XML Projet web

Dossier entities

Pour la gestion de la base de données, nous avons utilisé **MySQL**, qui nous a permis de mettre en place des fonctionnalités avancées. Pour utiliser MySQL avec Python, nous avons utilisé la bibliothèque ***mysql.connector***. Nous avons fait en sorte de **gérer les erreurs de manière précise.**

Fichier Database.py

L'objectif principal du fichier database.py est de permettre la commmunication **de la base de données** avec le reste du code et il contient les méthodes suivantes :

Méthode __init__

La méthode init, ou l'instanciateur de l'objet Database, permet de gérer les **opérations de connexion et de manipulation** de base de données, plus précisément elle permet :

- Stocker l'état de la connexion à la base de données une fois qu'elle est établie ;
- Stocker le curseur de la base de données ;
- Savoir si la base de données a déjà été créée ou non

Méthode connect

Cette méthode gère la **connexion à la base de données** en fonction de la propriété self.already_created. Elle utilise la bibliothèque mysql-connector pour établir la connexion en utilisant les informations de connexion spécifiées. En cas d'échec de la connexion, elle génère une erreur en appelant server_error. Cette dernière est une fonction permettant d'afficher un message d'erreur en rouge et souligné, et de faire soigneusement quitter l'application avant exécution du code suivant.

Méthode disconnect

Cette méthode est importante pour **libérer les ressources associées à la connexion** à la base de données une fois que les opérations nécessaires ont été effectuées. Fermer correctement la connexion et le curseur est une **bonne pratique** pour éviter les fuites de ressources et garantir que la base de données est déconnectée de manière propre

Méthode execute

Cette méthode gère **l'exécution de requêtes SQL** sur la base de données, en prenant en charge la connexion, l'exécution de la requête, la récupération des résultats, l'affichage de débogage et la déconnexion.

Méthode drop

Cette méthode permet de **supprimer toutes les données de la base de données**.

Méthode Create

La méthode create de la classe Database est conçue pour **créer la base de données** si elle n'existe pas déjà

Fichier movie.py

Méthode __init__
Cette méthode **vérifie si les données fournies sont valides** et contiennent toutes les informations nécessaires. Si les données sont invalides ou incomplètes, elle lève une **exception** avec un message d'erreur.

Méthode add
Cette méthode permet **d'ajouter un film depuis le site internet** dans la base de données. Elle crée une instance de la classe Database, puis exécute une requête SQL **INSERT INTO**, pour ajouter des données dans la table *movies* avec les valeurs des propriétés de l'objet *Movie*. **En cas d'erreur** lors de l'ajout du film, la fonction server_error est appelée et la méthode renvoie False.

Méthode get_all
Cette méthode **récupère tous les films depuis la base de données.** Si un emplacement est spécifié, elle **filtre les films par cet emplacement** en utilisant une jointure **INNER JOIN** entre les tables *movies* et *theaters*, liées par la colonne *theater.* La méthode utilise la classe Database définie dans le fichier *database.py* pour exécuter la requête SQL correspondante. Si une erreur se produit lors de la récupération des films, la fonction server_error est appelée et la méthode renvoie False. Sinon, elle renvoie les résultats de la requête sous forme de liste.

Fichier user.py
Ce fichier contient le code pour **la gestion des utilisateurs** du site et l'utilisation d'un **token JWT**.

Méthode __init__
La méthode __init__ initialise les propriétés de l'objet User avec les valeurs fournies dans le dictionnaire data. Plus spécifiquement, elle est utilisée pour initialiser un objet User avec un nom d'utilisateur (username) et un mot de passe (password).

Méthode login
La méthode login est utilisée pour **vérifier si l'utilisateur peut se connecter**. Elle crée une instance de la classe Database et exécute une requête SQL pour récupérer le **mot de passe haché** associé au nom d'utilisateur fourni. Si le nom d'utilisateur existe, elle appelle la méthode check_password pour vérifier si le mot de passe fourni correspond au mot de passe haché stocké dans la base de données.

Méthode check_password
La méthode check_password de la classe User est responsable **de vérifier si le mot de passe fourni par l'utilisateur correspond au mot de passe haché** stocké dans la base de données. La méthode commence par créer un objet de hachage **SHA-256** à l'aide de la fonction hashlib.sha256(). Ensuite, elle met à jour le hachage avec le mot de passe saisi par l'utilisateur et encodé en UTF-8 à l'aide de la méthode update(self.password.encode("utf-8")), puis elle calcule le **hachage hexadécimal** du mot de passe à l'aide de la méthode hexdigest(). Cela génère une représentation hexadécimale du hachage SHA-256 du mot de passe saisi par l'utilisateur. Enfin, la méthode compare le hachage calculé avec le mot de passe haché stocké dans la base de données (hashed_password). Si les deux hachages sont identiques, la méthode renvoie True, indiquant que le mot de passe fourni par l'utilisateur est correct.

