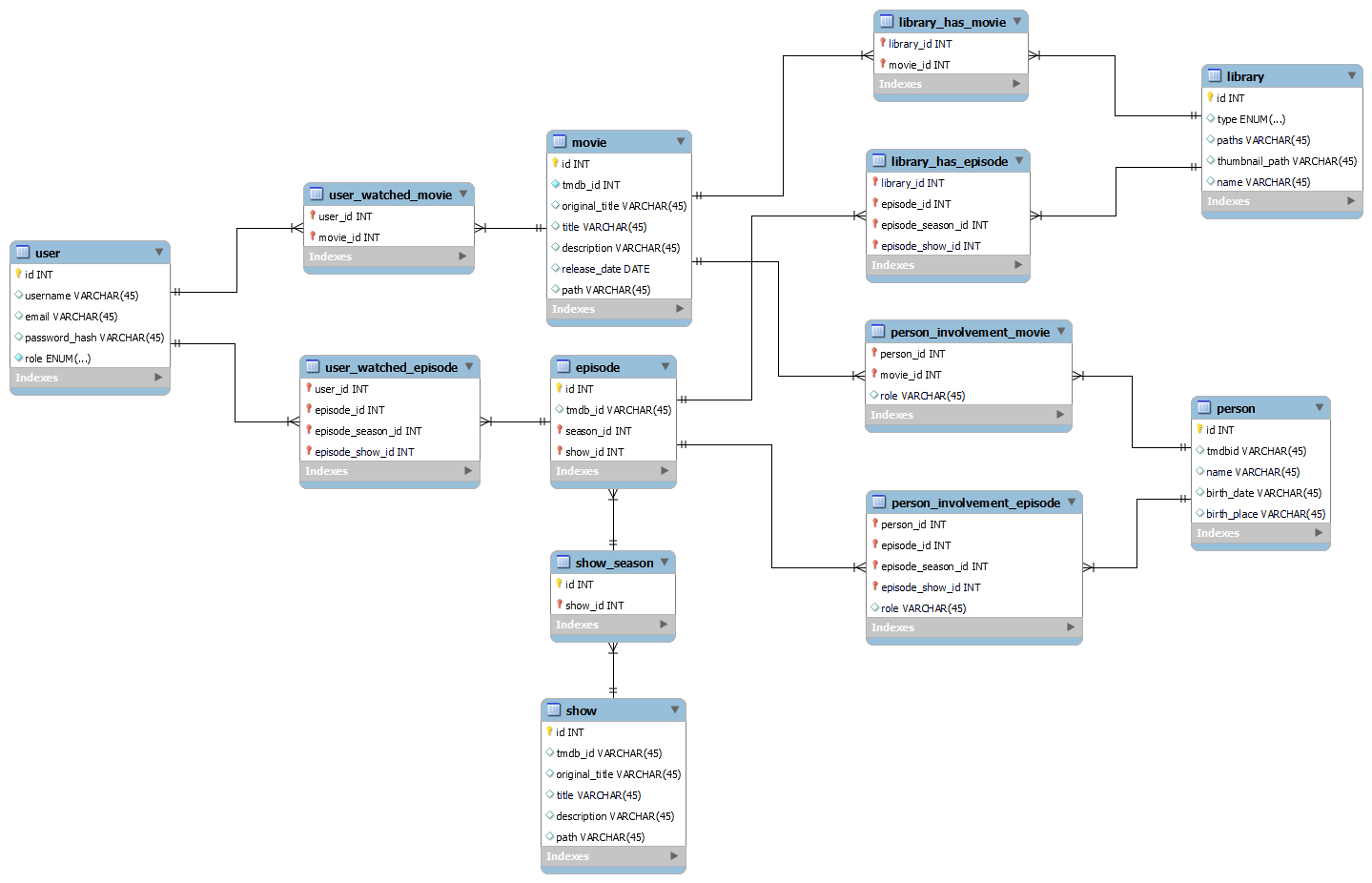
* **Main**: The main package that **assembles the project** using internal and external modules. This module is the one that starts the web server, registers the routes and calls de controller when new request come.
* **Config**: Used to define **server configurations**, like database type, basepath url, etc…
* **Router**: This package has the purpose of **registering routes** in a single place for later usage in any package. A route can be defined, for example “user profile” as “/user/profile”, and other packages (like main) can read that data.
* **Session**: Custom package in charge of **user sessions**. This package reads the session of a user (currently stored in a cookie, but can be anywhere) and makes the data within that session accessible and writable.
* **Database**: This package uses an external package to build a connection to a database. The type of database is provided by environment settings, previously loaded by the config package. It also makes accessible **the database connection** for other packages consumption.
* **Controller** & **ApiController**: Manager of valid network requests. This package contains **functions** that perform actions (like retrieve or update data) and / or **render pages**.
* **Model**: In charge of the whole app **data control**. This package is the one that reads from database and creates instances of objects like “User, Item, Library, etc…”
* **File**: Used to interact with the server filesystem and the bundled content.
* **Tmdb**: Custom built package to interact with an external package that provides an API for TMDB REST API.
* **Websocket**: Package that manages WebSocket upgrades and connections from a HTTP endpoint. This package is tightly built to interact with **Syncplay** package to provide player synchronization functionality, making possible to parse commands from clients, update sync state and updating clients of current state.
* **Syncplayer**: As stated previously, this package is responsible for the Syncplay functionality. It manages the state of sync and makes it accessible for other packages like **Websocket** or a controller (fe: ApiController).

