

Titre RNCP 37873 Niveau VI  
Concepteur développeur d'applications

Dossier de projet - Juillet 2025  
Promo Juillet 2024

# Queer Cinema Database

Apolline Diaz

The screenshot shows the homepage of the Queer Cinema Database. The background is a dark, moody photograph of two people's faces, one with curly hair and one with blonde hair, looking towards each other. Overlaid on the image is the text "queer cinema database" in a red sans-serif font. In the center, there is a large, bold title: "FILMS & ARCHIVES" in white, and "LGBTQI+" in a gradient purple-to-blue font. At the bottom left is a white search bar with the placeholder "Rechercher un titre, mot-clé..." and a magnifying glass icon. At the bottom right is a red button with the text "Explorer le catalogue >". A small pink downward-pointing arrow is located at the bottom center of the page.

queer cinema database

FILMS &  
ARCHIVES  
LGBTQI+

Rechercher un titre, mot-clé...

Explorer le catalogue >

<b>I. Description du projet et introduction.....</b>	<b>4</b>
1. The project.....	4
2. Le projet.....	4
<b>II. Liste des compétences du référentiel.....</b>	<b>5</b>
I. RNCP37873BC01 - Développer une application sécurisée.....	5
II. RNCP37873BC02 - Concevoir et développer une application sécurisée organisée en couches.....	5
III. RNCP37873BC03 - Préparer le déploiement d'une application sécurisée.....	5
<b>III. Spécifications fonctionnelles.....</b>	<b>6</b>
I. Contexte du projet.....	6
II. Utilisateurs cibles et personas.....	6
III. Objectifs de l'application.....	7
IV. Définition du MVP.....	7
V. Modélisation UML.....	8
1. Diagramme de cas d'utilisation.....	8
2. Diagramme d'activité.....	8
3. Diagramme de classes.....	8
VI. Maquette interface.....	9
1. Identité graphique.....	9
2. Wireframes.....	10
3. Maquettes.....	10
<b>III. Spécifications techniques.....</b>	<b>11</b>
I. Choix des technologies.....	11
1. Next.js 14 et TypeScript.....	11
2. PostgreSQL et Supabase.....	12
3. Prisma.....	12
4. React Hook Form et Zod.....	13
5. Tailwind CSS et Shadcn.....	13
6. Recharts.....	13
7. Jest.....	13
8. Playwright.....	14
III. Hébergement.....	14
IV. Objectifs de qualité.....	15
1. Respect des normes et conformité RGPD.....	15
2. Accessibilité et expérience utilisateur.....	15
3. Éco-conception numérique.....	15
4. Tests et assurance qualité.....	16
5. Performance et optimisation.....	16
V. Architecture.....	17
VI. Conception base de données.....	18
1. Modèle conceptuel de données.....	18
2. Modèle logique de données.....	18
<b>IV. Gestion du projet.....</b>	<b>19</b>
I. Equipe.....	19

II. Planning et suivi.....	19
III. Environnement technique.....	21
1. Outil de conception : Figma.....	21
2. Environnement de développement : Mac, VSCode et Github.....	21
IV. Risques et difficultés anticipées.....	22
<b>V. Développements.....</b>	<b>23</b>
I. Contexte.....	23
II. Structure de l'application.....	24
1. Présentation de l'architecture du projet.....	24
IV. Connexion à la base de données.....	25
1. Supabase pour la gestion des fichiers et de l'authentification.....	25
2. Prisma pour la gestion des données relationnelles.....	26
V. Réalisation des interfaces : création et modification de film.....	28
1. Page de formulaire d'ajout de film.....	28
2. Processus d'édition d'un film.....	30
Composant d'affichage du bouton d'édition.....	30
Page d'entrée pour la modification de film.....	31
Formulaire d'édition.....	31
VI. Réalisation des services : gestion des films.....	35
1. Server Action pour l'ajout d'un film.....	35
2. Server Action pour la mise à jour des données d'un film.....	39
3. Server Action pour la suppression d'un film.....	42
VII. Sécurité.....	43
1. Authentification et autorisation avec Supabase Auth.....	43
2. Sécurité des identifiants.....	44
1. Protection des mots de passe par Supabase Auth.....	44
2. Supabase et cookies sécurisés.....	44
3. Middleware d'authentification et protection des routes.....	45
4. Implémentation des opérations d'authentification.....	47
5. Vérification d'identité et confirmation par mail.....	47
6. Gestion des permissions et des rôles.....	48
7. Alignement avec les recommandations d'OWASP.....	51
VIII. Déploiement.....	52
1. Architecture de conteneurisation et orchestration avec Docker Compose.....	52
2. Intégration Supabase et gestion des données.....	52
3. Processus de déploiement.....	53
<b>VI. Tests.....</b>	<b>54</b>
I. Tests unitaires.....	54
1. Test de la fonction addMovie.....	54
2. Test de updateMovie.....	55
II. Tests end-to-end.....	57
1. Test de connexion d'un utilisateur.....	57
2. Automatisation des tests.....	58
III. Tests d'accessibilité, de performance et SEO.....	58

IV. Tests utilisateurs.....	58
<b>VII. Documentation.....</b>	<b>60</b>
I. README.....	60
II. API.....	61
<b>VIII. Veille.....</b>	<b>61</b>
<b>IX. R&amp;D / Innovation : Containerisation et environnement de développement local....</b>	<b>63</b>
I. Description du besoin d'information.....	63
II. Solution trouvée et mise en oeuvre.....	64
1. Recherche de solutions et adoption d'OrbStack.....	64
2. Configuration de l'environnement de développement et intégration avec Supabase Studio.....	64
<b>Conclusion.....</b>	<b>65</b>
<b>Annexes.....</b>	<b>66</b>
Annexe 1 : Diagramme des cas d'utilisation.....	66
Annexe 2 : Diagramme d'activité.....	67
Annexe 3 : Diagramme de classes.....	68
Annexe 4 : Wireframes.....	69
Annexe 5 : Maquettes.....	72
Annexe 6 : Modèle conceptuel de données.....	74
Annexe 7 : Modélisation logique de données.....	75
Annexe 8 : Registre des traitements de données.....	76
Annexe 9 : Dockerfile.....	78
Annexe 10 : Fichier compose.ymal.....	79
Annexe 11 : Résultats du questionnaire pour le test utilisateurs (extraits).....	80

## I. Description du projet et introduction

### 1. The project

Queer Cinema Database is a web application that highlights films, series and audiovisual archives dealing with LGBTQI+ themes or featuring queer characters.

The project was born of the need to perpetuate the content of an Instagram account dedicated to LGBTQI+ cinema, in the face of the instability of social networks and the risks of censorship. Rather than depend on platforms such as IMDB, Letterboxd or Mubi, I chose to create a more targeted tool that would allow for a precise exploration of queer works. These existing platforms don't always make it possible to clearly identify the subjects addressed in the films. Searches are often too broad or irrelevant.

Queer Cinema Database aims to fill this gap with a smaller but better indexed database, enabling more precise referencing of works. Users will be able to carry out simple or advanced searches, combine several criteria and navigate freely from one page to another. Each film listed will be associated with keywords, genres and people (director), making it easier to discover similar content. Users will be able to create an account, save films and build their own lists.

### 2. Le projet

Queer Cinema Database est une application web qui met en avant les films, séries et archives audiovisuelles traitant de thématiques LGBTQI+ ou mettant en scène des personnages queer.

Le projet est né du besoin de pérenniser les contenus d'un compte Instagram dédié au cinéma LGBTQI+, face à l'instabilité des réseaux sociaux et aux risques de censure. Plutôt que de dépendre de plateformes comme IMDB, Letterboxd ou Mubi, j'ai choisi de créer un outil plus ciblé, qui permette une exploration précise des œuvres queer. Ces plateformes existantes ne permettent pas toujours d'identifier clairement les sujets abordés dans les films. Les recherches y sont souvent trop larges ou peu pertinentes.

Queer Cinema Database vise à combler ce manque avec une base plus restreinte mais mieux indexée, permettant un référencement plus précis des œuvres. Les utilisateurs pourront effectuer des recherches simples ou avancées, combiner plusieurs critères et naviguer librement d'une page à l'autre. Chaque film référencé sera associé à des mots-clés, genres et personnes (réalisateur·rice), facilitant la découverte de contenus similaires. Les utilisateurs auront la possibilité de créer un compte, sauvegarder des films et constituer leurs propres listes.

## **II. Liste des compétences du référentiel**

### **I. RNCP37873BC01 - Développer une application sécurisée**

Installer et configurer son environnement de travail en fonction du projet.

Développer des interfaces utilisateur

Développer des composants métier

Contribuer à la gestion d'un projet informatique

### **II. RNCP37873BC02 - Concevoir et développer une application sécurisée organisée en couches**

Analyser les besoins et maquetter une application

Définir l'architecture logicielle d'une application

Concevoir et mettre en place une base de données relationnelle

Développer des composants d'accès aux données SQL et NoSQL

### **III. RNCP37873BC03 - Préparer le déploiement d'une application sécurisée**

Préparer et exécuter les plans de tests d'une application

Préparer et documenter le déploiement d'une application

Contribuer à la mise en production dans une démarche DevOps

## III. Spécifications fonctionnelles

### I. Contexte du projet

Aujourd’hui, peu de plateformes mettent véritablement en valeur les œuvres audiovisuelles LGBTQI+. Les bases de données existantes (IMDB, The Movie Database, Mubi, Letterboxd, etc.) intègrent des contenus queer, mais leur indexation reste limitée et souvent insuffisante pour répondre aux besoins spécifiques de recherche des utilisateurs. Les mots-clés liés aux thématiques LGBTQI+ sont souvent trop génériques et il est difficile d’identifier précisément les sujets abordés.

Face à ces lacunes, Queer Cinema Database se présente comme un outil de consultation plus intuitif et accessible qui met en avant la richesse et la diversité des récits queer à l’écran.

### II. Utilisateurs cibles et personas

J’ai identifié plusieurs profils d’utilisateurs que voici :

- **Cinéphile curieux** : Un spectateur cinéphile qui s’intéresse à l’actualité cinématographique et a envie de découvrir de nouveaux films.
- **Passionné d’archives** : Un spectateur qui s’intéresse au cinéma de patrimoine et aimerait découvrir des films plus anciens dont il a plus difficilement connaissance.
- **Chercheur ou étudiant** : Un utilisateur qui fait de la recherche ou s’intéresse à l’histoire du cinéma et aimerait explorer un sujet à travers des archives audiovisuelles.
- **Programmateur de films** : Un utilisateur qui travaille dans le milieu du cinéma ou de la culture et prépare la programmation d’une séance, d’un festival ou tout autre événement autour d’une thématique liée aux sujets LGBTQI+.

### **III. Objectifs de l'application**

L'application a pour objectif de :

- Référencer et cataloguer des œuvres audiovisuelles LGBTQI+ (films, séries, archives).
- Permettre une recherche simple et avancée selon différents critères thématiques ou techniques.
- Faciliter l'exploration des œuvres par navigation croisée (liens entre les pages de films via les mots-clés).
- Offrir à l'utilisateur la possibilité de sauvegarder des œuvres sous forme de listes personnalisées.
- Centraliser et pérenniser les données dans une base indépendante, administrée de manière autonome.