Dossier Utils

Dans le répertoire Utils il y a deux fichiers :

Fichier db.py

Ce script permet de créer et de remplir notre base de données.
La méthode construct_db va dans un premier temps faire un appel à *drop* pour supprimer l'entièreté de la base de données. Cette fonctionnalité est importante, car si on relance notre script dans la même base de données sans faire de suppression ont pourrait avoir un problème de **"chevauchement"** certaine table pourrait alors ne pas se créer où conserver d'ancienne informations. **Repartir de zéro permet d'éviter ce genre de problème.** La partie création et suppression de la table est appelé du fichier database.py. Dans *db.py* nous n'aurons que les méthodes qui ont pour but de **"remplir" la base de données.**

Après la création de la base de données **on vérifie qu'elle a été créée correctement**. Si ce n'est pas le cas, on renvoie un **message d'erreur**. Sinon on crée des tables vides qui vont occuper la base de données, une fois cela fait on revoit un **code de succès** de l'exécution

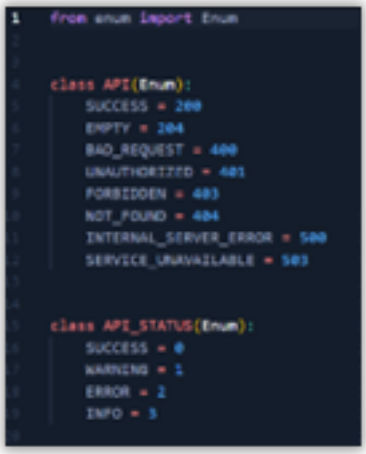
Le reste du script est composé de deux type de méthodes, les méthodes qui vont **créer les tables**, qui seront des commandes SQL suivie d'une **vérification de création**. Et des **méthodes de "populations"** qui vont remplir les tables de données. C'est ici que l'on va pouvoir choisir les données qui seront de base dans la base de données.

Fichier utils.py

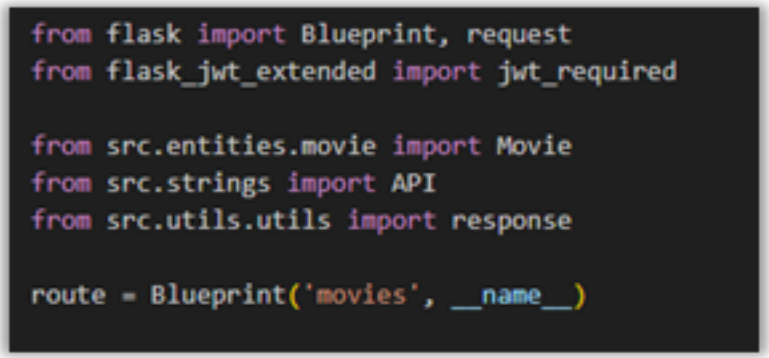
Ce script permet de **gérer les messages de retour lors de l'exécution de code**, dans ce code on va décrire deux messages d'erreurs :

- La méthode *server_error* est un message qui va être utilisé quand, lors de la création de base de données par exemple lorsque l'on constate un problème **et permettra potentiellement d'arrêter le serveur si c'est nécessaire.**
- La méthode *response* va pouvoir envoyer deux réponses. Si le code de retour vaut 200 (ce qui signifie un succès) on aura un retour "OK". Si jamais le code est différent de 200 on aura alors un échec et un retour "KO".

Le fichier string.py est pratique, car il permet de **paramétrer tous les codes de retour** utilisé et de les modifier facilement, on peut également modifier les catégories de message de retour.
Par ailleurs, si l'application devait être continué, il serait tout à fait aisé d'intégrer un logger pour avoir davantage d'informations sur les requêtes effectuées.



Fichier app.py



Ce script est le cœur du projet, puisque c'est ici que l'on va dans un premier temps va mettre en place une **technologie très intéressante et importante d'un point de vue sécurité**, l'authentification via JWT Token. Ce système d'authentification va permettre au client et au serveur de gérer le système de connexion de façon sécurisée à l'aide d'un token. Ce Token va être composé d'une **clé secrète** qui nous permettra de valider le Token et d'une durée de vie. Ainsi le Token doit être régulièrement rafraichi.