Finally, we have the “Frontend” packages. Note that frontend is quoted, because we perform server side rendering.

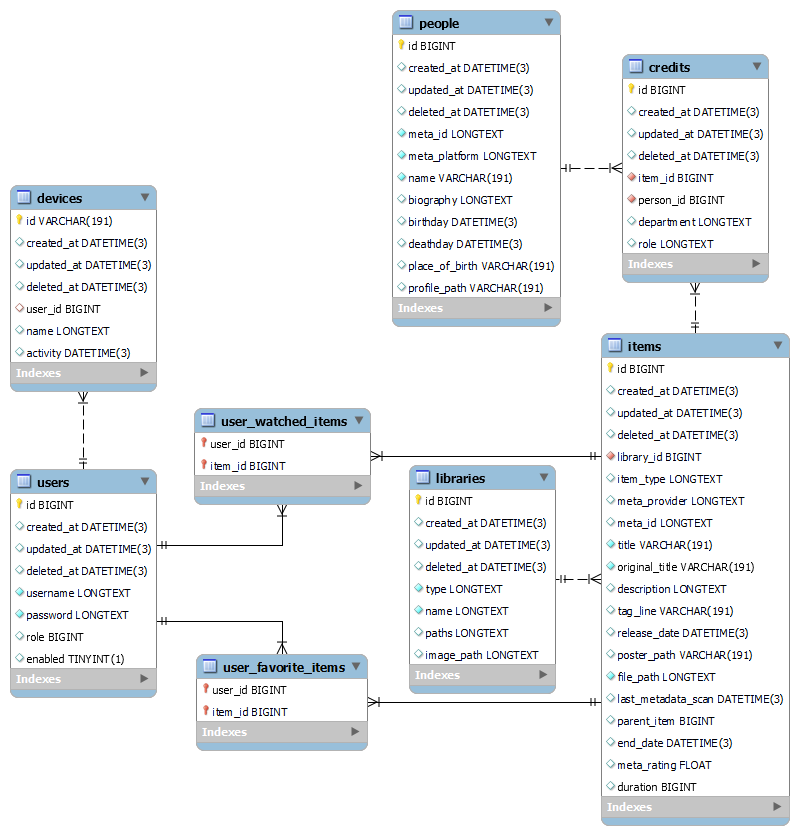
* **Component**: Package that provides HTML, CSS and JS **components for other packages** in order to repeat as little code as possible.
* **Shared**: This package provides what could be considered a component, but only contains the Page itself, for example, the **component that contains the Header** section of an HTML document.
* **Page**: Providing functions to render pages using component and shared functions to build pages. **This is the main** calling for a **render** when a request is processed in a controller.

## Database

This was the **initial** database schema. Here we can see that some entities like a Show or a Movie are in a separated context, with their own table and relations.



This is the **v3** of the database, the current state. Here we can see that a simplification work has been done. Now any entity like a movie, episode, show, are an “Item” entity. This moves the complexity from the database to the backend.