### **IV. Définition du MVP**

J'ai défini un produit minimum viable incluant les fonctionnalités suivantes :

#### **Pages et navigation :**

- Page d'accueil avec listes thématiques et récents ajouts
- Catalogue avec affichage de tous les films sous forme de grille
- Page d'œuvre détaillée : titre, titre original, synopsis, image, réalisateur, année de sortie, pays de production, durée, langue, genre, mots-clés
- Navigation croisée entre les fiches et le catalogue (via des mots-clés)

#### **Utilisateur :**

- Inscription et authentification sécurisées
- Page de paramètres : modification des informations personnelles (nom), changement de mot de passe et suppression de compte
- Page de listes personnelles : création, modification et suppression de listes

#### **Recherche**

- Recherche simple par mot (titre, réalisateur, genre, mot-clé, etc.).
- Recherche avancée avec filtres combinables (période de sortie, genre, pays de production, réalisation, format, mots-clés).

#### **Administration :**

- Création, modification et suppression d'œuvres par l'administrateur.

## V. Modélisation UML

### 1. Diagramme des cas d'utilisation

Le diagramme de cas d'utilisation est un outil de modélisation UML (Unified Modeling Language). Il permet de visualiser toutes les interactions possibles entre les deux acteurs principaux et le système. Il spécifie les fonctionnalités accessibles à l'utilisateur final et celles spécifiques à l'administrateur. J'ai distingué un statut utilisateur et un administrateur. L'utilisateur standard accède aux fonctionnalités de consultation et de gestion personnelle des listes de films. L'administrateur possède toutes les capacités de l'utilisateur, avec en plus des droits étendus de gestion du contenu de la plateforme (ajout, modification et suppression de films et de listes, comprenant l'ajout et la modification d'images).

*Voir Annexe 1 pour le diagramme des cas d'utilisation.*

### 2. Diagramme d'activité

Le diagramme d'activité représente les différentes étapes et flux d'interaction des utilisateurs (utilisateur standard et administrateur) au sein de l'application. Il illustre le parcours depuis l'entrée sur le site jusqu'aux actions possibles telles que la navigation dans le catalogue, la gestion de listes, l'ajout ou la modification de films, ainsi que les processus d'authentification. Ce modèle permet de visualiser les choix possibles à chaque étape, les transitions entre les actions, et d'identifier les responsabilités spécifiques (utilisateur inscrit, administrateur). Dans le cadre du développement, ce diagramme m'a aidé à structurer le parcours utilisateur, à anticiper les cas d'erreur, et à optimiser l'expérience de navigation.

*Voir Annexe 2 pour le diagramme d'activité.*

### 3. Diagramme de classes

Le diagramme de classes est un outil UML permettant de modéliser la structure statique d'un système, en représentant les classes, leurs attributs, leurs méthodes (fonctions) ainsi que les relations entre elles. J'y décris les principales entités du système : User, Movie, List, Country, Genre, Keyword, et Director, ainsi

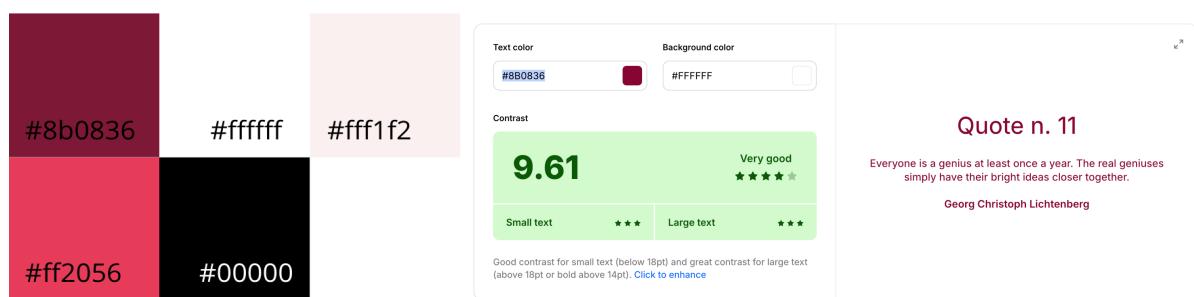
que leurs attributs (comme title, description, release\_date pour un film) et méthodes associées (par exemple, getMovie(), updateMovie() pour gérer les films). Ce modèle permet de structurer les données, de définir clairement les interactions possibles avec la base de données, et d'établir les liens logiques entre les différentes entités (par exemple, un utilisateur peut avoir plusieurs listes, un film peut avoir plusieurs genres). Il va me servir de référence pour la construction du backend et faciliter l'implémentation des fonctionnalités pour assurer la cohérence des données.

Voir **Annexe 3** pour le diagramme de classes.

## VI. Maquette interface

### 1. Identité graphique

L'application Queer Cinema Database est différente des autres plateformes de cinéma, qu'il s'agisse de celles qui proposent des services VOD (Cinetek, UniversCiné, Netflix) ou mettent à disposition une large bases de données (Letterboxd). Elles reproduisent l'atmosphère d'une salle de cinéma avec des tons sombres pour mettre en avant les visuels des films. Autrement, d'autres utilisent le blanc à la place avec des lignes droites et un style minimaliste, comme Mubi. Je souhaitais aller à contre-courant de ces designs pour rendre le site plus accueillant et inciter l'utilisateur à explorer le site. Le design tend à suivre une tendance moderne pour les formes arrondies (notamment pour les boutons et les vignettes de films), une esthétique épurée et trois grandes couleurs (rose, noir, blanc) avec des tons clairs. Le rose est également une couleur souvent reprise par les communautés LGBTQI+ qui renvoie à des émotions positives, de célébration et de fierté.



J'avais opté pour un fond rose clair #ffff1f2 en premier lieu, que j'ai utilisé pour mes maquettes. J'ai vérifié la lisibilité du texte avec un Contrast Checker avec [coolors.co](https://coolors.co) pour m'assurer de l'accessibilité. Le test a révélé qu'il était trop faible. J'ai donc opté pour du blanc dont l'accessibilité s'est confirmé (voir image ci-dessous).

## 2. Wireframes

J'ai réalisé des wireframes avec Figma pour avoir une représentation simple de l'interface utilisateur. J'ai ainsi structuré mon application en plusieurs pages et organisé les visuels (boutons, images, formulaires, etc.). Elles m'ont servi de guide pour planifier l'expérience utilisateur (UX). Elles m'ont permis d'avoir une vision claire de l'agencement des pages du MVP : l'accueil, le catalogue, la page d'un film, la page de listes, les fonctionnalités d'ajout, la modification.

Voir **Annexe 4** pour les wireframes

## 3. Maquettes

Les maquettes (ou mockups) vont représenter la version détaillée et visuellement aboutie de l'interface utilisateur. J'y ai intégré le design graphique final : couleurs, typographies, images, icônes. Je n'ai cependant pas intégré les interactions visuelles. Elles couvrent toutes les interfaces clés : la page d'accueil, la page de catalogue, la page d'un film, la page de listes et de création de liste, ainsi que les interfaces d'ajout et de modification de film.

Toutes les maquettes n'ont pas été réalisées, notamment celles pour la version mobile mais j'ai opté pour une approche mobile first. J'ai conçu la l'application en m'assurant d'adapter toujours les pages aux écrans de petite taille, notamment les smartphones. Cette approche répond aux usages actuels : aujourd'hui, quand on cherche des informations sur un film, c'est souvent sur son téléphone, en quelques secondes. Il fallait donc que l'application soit claire, rapide et agréable à utiliser sur un petit écran.

Au fil du développement, j'ai aussi fait évoluer le design des pages et la palette de couleurs pour améliorer l'accessibilité et la lisibilité. J'ai revu le fond d'écran (background) pour obtenir un meilleur contraste. J'ai retravaillé la structure de certaines pages pour mieux clarifier les rôles et les usages selon qu'on est simple utilisateur ou administrateur et distinguer les fonctionnalités des pages en tant qu'utilisateur connecté. Par exemple, au lieu d'une seule page de profil qui regroupait tout (comme les listes), j'ai ajouté un bouton avec un avatar dans la navbar qui donne accès à un menu déroulant contenant plusieurs sections bien distinctes comme "Mes listes", "Paramètres", et un bouton "Déconnexion". Pour les personnes ayant un rôle administrateur, une section supplémentaire intitulée "Contribuer" apparaît. Elle donne accès aux outils de gestion de films (ajout).

Voir **Annexe 5** pour les maquettes

## III. Spécifications techniques

### I. Choix des technologies

#### 1. Next.js 14 et TypeScript

J'ai choisi Next.js 14 comme fondation de mon application parce qu'il combine contenu statique et interactions dynamiques. Mon site doit gérer des fiches de films (contenu stable) et des recherches avancées (interactions temps réel), tout en maintenant des performances SEO optimales.

Next.js 14 est un framework React qui propose une approche "full-stack" avec rendu hybride, combinant le rendu côté serveur (SSR), la génération statique (SSG) et le rendu côté client selon les besoins de chaque page. Son App Router organise les routes via l'arborescence des dossiers et intègre automatiquement l'optimisation des performances (images, fonts, bundling).

Les Server Components s'exécutent côté serveur et génèrent du HTML complet, permettant aux moteurs de recherche d'indexer les pages de films avec toutes leurs métadonnées. Ils sont utilisés par défaut pour réduire le JavaScript envoyé au client.

Les Server Actions vont me permettre de gérer directement les opérations CRUD sans API REST séparée. Quand un administrateur ajoute un film ou qu'un utilisateur crée une liste, ces actions s'exécutent côté serveur directement depuis les composants.

L'Incremental Static Regeneration génère des pages statiques rapides qui se régénèrent automatiquement lors des mises à jour. Je l'utilise notamment pour ma page d'accueil avec un rafraîchissement régulier pour faire apparaître les nouveaux films ajoutés. Les pages de films sont aussi chargées instantanément pour un contenu toujours à jour.

TypeScript facilite l'autocomplétion et offre une documentation intégrée directement dans l'IDE. En créant des interfaces et types explicites, il permet de structurer les objets liés aux films, aux utilisateurs, et aux listes, ce qui rend le code plus lisible et maintenable. Le typage statique prévient ainsi les erreurs. Enfin, son intégration native avec Next.js s'effectue sans configuration supplémentaire, optimisant le workflow de développement.

## 2. PostgreSQL et Supabase

J'ai choisi d'utiliser une base de données relationnelle SQL, avec Postgresql car elle est bien adaptée à la multiplicité des données et à la complexité des relations de mon application. Chaque film est lié à plusieurs entités : genres, réalisateurs, mots-clés, pays, etc., avec des relations many-to-many fréquentes (par exemple : un film peut appartenir à plusieurs genres).

En tant qu'administrateur, on peut ajouter, modifier ou supprimer des films, tandis que les utilisateurs authentifiés peuvent créer et gérer des listes personnalisées de films. On peut également y faire une recherche textuelle (sur les titres, descriptions, réalisateurs, mots-clés, etc.) ou une recherche avancée combinant plusieurs filtres (genre, période de sortie, pays de production, réalisateur).