On a ensuite *CORS* qui a pour but de **réaliser des requêtes http depuis un client spécifique. Ici, pour des raisons évidentes de compatibilité, nous autorisons tous les clients à se connecter au serveur (*)**. Il s'agit cependant d'une pratique à abolir pour un logiciel en production, ou seul le frontend doit être autorisé dans cette partie.

On ajoute ensuite deux routes : une qui va avoir pour but de **gérer l'ajout de film** et une qui va être utiliser pour **l'authentification de l'utilisateur**.

Ensuite on va utiliser les méthodes *construct_db* et *populate_db* qui ont déjà été expliquées dans le script *db.py.* On va donc créer la base de données et son contenu. Il y a également la **gestion des erreurs** liées au Token qui sont prises en compte. On va donc être bloqué et avoir un message d'erreur si le Token a expiré, n'a pas la bonne valeur, n'est pas présent ou bien a été révoqué avant son expiration.

Routes

movies.py & source.py

Imports nécessaires surtout movie, & API.
Blueprint permet de rajouter des routes à notre serveur Flask de façon agile.

Route pour ajouter des films (add_movies) :

```python
@jwt_required
@route.route('/movies/add', methods=['POST'])
def add_movies():
    """
    Add movies
    :return:
    """

    data = request.get_json()
    if data is None:
        return response(API.BAD_REQUEST, "Invalid data")

    # Add the movie to the database
    movie = Movie(data)
    if not movie.add():
        return response(API.BAD_REQUEST, "Can't add movie")

    return response(API.SUCCESS, "OK")
```

```python
@route.route('/user/login', methods=['POST'])
def login():
    """
    Add movies
    :return:
    """

    # get data from request

    data = request.get_json()
    if data is None:
        return response(API.UNAUTHORIZED, "Invalid data")

    # Create the user and check if the login is correct
    user = User(data)

    if not user.login():
        return response(API.BAD_REQUEST, "Incorrect username or password.")

    token = create_access_token(identity=user.username)
    return response(API.SUCCESS, token)
```

Cette route est sécurisée avec JWT ce qui signifie qu'il y a besoin d'un token valide pour accéder à cette route. Ici double vérification : validité des données entrées, et vérification que l'objet movie est bien ajouté à la table. Avec deux codes d'erreurs respectifs et un code valide si réussite ;

Comme la route précédente, récupération du json et vérification de l'input. Ici, deux codes d'erreurs potentiels : Si le formulaire transmis est incorrect ; si l'utilisateur n'est pas connu de la BDD, alors un nouvel utilisateur est créé.

Front

```
// Assuming the server expects a Bearer token in the Authorization header
fetch('http://127.0.0.1:5000/movies/add', {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json; charset=UTF-8',
        'Authorization': 'Bearer ' + token
    },
    body: JSON.stringify(requestData)
})
    .then(response => response.json())
    .then(data => {
        if (data.response === "OK") {
            // Movie posted successfully, you can perform additional actions
            console.log('Movie posted successfully!');
            window.location.href = 'success.html';
        } else {
            errorMessage.innerText = 'Failed to post movie. Please check the form data.';
            errorMessage.style.color = 'red';
        }
    })
    .catch(error => {
        console.error('Error:', error);
        errorMessage.innerText = 'An error occurred. Please try again later.';
        errorMessage.style.color = 'red';
    });
});
```