# Implementation

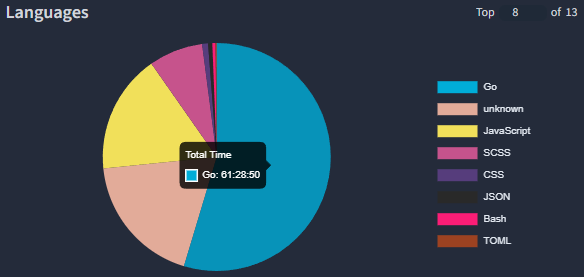
The implementation was a bit **shaky**. From the beginning, this project had in scope to be composited by **two source codes**. One would be the **frontend**, based in React, and the other would be the **backend** using the current stack, without the server side rendering.

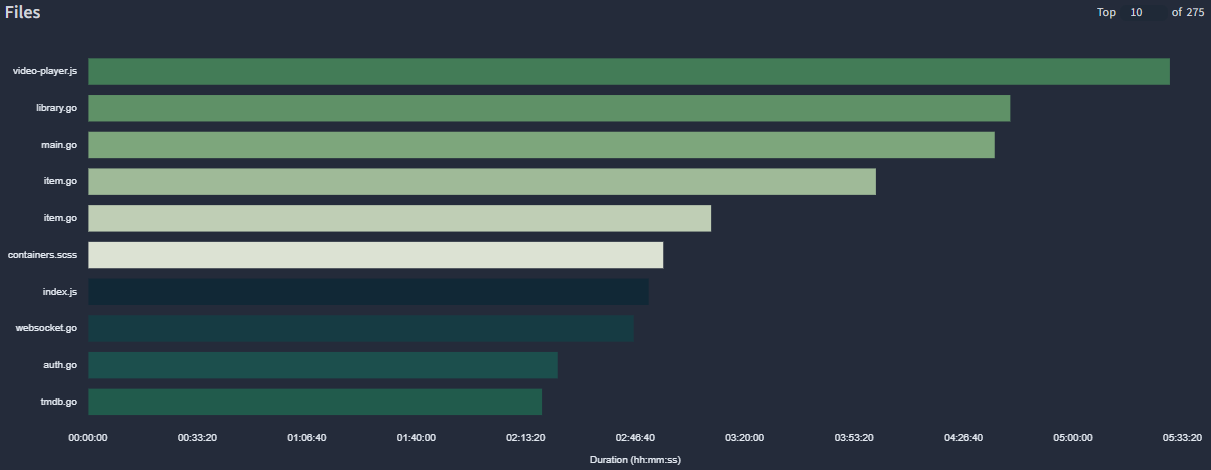
But we soon found that without prior training in **React**, it could be very difficult very fast. So, we opted for **server side** rendering.

Some of the initial requirements were discarded, not because a change of opinion or focus, but because we preferred to **quality over quantity**.

A **log of what has been done** can be found in the **main repo**, containing dates, explanations and screenshots.

As an extra, a time tracking has been used for the coding part, and there are tracking of spent time by file and date for the project.





NOTE: This only accounts for the writing and modifying text of the source code. This means that usually, **when working on a feature 1/3 or 1/2 of the time, there is no typing**. Also, **the “unknown” language, is Templ**, the used language for the template engine.

## Unit Testing

Although GO has a built-in testing package, there is no unit testing. There are a few **CMD programs** (located in **“cmd” folder** in code repo) that are used for **testing some** functionality but no close to what unit testing could be.

# Deployment

This section is very peculiar in our case. This is because **our project aims to be distributable for a normal user**, not only for developers. This makes the deployment really important, because **the easier the better for a user**.

Normally any web based service needs lots of files or technologies installed on the running server in order for the service to work. In our case, we opted for GO because we can simply **bundle the whole project in a binary file** and **distribute it**.

Go also makes multi OS builds easy, aiming Windows and Linux binaries without any hassle (even with macOS support).

For a deploy, we could ship a binary file or a docker containing a binary file.

For a demo instance, we choose to use a Dockerfile,

# Improvement proposals

# Conclusions

# Webography

## Documentation and help

* Tailwind : <https://tailwindcss.com/docs>
* SASS : <https://sass-lang.com/>
* SweetAlert2 : <https://sweetalert2.github.io/>
* JQuery : <https://jquery.com/>
* GO Lang : https://go.dev/
* Echo : https://echo.labstack.com/docs
* Templ : <https://templ.guide/>
* TMDB API Docs : https://developer.themoviedb.org/docs/getting-started
* GO TMDB API : https://github.com/ryanbradynd05/go-tmdb
* GO FFmpeg Libraries :
* - https://github.com/u2takey/ffmpeg-go
* - https://github.com/go-ffstatic/ffstatic

## Tooling

* NPM
* Air

## Resources