PostgreSQL me permet de répondre à ces exigences grâce à plusieurs atouts :

- Requêtes SQL complexes : pour croiser plusieurs critères lors de la recherche avancée.
- Intégrité des relations : pour éviter les incohérences entre les entités liées.
- Recherche textuelle : avec la possibilité d'utiliser des opérateurs ILIKE.
- Évolutivité : pour gérer un catalogue croissant de films et de métadonnées.

En complément, j'utilise Supabase comme Backend-as-a-Service, qui s'appuie directement sur PostgreSQL. Il apporte une solution complète clé en main pour gérer l'authentification avec prise en charge des rôles utilisateur et administrateur, ce qui facilite la sécurisation des accès. Supabase intègre aussi un stockage natif pour héberger les visuels de films, ce qui me permet de centraliser la gestion des médias dans la même plateforme.

## 3. Prisma

J'ai choisi comme ORM (Object-Relational Mapping) Prisma qui offre une interface type-safe pour interagir avec la base de données. Il me permet de définir mon modèle de données de manière claire, générant automatiquement des types TypeScript alignés avec ma base de données. Cette approche élimine les erreurs de typage. Les migrations automatisées gèrent les évolutions de schéma sans erreur manuelle. Prisma simplifie l'écriture des relations complexes many-to-many, générant des requêtes SQL optimisées pour maintenir la performance même avec des recherches avancées multi-critères.

## 4. React Hook Form et Zod

J'utilise React Hook Form pour la gestion des formulaires, car il utilise des références plutôt que des états pour suivre les champs, ce qui limite les re-rendus inutiles et améliore la fluidité de l'interface, même avec des formulaires complexes. La validation des champs est assurée par Zod, ce qui garantit des données fiables avant envoi. Il facilite la gestion des erreurs et s'adapte aux formulaires plus complexes, comme ceux pour ajouter un film ou effectuer une recherche avancée.

## 5. Tailwind CSS et Shadcn

J'utilise Tailwind CSS pour styliser l'interface, il permet de coder plus vite directement dans le JSX, sans jongler avec des fichiers CSS. Il optimise les performances en ne générant que le CSS réellement utilisé lors du build, ce qui réduit la taille des fichiers livrés. Pour les composants complexes comme les menus déroulants ou les onglets, j'ajoute shadcn/ui, qui repose sur Tailwind mais fournit des composants accessibles, bien structurés et facilement personnalisables, ce qui m'évite d'alourdir le code.

## 6. Recharts

J'utilise Recharts pour afficher les statistiques de la collection de films sous forme de graphiques interactifs et responsives. Son intégration avec React permet un rendu optimisé et une personnalisation simple pour visualiser des données comme la répartition de films par mot-clé.

## 7. Jest

Comme framework de test principal j'utilise Jest pour garantir la qualité et la stabilité du code. Il permet de réaliser des tests unitaires pour vérifier le bon fonctionnement des fonctions et composants de manière isolée. Il permet de simuler des dépendances externes grâce à des mockups tels que les appels API ou les connexions à la base de données, ce qui rend les tests plus rapides et indépendants de l'environnement. De plus, Jest fournit un rapport de couverture de code détaillé, permettant d'identifier les parties du code non testées.

## 8. Playwright

J'utilise Playwright pour les tests end-to-end qui simulent les interactions utilisateurs dans un environnement réel. Compatible avec plusieurs navigateurs, Playwright permet de valider le bon fonctionnement de l'application de bout en bout, assurant une expérience utilisateur fluide et cohérente.

## III. Hébergement

Pour l'hébergement de l'application, j'ai choisi d'utiliser Vercel, une plateforme optimisée pour les projets Next.js, avec un déploiement rapide, une scalabilité automatique et une intégration continue fluide. Vercel permet un déploiement facile grâce à la connexion directe avec le dépôt GitHub, qui déclenche automatiquement un déploiement à chaque modification du code. De plus, les fonctionnalités comme le CDN mondial, le rendu statique ou côté serveur (SSR), et l'optimisation des performances garantissent un chargement rapide et une expérience utilisateur optimale.

La connexion avec la base de données PostgreSQL gérée par Supabase se fait via des variables d'environnement définies directement dans Vercel. Ces variables stockent les informations sensibles telles que l'URL de la base de données, le mot de passe et les clés d'API. Lors du déploiement, Vercel les rend accessibles à l'application, permettant à Prisma de communiquer de manière sécurisée avec Supabase pour les opérations CRUD. Cette intégration simplifiée évite les problèmes de configuration réseau et garantit la sécurité des données échangées.

## IV. Objectifs de qualité

### 1. Respect des normes et conformité RGPD

L'application doit respecter les exigences du Règlement Général sur la Protection des Données (RGPD) pour assurer la confidentialité et la sécurité des informations personnelles des utilisateurs. Les données collectées (comme les informations de compte ou les listes de films) sont strictement limitées aux éléments nécessaires et sont stockées dans une base de données sécurisée via Supabase. Les utilisateurs ont la possibilité de modifier ou supprimer leurs informations personnelles à tout moment.

### 2. Accessibilité et expérience utilisateur

L'application doit être conforme aux critères du RGAA pour garantir l'accessibilité à tous les utilisateurs, y compris les personnes en situation de handicap. Les principales mesures prises incluent la navigation au clavier qui permet d'accéder à toutes les fonctionnalités sans souris, la compatibilité avec les lecteurs d'écran grâce à un balisage sémantique avec des rôles ARIA appropriés, des contrastes optimisés dans les choix de couleurs pour les utilisateurs ayant une déficience visuelle, et des tests d'accessibilité réguliers avec Lighthouse et axe-core qui valident la conformité lors de l'intégration continue.

### 3. Éco-conception numérique

L'application est développée en tenant compte des principes d'éco-conception numérique pour minimiser son impact environnemental. La réduction de la consommation de ressources est priorisée à chaque étape grâce à l'optimisation des performances avec les Server Components de Next.js qui réduisent la charge côté client et le code splitting qui évite le chargement inutile de ressources, l'utilisation d'images optimisées au format WebP chargées progressivement via Next.js Image pour réduire la taille des médias, la minimisation des requêtes réseau par la mise en cache côté serveur et l'utilisation de données statiques quand cela est possible, et un hébergement durable sur Vercel qui utilise une infrastructure optimisée pour l'efficacité énergétique.

## 4. Tests et assurance qualité

Pour garantir la fiabilité et la robustesse de l'application, une stratégie de tests complète est mise en place incluant des tests unitaires avec Jest pour assurer la fiabilité des fonctions critiques, des tests end-to-end avec Playwright pour vérifier les parcours utilisateurs critiques, des tests utilisateurs avec entretiens ou formulaires pour améliorer l'UX, et des tests d'accessibilité avec Lighthouse pour valider la conformité aux normes RGAA. La qualité du code est maintenue par un système de linting et formatage avec ESLint et Prettier qui garantissent un code propre et uniforme compatible avec TypeScript, ainsi qu'une revue systématique des pull requests pour identifier d'éventuelles erreurs ou améliorations.

## 5. Performance et optimisation

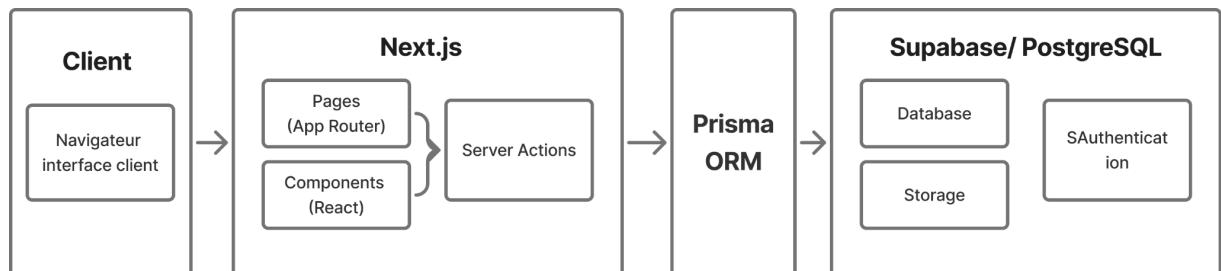
L'optimisation des performances est au cœur du développement pour offrir une expérience utilisateur rapide et fluide avec un objectif de First Contentful Paint sous 2 secondes et Time to Interactive sous 3,5 secondes. Cette optimisation repose sur la réduction de la taille des bundles grâce au code splitting natif de Next.js (au lieu d'un seul gros bundle, Next.js crée plusieurs petits bundles qui se chargent seulement quand nécessaire), l'utilisation de Server Components pour minimiser la charge client, et une analyse de performance avec Lighthouse pour surveiller et corriger les problèmes de latence.

## V. Architecture

L'architecture de l'application repose sur Next.js 14 et son App Router, qui permet une structuration claire par pages et composants serveur, optimisant le rendu via React Server Components et le streaming. Le backend est géré avec Supabase (PostgreSQL, Auth, Storage), et les accès à la base sont typés via Prisma. Les données sont validées côté serveur avec Zod, et côté client via React Hook Form, ce qui apporte une validation sécurisée. L'interface est construite avec Tailwind CSS et enrichie de composants accessibles de Shadcn, tandis que les statistiques sont visualisées avec Recharts. Le développement local est containerisé avec Orbstack (Docker), les tests sont assurés par Jest (unitaires) et Playwright (end-to-end), et le déploiement est géré par Vercel, garantissant performance et scalabilité. J'utilise également Resend pour le formulaire de contact.

L'architecture suit un modèle inspiré de MVC (Modèle-Vue-Contrôleur), adapté aux spécificités de Next.js :

- Modèle : gère les données et la logique métier avec Prisma qui est connecté à PostgreSQL via Supabase
- Vue : interface utilisateur avec les composants React
- Contrôleur : lien entre modèle et vue qui gère les interactions utilisateur avec les Server Actions de Next.js



## VI. Conception base de données

### 1. Modèle conceptuel de données

Le Modèle Conceptuel de Données (MCD) représente de manière abstraite les informations et les relations entre les différentes entités du système. Il sert à structurer les données en définissant les entités principales (comme Movie, User, List, Country, Genre, Keyword, Director) et leurs relations. Ce modèle permet de visualiser les dépendances et les cardinalités, facilitant la création de la base de données relationnelle lors du développement.

Dans ce MCD, chaque film (Movie) est lié à un ou plusieurs genres (Genre), pays (Country), mots-clés (Keyword) et réalisateurs (Director). Un utilisateur (User) peut créer plusieurs listes (List) et ajouter des films, tandis qu'un film est associé à un seul utilisateur pour son ajout. Les cardinalités (1,n, 0,n) illustrent les liens de dépendance : par exemple, un film peut être associé à plusieurs genres (1,n), et une liste est toujours liée à un seul utilisateur (1,1). Ce modèle guide la structuration de la base de données et assure la cohérence des relations entre les données.

Voir **Annexe 6** pour le modèle conceptuel de données

### 2. Modèle logique de données

Le Modèle Logique de Données (MLD) est une représentation plus technique et détaillée du Modèle Conceptuel de Données (MCD), spécifiant la structure de la base de données relationnelle. Il décrit précisément les tables, les attributs, les types de données, les clés primaires (PK), les clés étrangères (FK) ainsi que les relations entre les entités.