Partie JS : Saisie sécurisée

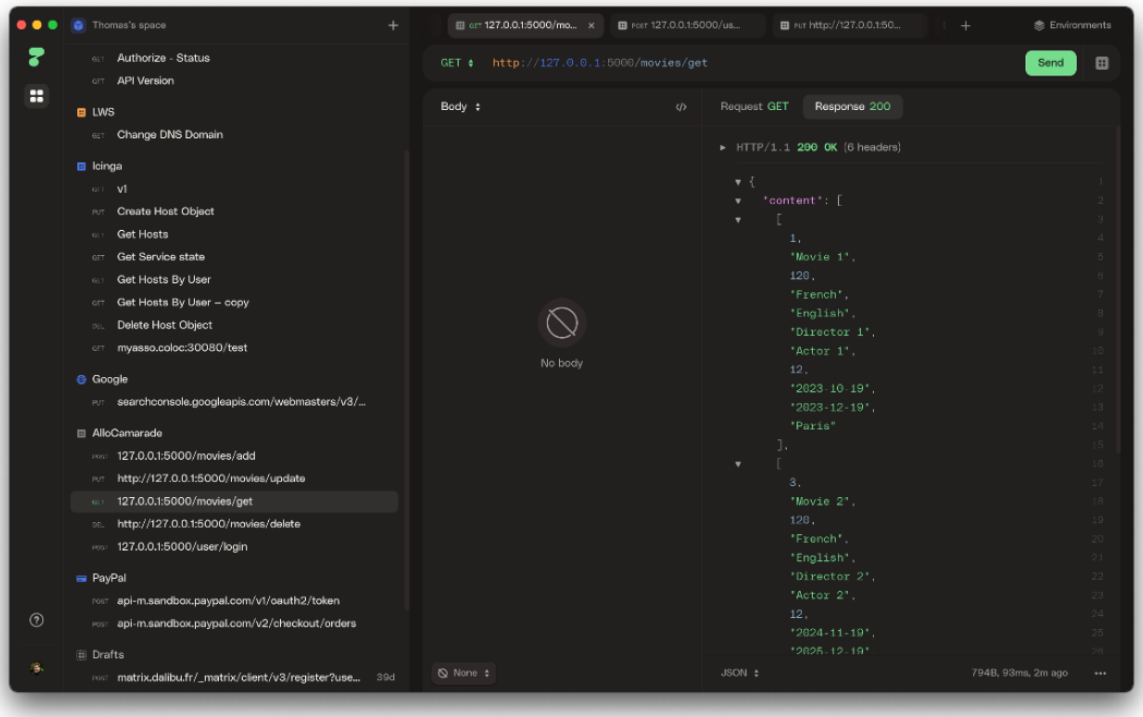
Utilisation de la fonction fetch pour envoyer une requête POST au serveur flask.
Les headers spécifient le type de contenu (Content-Type) et incluent un token d'authentification (Authorization) de type Bearer Token.
Le body de la requête contient les données (requestData) formatées en JSON.

Une fois la requête envoyée, le code traite la réponse du serveur.

Si la réponse est positive (data.response === "OK"), un message de succès est affiché et l'utilisateur est redirigé vers success.html.
Si la réponse indique un échec, un message d'erreur est affiché à l'utilisateur.

Idem que tout à l'heure, saisie sécurisée sur la page add. Envoie des données de connexion (requestData) formatées en JSON à l'API de connexion.

Si la réponse est "OK" et que les identifiants sont corrects, il stocke le token dans le stockage local et redirige l'utilisateur vers la page add.html. En cas d'identifiants incorrects, affiche un message d'erreur.



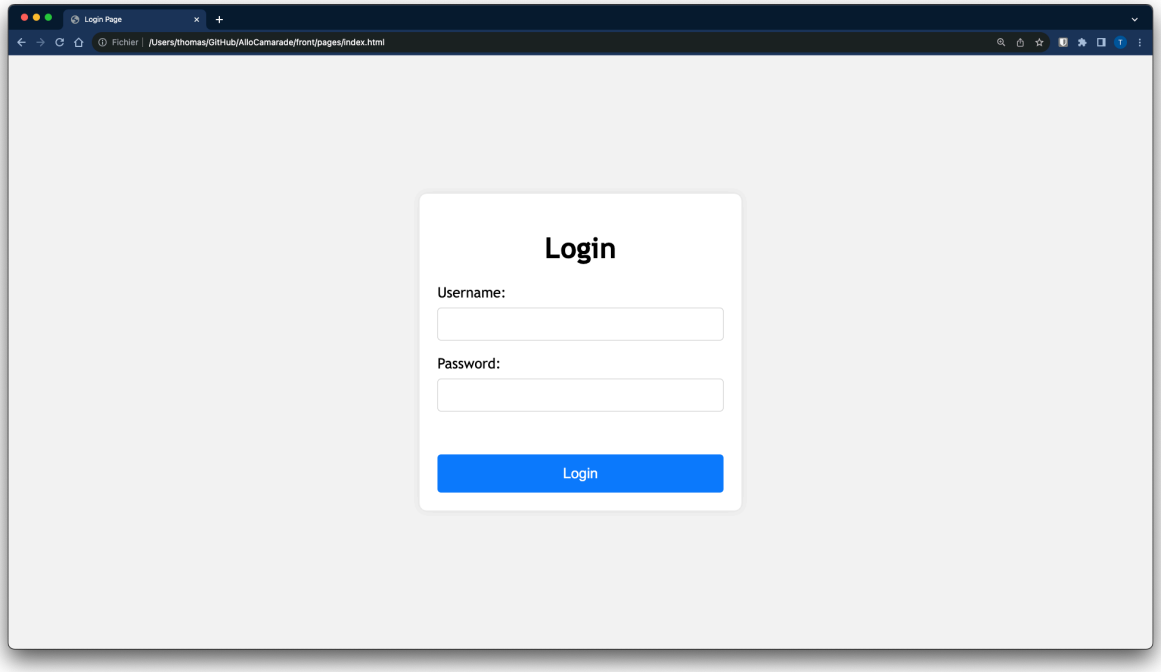
Le code gère et affiche les erreurs de la requête.