Il comprend les tables principales comme movies, users, directors, countries, genres, keywords et lists, qui stockent les informations clés. Les associations many-to-many sont gérées par des tables intermédiaires (movies\_directors, movies\_genres, movies\_countries, movies\_keywords, lists\_movies) pour relier les films à leurs réalisateurs, genres, pays, mots-clés et listes d'utilisateurs. Ce modèle permet de normaliser les données, d'optimiser les requêtes SQL et de garantir l'intégrité des relations lors du développement.

Voir **Annexe 7** pour le modèle logique de données

## **IV. Gestion du projet**

### **I. Équipe**

J'ai développé seul le projet, avec une approche full-stack où j'ai géré l'ensemble des étapes de conception, développement et déploiement. J'ai donc fait de l'auto-review de code, en mettant en place des temps de relecture et refactoring pour pallier l'absence de revue par des pairs. J'ai donc porté différents rôles, de la conception UX/UI au développement front-end et back-end en passant par l'administration système.

Pendant ma formation, j'ai bénéficié d'un encadrement pédagogique avec des sessions hebdomadaires. Des encadrants suivaient régulièrement l'avancement de mon projet, avec des points hebdomadaires pour discuter de mes choix techniques, résoudre d'éventuels blocages et recevoir des conseils avisés. Ce suivi m'a permis de progresser et de prendre des décisions éclairées tout au long du développement.

### **II. Planning et suivi**

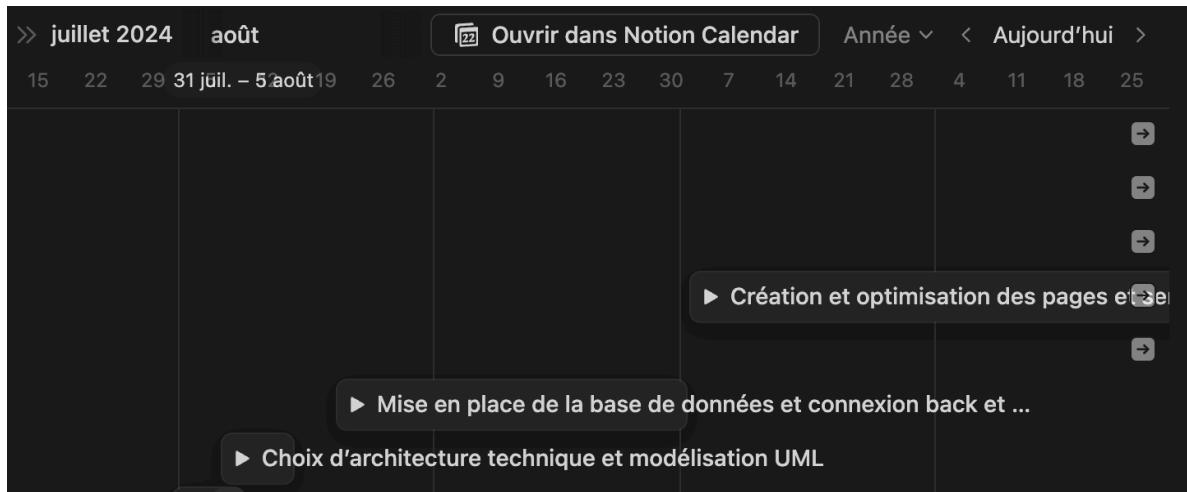
J'ai utilisé Notion comme espace de travail. Il est très pratique car il est connecté à d'autres outils de productivité ou design. J'ai pu intégrer mes fichiers Figma (maquettes, modélisation conceptuel de données, wireframes) et Canva (pour les démos) par exemple. J'ai créé un espace où j'ai ajouté tous les documents utiles pour le développement de mon application : recherche, étude de marché, enquête utilisateur, persona, scénario, MCD et MLD, wireframes, prototypes, *moodboard*, charte graphique, maquette, roadmap, backlog, table de tâches, chronologie (diagramme de Gantt), tests utilisateurs, démos, dossier de projet et dossier professionnel.

J'ai créé dès le lancement du projet, en début d'année, un tableau de tâches et une timeline qui ont fait œuvre de roadmap sur Notion. J'ai créé une base de données avec le modèle de Table. J'ai ajouté toutes les tâches que je prévoyais de faire et les deadlines que je me suis fixées. J'ai créé une autre vue chronologique de cette même base de données pour avoir une vision dans le temps de chaque tâche. J'ai organisé mes tâches par grande catégorie (la conception, le back, le front, les fonctionnalités, les tests, etc.) sur les 12 mois. J'ai adapté au fil du temps le calendrier en fonction des avancées.

J'ai créé des sous-éléments qui sont traités dans les *pull requests* qui sont mentionnées sur la même ligne. J'indique l'état d'avancée par plusieurs statuts : "pas commencé", "en cours" et "pas commencé". J'ai regroupé les éléments dans des groupes qui représentent les étapes du développement du projet (la conception, le développement des pages, l'installation de l'ORM, la dockerisation, les tests, etc.).

Debug : affichage de l'image (utilisation Image Composant de Next)	14/04/2025	Fix	#55 feat: use Image import from next Merged	Terminé
Ajout de la taille à l'image dans Hero	14/04/2025	Feature	#56 fix: add size to hero image Merged	Terminé
Ajout d'une configuration pour le port dans next config	14/04/2025	Chore	#57 chore: add url for local image Merged	Terminé
Réduction des requêtes dans server-action get-movies par une seule requête	14/04/2025	Feature	#58 feat: reduce server action get movies by word in one only re... Merged	Terminé

Et voici un extrait de ma timeline adaptée de la liste de tâches :



La démarche Scrum, que j'ai adaptée à mon travail en solo, m'a permis de m'organiser de manière flexible et agile, en m'adaptant au fil du développement. J'ai découpé mon projet en sprints hebdomadaires, chacun dédié à une tâche précise : implémentation d'une nouvelle fonctionnalité, correction de bugs ou mise en place de nouveaux outils (comme les tests par exemple). Chaque semaine, j'évaluais mes avancées pour ajuster les priorités.



Pour le suivi des tâches j'ai utilisé un modèle de tableau kaban proposé par Notion. Je l'ai divisé en 3 groupes de tâches intitulés respectivement "terminé", "en cours", et "pas commencé" pour avoir un suivi de mes avancées et des tâches en cours de traitement et à venir.

### III. Environnement technique

#### 1. Outil de conception : Figma

Figma est un outil de conception en ligne idéal pour le développement d'une application. Je l'ai utilisé pour créer les modèles de données UML et Merise pour structurer les informations, ainsi que les wireframes et les maquettes pour visualiser l'interface utilisateur.

#### 2. Environnement de développement : Mac, VSCode et Github

J'utilise un ordinateur MacBook Air (processeur Apple Silicon M2, 16 Go RAM), avec un système d'exploitation macOS Monterey (version 12.6). J'utilise comme terminal iTerm2 avec la configuration Oh My Zsh personnalisée pour l'efficacité des commandes.

J'utilise comme IDE principal Visual Studio Code, qui est enrichi par des extensions spécialisées tel que ESLint et Prettier pour garantir la qualité et le formatage du code, et GitLens pour une gestion avancée des versions et un historique détaillé des modifications.

Pour le développement de l'application, j'ai utilisé GitHub comme outil de gestion de version en suivant une approche Feature-Based Development. Cette méthode consiste à créer une branche dédiée pour chaque fonctionnalité ou correction, nommée de façon descriptive avec des préfixes standardisés (feat: pour les nouvelles fonctionnalités, fix: pour les corrections de bugs). Pour maintenir un workflow optimal, je limitais le travail à deux pull requests maximum simultanément, ce qui permet d'avancer méthodiquement tout en évitant la dispersion du code dans de multiples branches et les conflits lors des fusions. Chaque pull request était systématiquement ouverte pour permettre une revue de code. J'ai pu ainsi faire un suivi des modifications, procéder à une validation systématique avant intégration, et progresser de manière plus structurée avec un historique traçable.

Le dépôt était structuré avec une branche principale (main) pour le déploiement en production et des branches secondaires pour chaque ajout ou modification. J'ai également automatisé les workflows avec GitHub Actions, avec des tests unitaires et end-to-end (Jest et Playwright), et un build de l'application avec Vercel.

## IV. Risques et difficultés anticipées

Le développement en solo d'un site de référencement de films LGBTQI+, sur une durée d'un an, a soulevé plusieurs risques et difficultés anticipées en amont du projet, que j'ai tenté de prévenir ou de gérer de manière proactive. Travailler seule impliquait une prise en charge complète du cycle de vie du projet, ce qui génère une charge de travail élevée et demande une polyvalence constante. Le risque principal était lié à la gestion du temps : avec un volume important de tâches à réaliser (conception, développement, tests, déploiement), il était crucial d'adopter une méthode de travail rigoureuse pour éviter les retards et l'accumulation de dettes techniques. J'ai donc structuré mon organisation autour d'un tableau de bord détaillé (via Notion) et d'une démarche Scrum adaptée à mon rythme hebdomadaire.

Un second risque concernait la complexité technique liée à la stack choisie : Next.js, Supabase, PostgreSQL, Tailwind CSS, Prisma et TypeScript. Ces technologies, bien que puissantes et modernes, impliquent une courbe d'apprentissage importante, notamment pour des sujets comme le typage strict en TypeScript, la gestion des relations entre tables avec Prisma, ou encore l'optimisation des performances avec Next.js (rendering SSR/ISR). J'ai dû approfondir certains concepts en autonomie, ce qui a parfois ralenti l'avancement, mais a aussi renforcé mes compétences full-stack. Mon alternance m'a permis de mieux prendre en main ces technologies qui y étaient aussi utilisées, avec le soutien de mon mentor.

En tant que projet thématiquement engagé (référencement de films LGBTQI+), une autre difficulté potentielle résidait dans la qualité et la sensibilité des contenus référencés. J'ai pris soin de définir des critères de sélection clairs et documentés, en privilégiant des sources fiables, pour garantir une offre de contenu pertinente et complète.

Enfin, l'absence de revue de code par les pairs constituait un point de vigilance. Pour limiter les erreurs et maintenir la qualité du code, j'ai mis en place un processus systématique d'auto-relecture, de refactoring régulier, et de tests automatisés (unitaires et end-to-end avec Playwright) ainsi qu'auprès d'utilisateurs. L'utilisation de GitHub avec des pull requests, même en solo, m'a permis de ritualiser ces étapes et d'apporter une rigueur dans mon développement.

Malgré ces difficultés, l'encadrement pédagogique hebdomadaire m'a apporté un appui précieux pour valider mes choix, débloquer certains points techniques, et maintenir un cap clair sur la progression du projet.

## V. Développements

### I. Contexte

Mon projet consiste en une plateforme de référencement de films LGBTQI+, conçue pour valoriser les œuvres cinématographiques à thématique LGBTQI+. Cette plateforme répond à un besoin d'accessibilité et de visibilité pour ces contenus souvent sous-représentés dans les catalogues de streaming traditionnels et les bases de données de cinéma. L'application permet aux utilisateurs de découvrir des films et séries à travers une interface intuitive avec des fonctionnalités de recherche avancées et des listes personnalisées.

L'objectif principal est de créer un écosystème complet où les utilisateurs peuvent explorer et rechercher des films LGBTQI+ selon leurs préférences. L'administrateur peut, quant à lui, gérer le contenu de la plateforme grâce à des fonctionnalités comme l'ajout, la modification et la suppression de films.

Dans le cadre de ce projet, l'application permet à un utilisateur avec un rôle administrateur de modifier les informations d'un film. Cette fonctionnalité implique un ensemble de mécanismes de sécurité, de gestion de données, et d'interface utilisateur pour garantir une expérience fluide et sécurisée. Il peut également créer des listes et en faire des collections pour qu'elles soient publiques. Cette dernière fonctionnalité n'était pas un requis dans le MVP, je ne développerais pas sa mise en place.

Dans cette partie, je vais expliquer le besoin métier : permettre à un administrateur d'ajouter un film, de modifier les informations d'un film et le supprimer directement depuis l'interface utilisateur, en respectant les permissions d'accès et en garantissant la cohérence des données.

## II. Structure de l'application

### 1. Présentation de l'architecture du projet

```
> .github
> .next
> .vscode
< app
  > about
  > account
  > api
  > auth
  > components
  > contact
  > error
  > lists
  > login
  > logout
  > movies
  > privacy-policy
  > server-actions
  > signup
  > stats
  > types
  ★ favicon.ico
  # globals.css
  ⚡ layout.tsx
  📄 page.tsx
  TS sitemap.ts
  > components
  > coverage
  > lib
  > node_modules
  > playwright-report
  > test-results
  > tests
  > utils
  📄 .dockerignore
  ⚙ .env
  $ .env.example
  $ .env.local
  ⚙ .eslintrc.json
  📄 .gitignore
  { } components.json
  📄 compose.yaml
  📄 Dockerfile
  TS jest.config.ts
  TS middleware.ts
  TS next-env.d.ts
  JS next.config.mjs
  { } package-lock.json
  { } package.json
  TS playwright.config.ts
  ! pnpm-lock.yaml
  JS postcss.config.mjs
  📄 README.Docker.md
  ⓘ README.md
  TS tailwind.config.ts
  TS tsconfig.json
```

Le projet est structuré selon une architecture modulaire. Le dossier **app** contient les routes et les pages principales, chacune isolée dans un sous-dossier (ex : **account**, **login**, **movies**, etc.). On y trouve les composants globaux dans **components**, ainsi que des actions serveur dans **server-actions** et spécifiquement dans les pages **account** et **login**.

Le dossier **types** contient les définitions de types TypeScript.

Le dossier **lib** contient des fonctions utilitaires centralisées (comme supabase, prisma, ou markdown) qui facilitent l'accès aux services externes (base de données, authentification, parsing) à travers l'application.

Le schéma avec Prisma est défini dans **prisma/schema.prisma**.

Les tests sont organisés avec une distinction entre **e2e** (end-to-end) et **unit**, placés dans le dossier **tests**. Des outils de test tels que Playwright (**playwright.config.ts**) et Jest (**jest.config.ts**) sont configurés.

Le dossier **utils** regroupe des fonctions utilitaires facilitant la gestion des utilisateurs, la vérification des rôles (**is-user-admin.ts**), ou encore l'upload d'images (**upload-image.ts**).

L'environnement est géré via des fichiers **.env** et **.env.local**, tandis que la configuration du projet est assurée par plusieurs fichiers (**Next.js**, **Tailwind**, **ESLint**, **Docker**, etc.). Enfin, le projet utilise pnpm comme gestionnaire de paquets (**pnpm-lock.yaml**).

## IV. Connexion à la base de données

L'application utilise une architecture hybride combinant Supabase et Prisma pour accéder à la même base de données PostgreSQL, mais avec des rôles différents. Supabase gère l'authentification et certaines opérations via son API (temps réel, stockage, fonctions, etc.). Prisma donne un accès direct à la base de données avec vérification de type TypeScript.

### 1. Supabase pour la gestion des fichiers et de l'authentification

Supabase joue un rôle central dans la gestion des services externes de l'application, principalement pour l'authentification des utilisateurs et le stockage des fichiers images des films. En tant que couche d'API par-dessus PostgreSQL, Supabase ne connecte pas directement à la base de données mais fournit une interface sécurisée qui gère automatiquement l'authentification, les autorisations et les sessions utilisateur via des cookies. L'application utilise deux configurations distinctes : le client côté navigateur (utils/supabase/client.ts) pour les interactions directes comme la réinitialisation de mot de passe, et le client côté serveur (utils/supabase/server.ts) via createServerClient() pour les Server Actions et la gestion des sessions dans les composants serveur.

#### utils/supabase/client.ts

```
1 import { createBrowserClient } from "@supabase/ssr";
2
3 const supabaseUrl = process.env.NEXT_PUBLIC_SUPABASE_URL!;
4 const supabaseAnonKey = process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY!;
5
6 export const supabase = createBrowserClient(supabaseUrl, supabaseAnonKey);
```

#### utils/supabase/server.ts

```
1 import { createServerClient, type CookieOptions } from "@supabase/ssr";
2 import { cookies } from "next/headers";
3
4 export function createClient() {
5   const cookieStore = cookies();
6
7   return createServerClient(
8     process.env.NEXT_PUBLIC_SUPABASE_URL!,
9     process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY!,
10    {
11      cookies: {
12        getAll() {
13          return cookieStore.getAll();
14        },
15        setAll(cookiesToSet) {
16          try {
17            cookiesToSet.forEach(({ name, value, options }) =>
18              cookieStore.set(name, value, options)
19            );
19          }
20        }
21      }
22    )
23  }
24
25  
```

## 2. Prisma pour la gestion des données relationnelles

Prisma sert d'ORM (Object-Relational Mapping) pour l'accès direct et typé à la base de données PostgreSQL, offrant une interface moderne et sécurisée pour les opérations complexes. Configuré dans lib/prisma.ts, il utilise une instance globale unique pour éviter les connexions multiples en développement et intègre un système de cache avancé avec unstable\_cache de Next.js pour optimiser les performances des requêtes. Dans les Server Actions comme addMovie.ts, Prisma orchestre des opérations CRUD complexes avec validation Zod robuste en TypeScript, permettant des requêtes transactionnelles avancées telles que l'upsert conditionnel de réalisateurs, l'insertion relationnelle via createMany, et la gestion sécurisée des fichiers via Supabase.

### lib/prisma.ts

```
1  import { PrismaClient } from "@prisma/client";
2  import { unstable_cache } from "next/cache";
3
4  const globalForPrisma = global as unknown as { prisma: PrismaClient };
5
6  export const prisma =
7    globalForPrisma.prisma ||
8    new PrismaClient({
9      datasources: {
10        db: {
11          url: process.env.DATABASE_URL,
12        },
13      },
14      log: ["query", "info", "warn", "error"],
15    });
16
17 if (process.env.NODE_ENV !== "production") globalForPrisma.prisma = prisma;
18
```

## prisma/schema.prisma (extrait)

```
1  generator client {
2    provider = "prisma-client-js"
3    |   binaryTargets = ["native", "linux-musl-arm64-openssl-3.0.x"] // allow Prisma to
4  }
5
6  datasource db {
7    provider = "postgresql"
8    url      = env("DATABASE_URL")
9  }
10
11 /// This model or at least one of its fields has comments in the database, and requ
12 /// This model contains row level security and requires additional setup for migrat
13 model countries {
14   created_at    DateTime          @default(now()) @db.Timestamptz(6)
15   name          String           @unique @db.VarChar
16   id            Int              @id @default(autoincrement())
17   code          String?
18   movies_countries movies_countries[]
19 }
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104 /// This model contains row level security and requires additional setup for migrations. Vi
105 model movies_keywords {
106   movie_id  String  @db.Uuid
107   keyword_id Int
108   keywords   keywords @relation(fields: [keyword_id], references: [id], onDelete: Cascade, o
109   movies     movies  @relation(fields: [movie_id], references: [id], onDelete: Cascade, on
110
111   @@id([movie_id, keyword_id])
112 }
113
114 /// This model contains row level security and requires additional setup for migrations. Vi
115 model movies {
116   title        String          @db.VarChar
117   original_title String?       @db.VarChar
118   description   String?       @db.VarChar
119   release_date  String?
120   created_at    DateTime?     @default(now()) @db.Timestamptz(6)
121   updated_at    DateTime?     @default(now()) @db.Timestamptz(6)
122   language      String?       @db.VarChar
123   runtime       Decimal?     @db.Decimal
124   image_url    String?
125   id            String        @id @default(dbgenerated("gen_random_uuid()")) @db.Uuid
126   boost          Boolean?     @default(false)
127   type          String?
128   lists_movies  lists_movies[]
129   movies_countries movies_countries[]
130   movies_directors movies_directors[]
131   movies_genres   movies_genres[]
132   movies_keywords  movies_keywords[]
133 }
```

## V. Réalisation des interfaces : création et modification de film

Afin d'illustrer la mise en œuvre technique de l'application, cette section présente en détail les étapes de développement de la fonctionnalité d'ajout et de modification de films. Cette fonctionnalité mobilise plusieurs composantes du système : de l'interface utilisateur jusqu'à la gestion et la persistance des données.

### 1. Page de formulaire d'ajout de film

app/movies/create/page.tsx (extrait)

```
1  import { redirect } from "next/navigation";
2  import CreateMoviePage from "./client";
3  import { isAdmin } from "@/utils/is-user-admin";
4
5  export default async function Page() {
6    const userIsAdmin = await isAdmin(); // Auth check
7
8    if (!userIsAdmin) [
9      redirect("/login"); // Redirect to login page if not admin
10    ]
11
12    // Renders the movie form if authorized
13    return <CreateMoviePage />;
14 }
```

Dans le fichier page.tsx, j'utilise une fonction utilitaire `isAdmin()` pour vérifier si l'utilisateur est un administrateur connecté.

app/movies/create/client.tsx (extrait)

```
15  const CreateMoviePage: React.FC = () => {
16    // Initialize react-hook-form with default values
17    const {
18      register,
19      control,
20      handleSubmit,
21      setValue,
22      formState: { errors, isSubmitting },
23    } = useForm({
24      defaultValues: {
25        title: "",
26        original_title: "",
27        director_name: "",
28        description: "",
29        release_date: "",
```

J'ai créé un composant client où j'utilise react-hook-form pour la gestion des formulaires, **useState/useEffect** pour charger les données dynamiques (genres, pays, mots-clés) et **FormData** pour envoyer les données au back-end, y compris les fichiers.

```
38 |   const [countries, setCountries] = useState<any>([]);
39 |   const [genres, setGenres] = useState<any>([]);
40 |   const [keywords, setKeywords] = useState<any>([]);
41 |   const [selectedKeywords, setSelectedKeywords] = useState<any>([]);
42 |
43 |   useEffect(() => {
44 |     const fetchData = async () => {
45 |       const [genresData, countriesData, keywordsData] = await Promise.all([
46 |         getGenres(),
47 |         getCountries(),
48 |         getKeywords(),
49 |       ]);
50 |       setGenres(genresData);
51 |       setCountries(countriesData);
52 |       setKeywords(keywordsData);
53 |     };
54 |     fetchData();
55 |   }, []);
56 | }
```