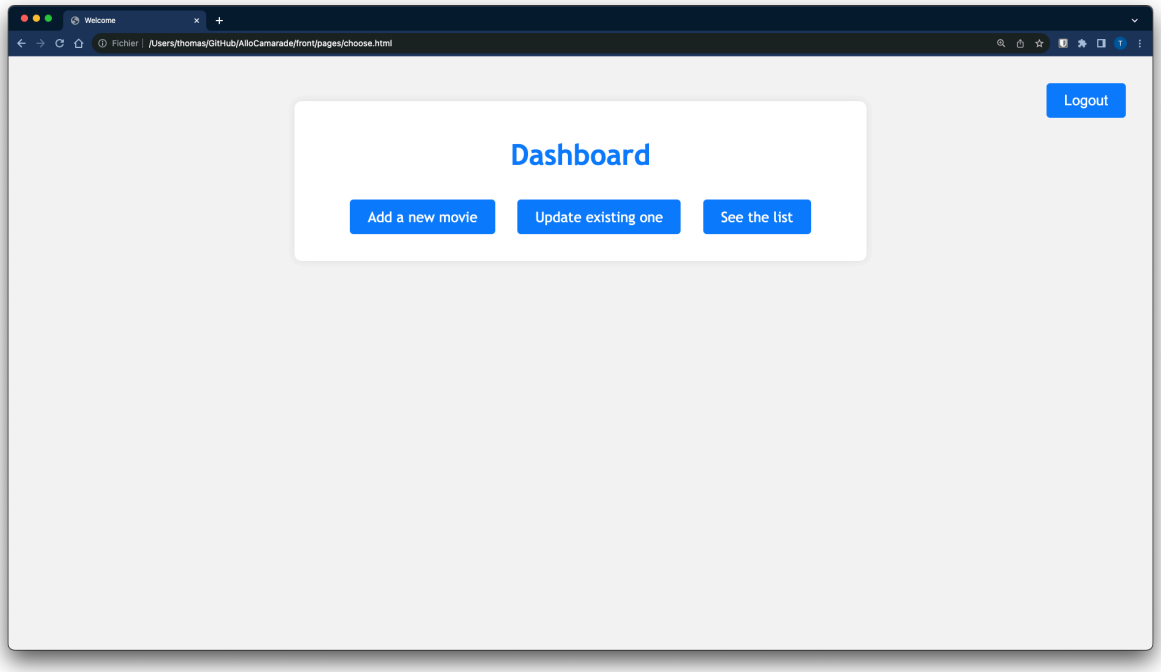
Ici grâce au logiciel HTTPIE vous pouvez voir que notre requête GET fonctionne correctement grâce au code 200.