Je précharge les données de manière dynamique dans la page en faisant appel à trois fonctions asynchrones pour récupérer les genres, les pays et les mots-clés. Ces données sont ensuite utilisées pour alimenter les menus déroulants (**<select>**) et le composant personnalisé **MultiSelect.ts**.

```
57 |   const onSubmit = async (data: any) => {
58 |     try {
59 |       const formData = new FormData();
60 |       formData.append("title", data.title);
61 |       formData.append("original_title", data.original_title);
62 |       formData.append("director_name", data.director_name);
63 |       formData.append("description", data.description);
64 |       formData.append("release_date", data.release_date);
65 |       formData.append("country_id", data.country_id);
66 |       formData.append("runtime", data.runtime);
67 |       formData.append("genre_id", data.genre_id);
68 |       formData.append("type", data.type);
69 |       if (selectedKeywords && selectedKeywords.length > 0) {
70 |         const keywordIds = selectedKeywords.map((k) => k.value).join(",");
71 |         formData.append("keyword_id", keywordIds);
72 |       } else {
73 |         formData.append("keyword_id", "");
74 |       }
75 |       if (data.image_url[0]) {
76 |         formData.append("image_url", data.image_url[0]);
77 |       }
78 |
79 |       const result = await addMovie(formData);
80 |
81 |       if (result.type === "success") {
82 |         router.push("/movies");
83 |       }
84 |     } catch (error) {
85 |       console.error(error);
86 |     }
87 |   }
88 | }
```

Chaque champ est relié au formulaire via `register()` (ou **Controller** pour les composants personnalisés) et possède des règles de validation (ex : champ requis, nombre minimal de caractères, etc.). À la soumission (`onSubmit`), je prépare les données dans un objet **FormData**. Cela permet de gérer l'envoi de texte et de fichiers dans une même requête. Les mots-clés sélectionnés sont convertis en une liste d'IDs, séparés par des virgules, puis ajoutés au **FormData**. Enfin, j'appelle la fonction `addMovie(formData)` pour envoyer les données au serveur. Si l'ajout est un succès, je redirige l'utilisateur vers la liste des films.

J'ai intégré une gestion des erreurs pour une meilleure expérience utilisateur : si un champ est mal rempli, un message d'erreur clair s'affiche en dessous, grâce à l'objet `errors` de `react-hook-form`. Cela permet d'éviter les soumissions incomplètes ou incorrectes.

## 2. Processus d'édition d'un film

Le processus de modification d'un film se déroule en plusieurs étapes :

1. L'utilisateur accède à la page d'un film et, s'il est administrateur, voit un bouton "Edit" (Modifier).
2. En cliquant sur ce bouton, il est redirigé vers un formulaire pré-rempli avec les données existantes du film.
3. L'utilisateur peut modifier les champs et soumettre le formulaire.
4. Les données sont validées puis enregistrées dans la base de données.
5. L'utilisateur est redirigé vers la page du film mise à jour.

### Composant d'affichage du bouton d'édition

Le bouton d'édition n'est affiché que si l'utilisateur est admin. Cette vérification est effectuée côté serveur puis transmise en prop au composant client :

`app/movies/[slug]/client.tsx`

```

165     {userIsAdmin && (
166       <div className="">
167         <Link href={`/movies/edit/${movieId}`}>
168           <button className="right-2 □bg-black bg-opacity-60 p-2 border border-rose-500">
169             <Icon icon="lucide:edit" />
170           </button>
171         </Link>
172       </div>
173     );
174   );
175 };
176 }
```

## Page d'entrée pour la modification de film

La page de modification (`app/movies/edit/[id]/page.tsx`) est un composant serveur qui récupère les données du film à modifier, vérifie l'authentification de l'utilisateur, redirige vers la page de connexion si l'utilisateur n'est pas connecté. Elle renvoie enfin le formulaire d'édition avec les données du film.

```
12
13  export default async function EditMoviePage({  
14    params,  
15  }: {  
16    params: { id: string };  
17  }) {  
18    const { movie, error } = await getMovie(params.id);  
19    const session = await auth();  
20  
21    if (!session) {  
22      redirect("/login");  
23    }  
24  
25    if (error || !movie) {  
26      return notFound();  
27    }  
28  
29    return (  
30      <div className="px-10 py-20 flex justify-start">  
31        <div className="w-full max-w-3xl">  
32          <h1 className="text-2xl font-medium text-rose-900 mb-5">  
33            <BackButton />  
34            Modifier le film  
35          </h1>  
36          <EditMovieForm movie={movie} />  
37        </div>  
38      </div>  
39    );  
40  }  
};
```

## Formulaire d'édition

Le formulaire d'édition (`app/movies/edit/[id]/edit-movie-form.tsx`) est un composant client qui gère l'état du formulaire avec React Hook Form, la gestion des sélecteurs multiples (genres, mots-clés, réalisateurs, la visualisation de l'image et la soumission du formulaire

J'utilise `useForm` pour initialiser le formulaire avec les valeurs existantes du film.

## app/movies/edit/[id]/edit-movie-form.tsx (extrait)

```
87  | const [
88  |   register,
89  |   handleSubmit,
90  |   control,
91  |   watch,
92  |   setValue,
93  |   formState: { errors },
94  | ] = useForm<FormData>({
95  |   defaultValues: {
96  |     title: movie.title,
97  |     description: movie.description || '',
98  |     release_date: movie.release_date || '',
99  |     language: movie.language || '',
100 |     type: movie.type || '',
101 |     runtime: movie.runtime || null,
102 |     image_url: movie.image_url || '',
103 |     image: null,
104 |     directors: [],
105 |     country_id:
106 |       movie.countries && movie.countries[0]
107 |         ? movie.countries[0].id.toString()
108 |         : '',
109 |     genres: [],
110 |     keywords: [],
111 |   },
112 | });

113 |
```

Au chargement du composant, je récupère toutes les données de référence (mots-clés, pays, genres, réalisateurs) et initialisons les sélections.

```
119 | useEffect(() => {
120 |   const fetchReferenceData = async () => {
121 |     // Fetch keywords
122 |     const keywords = await getKeywords();
123 |     setAvailableKeywords(
124 |       keywords.map((k) => ({
125 |         value: k.value.toString(),
126 |         label: k.label || '',
127 |       }))
128 |     );
129 |
130 |     if (movie.countries && movie.countries.length > 0) {
131 |       setValue("country_id", movie.countries[0].id.toString());
132 |     }
133 |   };
134 |
135 |   fetchReferenceData();
136 | }, [movie, setValue]);
```

Pour la sélection de multiples valeurs (genres, mots-clés, réalisateurs), j'utilise un composant **MultiSelect** personnalisé.

```

489 <div>
490   <label className="text-rose-500 block text-sm font-medium mb-1">
491     Genres
492   </label>
493   <MultiSelect
494     name="genres"
495     control={control}
496     options={availableGenres}
497     label="Genres"
498     placeholder="Chercher et ajouter des genres..."
499     onChange={(selected) => {
500       setSelectedGenres(selected);
501       setValue(
502         "genres",
503         selected.map((g) => g.value)
504       );
505     }
506     defaultValues={selectedGenres}
507   />
508   <p className="text-gray-600 text-xs mt-1">
509     Vous pouvez sélectionner plusieurs genres et en retirer.
510   </p>
511 </div>

```

La fonction `onSubmit()` traite la soumission du formulaire.

```

227 const onSubmit = async (data: FormData) => {
228   setIsSubmitting(true);
229   setError(null);
230
231   try {
232     let imageUrl = data.image_url;
233
234     if (imageFile && imageFile instanceof File) {
235       imageUrl = await uploadImage(imageFile, data.title);
236     }
237
238     const formDataToUpdate = new FormData();
239     formDataToUpdate.append("id", movie.id);
240     formDataToUpdate.append("title", data.title);
241     formDataToUpdate.append("original_title", data.original_title ?? "");
242     formDataToUpdate.append("description", data.description ?? "");
243     formDataToUpdate.append("release_date", data.release_date ?? "");
244     formDataToUpdate.append("language", data.language ?? "");
245     formDataToUpdate.append("type", data.type ?? "");
246     formDataToUpdate.append("runtime", data.runtime?.toString() ?? "");
247     formDataToUpdate.append("image_url", imageUrl);
248

```

```

248
249     if (data.image && data.image.length > 0) {
250         formDataToUpdate.append("image", data.image[0]);
251     }
252
253     formDataToUpdate.append("director_ids", JSON.stringify(data.directors));
254     formDataToUpdate.append("country_id", data.country_id);
255     formDataToUpdate.append("genre_ids", JSON.stringify(data.genres));
256     formDataToUpdate.append("keyword_ids", JSON.stringify(data.keywords));
257
258     const { success, error } = await updateMovie(formDataToUpdate);
259
260     if (success) {
261         router.push(`/movies/${movie.id}`);
262         router.refresh();
263     } else {
264         setError(error || "Something went wrong");
265     }

```

```

280     return (
281         <>
282         <form
283             onSubmit={handleSubmit(onSubmit)}
284             className="rounded-lg text-rose-900 justify-start mx-auto"
285         >
286             {error && (
287                 <div className="mb-4 p-4 bg-red-500 text-white rounded-md">
288                     {error}
289                 </div>
290             )}
291
292             <div className="flex flex-col gap-6">
293                 {/* Title */}
294                 <div className="col-span-2">
295                     <label className="block text-sm font-medium mb-1">Titre</label>
296                     <input
297                         {...register("title", {
298                             required: "Le titre est obligatoire. Au moins 1 caractère.",
299                         })}
300                         className="w-full py-2 text-sm font-light border rounded-md px-2 bg-white"
301                     />
302                     {errors.title && (
303                         <p className="text-red-500 text-sm mt-1">
304                             {errors.title.message}
305                         </p>
306                     )}
307                 </div>
308
309             <SubmitButton
310                 defaultText="Enregistrer les modifications"
311                 loadingText="Chargement..."
312                 isSubmitting={isSubmitting}
313             />
314         </div>
315     </form>
316 
```

## VI. Réalisation des services : gestion des films

Je vais maintenant aborder la partie back-end de l'application, en m'appuyant sur les Server Actions de Next.js. Ces fonctions, exécutées exclusivement côté serveur grâce à la directive "use server", me permettent de gérer des opérations sensibles comme la validation des données, les accès à la base via Prisma, ou encore l'upload de fichiers sur Supabase.

### 1. Server Action pour l'ajout d'un film

Ce server action addMovie permet à un administrateur d'ajouter un nouveau film dans la base de données via un formulaire, en validant les données saisies, en gérant l'upload d'une image sur Supabase, et en enregistrant les différentes relations (réalisateur, pays, genres, mots-clés) dans Prisma de manière transactionnelle.

```
10  const MAX_FILE_SIZE = 5000000;
11
12  const ACCEPTED_IMAGE_TYPES = [
13    "image/jpeg",
14    "image/png",
15    "image/jpg",
16    "image/webp",
17  ];
18
19  export async function addMovie(formData: FormData) {
20    const userIsAdmin = await isAdmin();
21
22    if (!userIsAdmin) {
23      return {
24        type: "error",
25        message: "You must be admin to add a movie",
26        errors: null,
27      };
28    }
29  }
```

#### Vérification des autorisations

L'action vérifie que l'utilisateur est administrateur avant de permettre l'ajout.  
Utilisation de la fonction **isAdmin()** pour contrôler les droits.

```

30  const schema = z.object({
31    title: z
32      .string()
33      .min(1, "Title is required")
34      .regex(/^[\\s\\S]*$/, "Title contains invalid characters"),
35    original_title: z
36      .string()
37      .regex(/^[\\s\\S]*$/, "Title contains invalid characters")
38      .optional(),
39    director_name: z.string().min(1, "Director is required"),
40    description: z
41      .string()
42      .min(5, "Synopsis must be at least 5 characters long"),
43    release_date: z.string().min(4, "Release year is required"),
44    runtime: z.number().min(4, "Runtime is required"),
45    country_id: z.string().min(1, "Country is required"),
46    genre_id: z.string().min(1, "Genre is required"),
47    type: z.string().min(1, "Format is required"),
48    keyword_id: z.string().min(1, "At least one keyword is required"),
49    image_url: z
50      .any()
51      .refine(
52        (file) => file?.size <= MAX_FILE_SIZE,
53        `Maximum image size is 5MB`
54      )
55      .refine(
56        (file) => ACCEPTED_IMAGE_TYPES.includes(file?.type),
57        `Only .jpg, .jpeg, .png, and .webp formats are allowed`
58      ),
59  });
60

```

## Validation des données

J'utilise Zod pour valider à nouveau les données du formulaire côté serveur, ce qui est plus fiable car il empêche les injections, les erreurs de types et les champs manquants. Il va ajouter une couche supplémentaire de sécurité au cas où la validation avec React Hook Form aurait été détournée côté client.

Contrôles sur les champs obligatoires (titre, réalisateur, description, etc.)

Validation du format et de la taille des images (max 5MB, formats jpg/png/webp)

```

101  const safeTitle = title
102    .replace(/\\s+/g, "-")
103    .replace(/[^\\w\\-]/g, "")
104    .toLowerCase();
105  const fileName = `${Math.random().toString(36).substring(2, 10)}-${safeTitle}`;
106  const supabase = createClient();
107  const { data: imageData, error: imageError } = await supabase.storage
108    .from("storage")
109    .upload(fileName, image_url, {
110      cacheControl: "3600",
111      upsert: false,
112    });
113
114  if (imageError) {
115    return {
116      type: "error",
117      message: "Database error: Failed to upload the image",
118    };
119  }
120
```

Upload de l'image vers le bucket de stockage Supabase  
Génération d'un nom de fichier unique

```

115  const result = await prisma.$transaction(async (prisma) => {
```

### Transaction de base de données avec Prisma

Utilisation de **prisma.\$transaction** pour garantir l'intégrité des données  
Les opérations sont exécutées en une seule transaction

```

121  const result = await prisma.$transaction(async (prisma) => {
122    // Upsert director
123    const director = await prisma.directors.upsert({
124      where: { id: 0 },
125      update: { name: director_name },
126      create: { name: director_name },
127    });
128
129    // Create movie
130    const movie = await prisma.movies.create({
131      data: {
132        title,
133        original_title,
134        release_date,
135        runtime,
136        type,
137        description,
138        image_url: imageData?.path,
139      },
140    });
141
```

```

141      // Insert movie countries
142      const countryIds = country_id.split(",").map(Number);
143      await prisma.movies_countries.createMany({
144        data: countryIds.map((countryId) => ({
145          movie_id: movie.id,
146          country_id: countryId,
147        })),
148      });
149    );
150
151    // Link director to movie
152    await prisma.movies_directors.create({
153      data: {
154        movie_id: movie.id,
155        director_id: director.id,
156      },
157    });
158  );

```

## Opérations de base de données

Création/mise à jour du réalisateur (**upsert**)

Création de l'entrée du film

Association avec les pays de production, le réalisateur, les genres et les mots-clés (toutes des relations many-to-many).

## Gestion des relations complexes

Traitement des champs qui acceptent plusieurs valeurs (pays, genres, mots-clés)

Conversion des chaînes séparées par des virgules en tableaux d'IDs

```

183
184    return movie;
185  );
186 } catch (error) {
187   console.error("Error", error);
188   return {
189     type: "error",
190     message: "Database error: Failed to add the movie",
191   };
192 }
193
194 revalidatePath("/");
195 redirect("/");
196 }

```

## Redirection et revalidation

Utilisation de **revalidatePath("/")** pour rafraîchir les données affichées

Redirection vers la page d'accueil après ajout réussi

## 2. Server Action pour la mise à jour des données d'un film

Cette fonction permet à un administrateur de mettre à jour les informations d'un film dans la base de données, y compris les relations associées (réaliseurs, pays, genres, mots-clés) et l'image, si elle est modifiée.

### app/server-actions/movies/update-movie.ts (extrait)

```
9  const movieUpdateSchema = z.object({
10    id: z.string().min(1, { message: "ID missing." }),
11    title: z.string().min(1, { message: "Title is required." }),
12    original_title: z.string().nullable().optional(),
13    description: z.string().min(1, { message: "Description is required." }),
14    release_date: z.string().min(1, { message: "Date is required." }),
15    language: z.string().min(1, { message: "Language is required." }),
16    type: z.string().min(1, { message: "Format is required." }),
17    runtime: z
18      .string()
19      .nullable()
20      .optional()
21      .transform((val) => (val ? Number(val) : null))
22      .refine((val) => val === null || !isNaN(val), {
23        message: "Runtime must be a number.",
24      }),
25    image_url: z.string().min(1, { message: "Image url is required." }),
26
27    director_ids: z
28      .string()
29      .transform((val) => JSON.parse(val))
30      .refine((val) => Array.isArray(val) && val.length > 0, {
31        message: "Select at least one director minimum.",
32      }),
33
34    country_id: z.string().min(1, { message: "Country is required." }),
35  }
```

### Validation des données avec Zod

Le schéma movieUpdateSchema valide avec Zod et transforme strictement les données du formulaire entrant sur le serveur, en s'assurant que chaque champ obligatoire (comme le titre, la description, la date de sortie, etc.) est correctement tapé, formaté et présent avant d'effectuer toute opération sur la base de données.

```
7
8  export async function updateMovie(formData: FormData) {
9    const supabase = createClient();
10   const userIsAdmin = await isAdmin();
11
12   if (!userIsAdmin) {
13     return { type: "error", message: "You must be admin to update a movie" };
14   }
15 }
```

## Vérification des droits d'accès

Seuls les utilisateurs ayant le rôle d'administrateur peuvent mettre à jour un film. Si ce n'est pas le cas, la fonction retourne une erreur immédiatement.

```
18 | try {
19 |   const id = formData.get("id") as string;
20 |   const title = formData.get("title") as string;
21 |   const description = formData.get("description") as string;
22 |   const release_date = formData.get("release_date") as string;
23 |   const language = formData.get("language") as string;
24 |   const type = formData.get("type") as string;
25 |   const runtime = formData.get("runtime")
26 |     ? Number(formData.get("runtime"))
27 |     : null;
28 |   const image_url = formData.get("image_url") as string;
29 |   const image = formData.get("image") as File | null;
30 |   const director_ids = JSON.parse(
31 |     formData.get("director_ids") as string
32 |   ) as string[];
33 |   const country_id = formData.get("country_id") as string;
34 |   const genre_ids = JSON.parse(
```

## Récupération et préparation des données du formulaire

Tous les champs nécessaires à la mise à jour sont extraits de l'objet FormData : titre, description, langue, type, date de sortie, durée, image, etc.

Certaines données sont transformées (ex. : JSON.parse pour les tableaux d'IDs, conversion en nombre pour runtime).

```
40 |
41 | let imageUrl = image_url;
42 |
43 | // Upload the image if provided
44 | if (image) {
45 |   const filename = `${Date.now()}-${image.name.replace(/\s+/g, "-")}`;
46 |   const { data, error } = await supabase.storage
47 |     .from("storage")
48 |     .upload(filename, image, { cacheControl: "3600", upsert: true });
49 |
50 |   if (error) {
51 |     throw new Error(error.message);
52 |   }
53 |   imageUrl = `${data?.path}`;
54 | }
```

## Téléversement d'une nouvelle image (si fournie)

Si une nouvelle image est envoyée, elle est uploadée sur Supabase Storage. En cas d'erreur, une exception est levée pour interrompre le processus.

```

55
56   await prisma.movies.update({
57     where: { id },
58     data: [
59       title,
60       description,
61       release_date,
62       language,
63       type,
64       runtime,
65       imageUrl: imageUrl,
66       updated_at: new Date(),
67     ],
68   });

```

### Mise à jour du film principal dans la table movies

Le film est mis à jour avec les nouvelles données, y compris `updated_at` pour indiquer une modification récente.

```

71   await prisma.movies_directors.deleteMany({ where: { movie_id: id } });
72   for (const directorId of director_ids) {
73     await prisma.movies_directors.create({
74       data: { movie_id: id, director_id: BigInt(directorId) },
75     });
76   }
77
78   await prisma.movies_countries.deleteMany({ where: { movie_id: id } });
79   await prisma.movies_countries.create({
80     data: { movie_id: id, country_id: parseInt(country_id) },
81   );
82
83   await prisma.movies_genres.deleteMany({ where: { movie_id: id } });
84   for (const genreId of genre_ids) {
85     await prisma.movies_genres.create({
86       data: { movie_id: id, genre_id: BigInt(genreId) },
87     });
88   }

```

### Mise à jour des relations (nettoyage puis réinsertion)

Chaque relation est d'abord supprimée pour éviter les doublons ou incohérences, puis **recréée** avec les nouvelles valeurs fournies.

```

97   revalidatePath(`/movies/${id}`);
98   return { success: true };
99 }
100 catch (error) {
101   console.error("Error updating movie:", error);
102   return {
103     success: false,
104     error: error instanceof Error ? error.message : "An error occurred",
105   };

```

### Revalidation du cache côté Next.js

Cette ligne permet à Next.js de rafraîchir la page du film concerné afin que les nouvelles données soient visibles immédiatement.

### Déconnexion de Prisma

La connexion à la base de données est fermée, quelle que soit l'issue de la fonction.

### 3. Server Action pour la suppression d'un film

Cette fonction asynchrone permet à l'administrateur de supprimer un film de la base de données ainsi que toutes ses relations associées, de manière sécurisée.

#### app/server-actions/movies/delete-movie.ts (extrait)

```
6
7  export async function deleteMovie(movieId: string) {
8    const userIsAdmin = await isAdmin();
9
10   if (!userIsAdmin) {
11     return {
12       type: "error",
13       message: "You must be admin to delete a movie",
14       errors: null,
15     };
16   }
17 }
```

#### Vérification des droits d'accès

Seuls les utilisateurs administrateurs peuvent effectuer cette action.

Si l'utilisateur n'est pas admin, la fonction retourne une erreur immédiatement.

```
17  try {
18    const movie = await prisma.movies.findUnique({
19      where: { id: movieId },
20    });
21
22    if (!movie) {
23      return { success: false, message: "Movie doesn't exist" };
24    }
25  }
```

#### Vérification de l'existence du film

Avant toute suppression, on s'assure que le film existe.