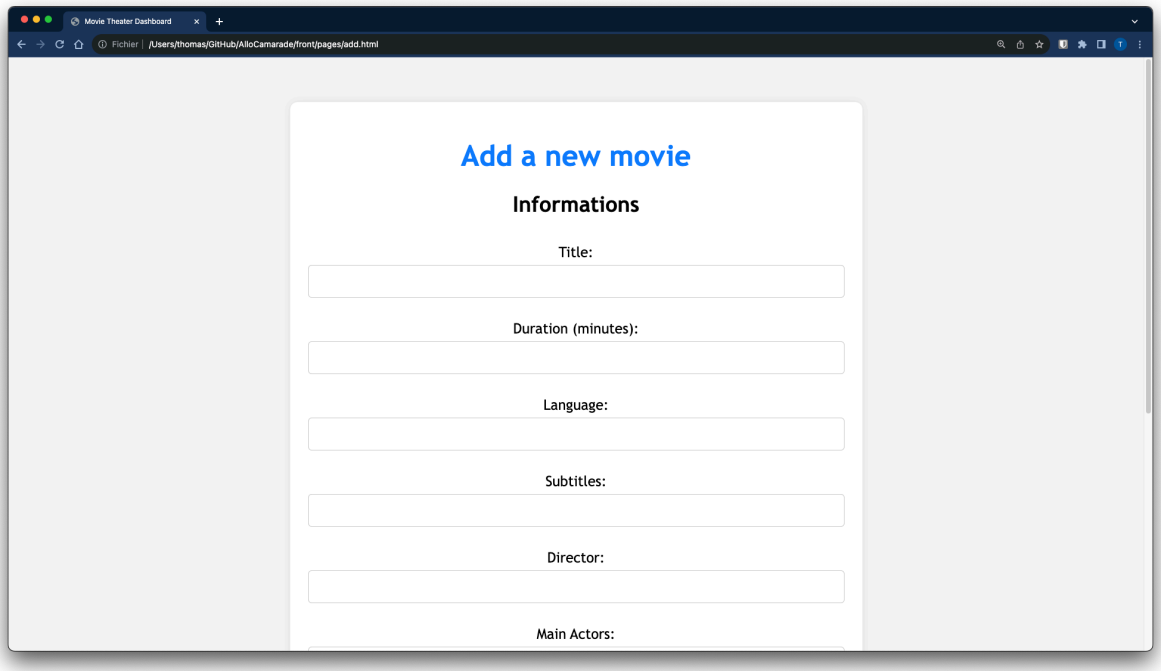
Pour terminer, il va de soi que projet a été développé en utilisant Github. Le code de test a été produit sur la branche staging, puis a été merge vers la branche de production. Le code source est disponible à ce lien :

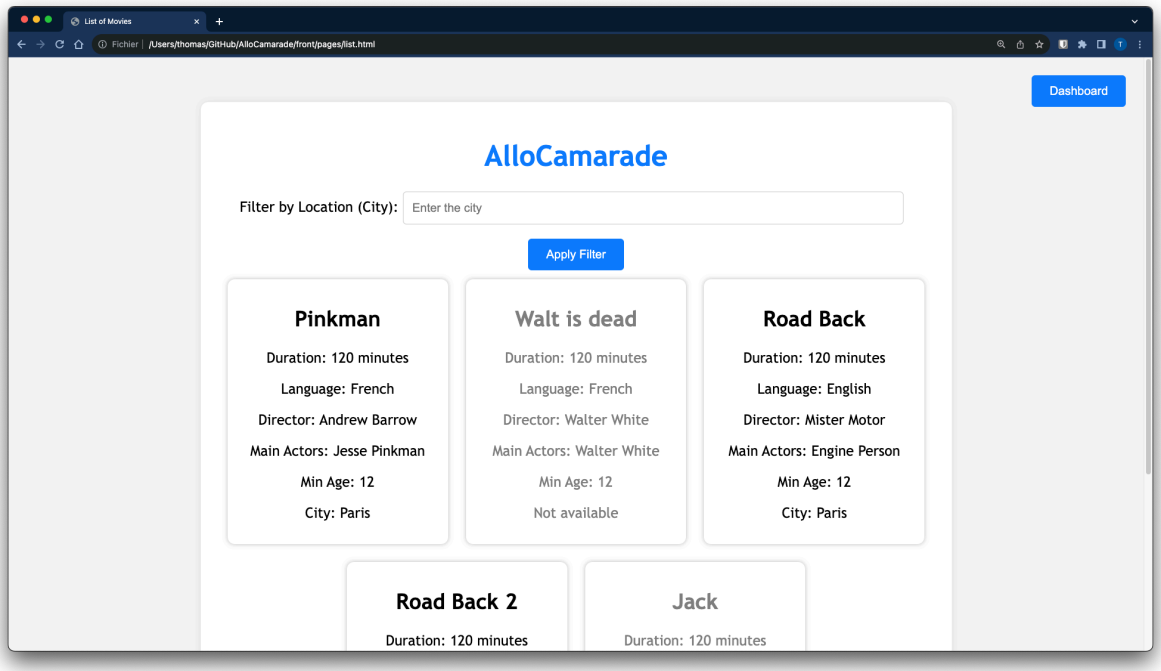
Si une démonstration est souhaitée, il suffit d'utiliser le docker-compose.yaml pour lancer la base de donnée, puis de lancer le serveur Flask.

Pour finir, voici quelques captures d'écran du frontend :









Il existe bien sûr les options pour modifier des films déjà créés, ou pour les supprimer.