Si aucun film correspondant n'est trouvé, on retourne un message d'échec.

```
26  await prisma.movies_countries.deleteMany({ where: { movie_id: movieId } });
27  await prisma.movies_directors.deleteMany({ where: { movie_id: movieId } });
28
29
30
31
32
33
34
```

#### Suppression des relations associées au film

On supprime toutes les entrées liées au film dans les tables de relations. Cela évite toute erreur de contrainte étrangère au moment de supprimer le film lui-même.

```
30
31  await prisma.movies.delete({ where: { id: movieId } });
32  revalidatePath("/");
33  return { success: true, message: "Success to delete the movie" };
34 }
```

#### Suppression du film principal

Après les dépendances supprimées, le film peut être supprimé de la table **movies**.

#### Revalidation du cache côté client

Permet de rafraîchir la page d'accueil pour refléter la suppression en temps réel.

## VII. Sécurité

Conformément au Règlement Général sur la Protection des Données (RGPD), l'application a été conçue de manière à respecter la vie privée des utilisateurs dès la conception. Aucune donnée sensible (au sens du RGPD telle que l'orientation sexuelle, les opinions politiques, la santé, etc.) n'est traitée ou collectée. Seules des données personnelles sont utilisées, nécessaires à l'authentification des utilisateurs et au bon fonctionnement de l'application (e-mail, ID utilisateur).

J'ai rédigé une politique de confidentialité que j'ai mise en ligne et rendue accessible via le footer de l'application. Elle décrit les données collectées, les droits des utilisateurs (accès, rectification, suppression), ainsi que les modalités d'exercice de ces droits via le formulaire de contact. Par ailleurs, j'ai rédigé un registre des traitements de données pour documenter précisément les finalités, les catégories de données personnelles traitées (adresse e-mail uniquement), les destinataires, les durées de conservation et les garanties apportées en matière de sécurité.

*Voir **Annexe 10** pour le registre des traitements de données*

### 1. Authentification et autorisation avec Supabase Auth

L'authentification est gérée par Supabase Auth pour la gestion des utilisateurs. Un middleware Next.js est implémenté pour assurer que chaque requête est correctement authentifiée et autorisée. L'architecture d'authentification repose sur quatre couches principales selon la documentation de Supabase :

1. Couche Client : SDKs Supabase ou requêtes HTTP manuelles
2. Passerelle API Kong : partagée entre tous les produits Supabase
3. Service Auth (anciennement GoTrue) : serveur d'API d'authentification
4. Base de données PostgreSQL : stockage des données utilisateur

L'implémentation dans mon application comprend :

- Clients Supabase : différents clients pour les contextes serveur, navigateur et middleware
- Middleware d'authentification : gère la persistance de session et la protection des routes
- Server Actions : gère les opérations d'authentification côté serveur
- Synchronisation avec base de données : maintient la cohérence entre Supabase Auth et Prisma

Le processus d'authentification suit ces étapes :

1. L'utilisateur accède à l'application
2. Le middleware vérifie la présence d'un cookie de session valide
3. Si l'utilisateur n'est pas authentifié et tente d'accéder à une route protégée, il est redirigé vers la page de connexion
4. L'utilisateur soumet ses identifiants via un formulaire
5. Les Server Actions traitent la demande et communiquent avec Supabase
6. Après authentification réussie, Supabase émet des cookies de session sécurisés
7. L'utilisateur est redirigé vers la page demandée
8. Le middleware synchronise les données de l'utilisateur entre Supabase et Prisma

## 2. Sécurité des identifiants

### 1. Protection des mots de passe par Supabase Auth

Supabase Auth gère automatiquement la sécurité des mots de passe sans nécessiter d'implémentation manuelle du hash, du salt, ou de la gestion des sessions. Le service repose sur Service Auth (GoTrue), un fournisseur d'identité open-source qui applique les standards de sécurité suivants :

#### **Hash de mot de passe sécurisé :**

- Utilisation de l'algorithme bcrypt, reconnu comme norme sécurisée par l'OWASP
- Hash côté serveur avec un salt aléatoire unique pour chaque mot de passe
- Aucune copie du mot de passe en clair n'est conservée.

#### **Protection contre les attaques par force brute :**

- Limitation du nombre de tentatives de connexion.
- Verrouillage temporaire de compte après plusieurs échecs consécutifs.

#### **Stockage sécurisé :**

- Les hachages sont stockés dans une base de données PostgreSQL sécurisée, sans jamais transmettre ni stocker le mot de passe d'origine.

### 2. Supabase et cookies sécurisés

À chaque connexion réussie, Supabase Auth crée un cookie sécurisé avec les attributs suivants :

- **HttpOnly** : prévient l'accès via JavaScript côté client

- **SameSite** : protection contre les attaques CSRF
- **Secure** : transmission uniquement via HTTPS en production

Ce cookie contient un **JWT (JSON Web Token)** qui identifie la session utilisateur et est utilisé pour vérifier l'identité dans le middleware, les Server Actions et les appels client.

### 3. Middleware d'authentification et protection des routes

Le middleware (**/middleware.ts**) joue un rôle central dans le système d'authentification et accomplit plusieurs fonctions critiques :

- Persistance de session : gère les cookies de session entre les requêtes
- Protection des routes : restreint l'accès aux routes privées pour les utilisateurs non authentifiés
- Synchronisation de base de données : maintient la cohérence entre Supabase Auth et la base de données Prisma
- Gestion d'état utilisateur : fournit le contexte utilisateur à l'application

```

5  export async function updateSession(request: NextRequest) {
6    let supabaseResponse = NextResponse.next({
7      request,
8    });
9
10   const supabase = createServerClient(
11     process.env.NEXT_PUBLIC_SUPABASE_URL!,
12     process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY!,
13     {
14       cookies: {
15         getAll() {
16           return request.cookies.getAll();
17         },
18         setAll(cookiesToSet) {
19           cookiesToSet.forEach(({ name, value, options }) =>
20             request.cookies.set(name, value)
21           );
22           supabaseResponse = NextResponse.next({
23             request,
24           });
25           cookiesToSet.forEach(({ name, value, options }) =>
26             supabaseResponse.cookies.set(name, value, options)
27           );
28         },
29       },
30     }
31   );
32
33   const {
34     data: { user },
35   } = await supabase.auth.getUser();

```

La synchronisation avec Prisma garantit que chaque utilisateur authentifié via Supabase possède une entrée correspondante dans la base de données PostgreSQL. Cette approche permet de lier les données métier (listes de films) à l'utilisateur, maintenir la cohérence des données entre les deux systèmes et permettre des requêtes complexes utilisant les relations Prisma.

```
42  if (user) {
43    try {
44      // Check if the user already exist in Prisma
45      const existingUser = await prisma.users.findUnique({
46        where: { id: user.id },
47      });
48      // If doesn't exist, create the user
49      if (!existingUser) {
50        await prisma.users.create({
51          data: {
52            id: user.id,
53            email: user.email!,
54            role: "user",
55            created_at: new Date(),
56          },
57        });
58      }
59    } catch (err) {
60      console.error("Error during user synchronization:", err);
61    }
62  }
63  if (
64    !user &&
65    !request.nextUrl.pathname.startsWith("/") &&
66    !request.nextUrl.pathname.startsWith("/login") &&
67    !request.nextUrl.pathname.startsWith("/signup") &&
68    !request.nextUrl.pathname.startsWith("/auth")
69  ) {
70    const url = request.nextUrl.clone();
71    url.pathname = "/login";
72    return NextResponse.redirect(url);
73  }
74
75  return supabaseResponse;
76}

92
93  export async function middleware(request: NextRequest) {
94    return await updateSession(request);
95  }
96
97  export const config = {
98    matcher: [
99      "/((?!_next/static|_next/image|favicon.ico|public|signup|stats|about|movies|catalog).*)",
100    ],
101  };

```

## 4. Implémentation des opérations d'authentification

Les opérations d'authentification sont gérées par des Server Actions ([app/login/actions.ts](#)) pour traiter les informations d'identification. Les identifiants ne transitent jamais par le navigateur en clair. Les données d'authentification sont validées sur le serveur. Les erreurs sont gérées de manière centralisée. Le cache est ensuite régénéré avec l'invalidation automatique après modification de l'état d'authentification.

```
8  export async function login(formData: FormData) {
9    const supabase = createClient();
10
11    const data = {
12      email: formData.get("email") as string,
13      password: formData.get("password") as string,
14    };
15
16    const { error } = await supabase.auth.signInWithEmailAndPassword(data);
17
18    if (error) {
19      redirect("/error");
20    }
21
22    revalidatePath("/", "layout");
23    redirect("/");
24  }
25
26  export async function signup(formData: FormData) {
27    const supabase = createClient();
28
29    const data = {
30      email: formData.get("email") as string,
31      password: formData.get("password") as string,
32    };
33
34    const { error } = await supabase.auth.signUp(data);
35
36    if (error) {
37      redirect("/error");
38    }
39
40    revalidatePath("/", "layout");
41    return { success: true, message: "Confirmation email sent" };
42  }
```

## 5. Vérification d'identité et confirmation par mail

J'ai implémenté un processus de vérification d'identité par email ([app/auth/confirm/route.ts](#)) qui reçoit un token de vérification via URL, vérifie la

validité du token auprès de Supabase, confirme l'identité de l'utilisateur si le token est valide et redirige l'utilisateur vers la page d'accueil connecté.

```
7  export async function GET(request: NextRequest) {
8    const { searchParams } = new URL(request.url);
9    const token_hash = searchParams.get("token_hash");
10   const type = searchParams.get("type") as EmailOtpType | null;
11   const next = searchParams.get("next") ?? "/";
12
13   if (token_hash && type) {
14     const supabase = createClient();
15     const { error } = await supabase.auth.verifyOtp({
16       type,
17       token_hash,
18     });
19     if (!error) {
20       redirect(next);
21     }
22   }
23   redirect("/error");
24 }
```

## 6. Gestion des permissions et des rôles

L'architecture d'authentification repose sur un système RBAC (Role-Based Access Control), qui permet de gérer l'accès aux fonctionnalités selon le rôle attribué à chaque utilisateur (user ou admin). Ce système est appliqué à différents niveaux de l'application :

- Un middleware Next.js intercepte les requêtes entrantes et bloque l'accès aux routes protégées si l'utilisateur n'est pas authentifié.
- Une fonction **isAdmin()** permet de vérifier si l'utilisateur connecté possède les priviléges administrateur.
- Avant d'exécuter des actions sensibles (comme la suppression d'un film), des vérifications sont effectuées pour s'assurer que l'utilisateur dispose des autorisations nécessaires.
- Certains composants ne sont visibles que si l'utilisateur a les droits requis.

### Synchronisation des utilisateurs

Pour garantir la cohérence entre le système d'authentification (Supabase) et la base de données interne (via Prisma), j'ai intégré une fonction **ensureUserExists()**. Elle permet de créer automatiquement l'utilisateur dans la base de données s'il n'existe pas encore, à la connexion.

## utils/ensure-user-exist.ts

```
5
6  export async function ensureUserExists() {
7    const supabase = createClient();
8    const { data, error } = await supabase.auth.getUser();
9
10   if (error || !data?.user) {
11     return { success: false, message: "User not authenticated" };
12   }
13
14   try {
15     // Attempts to create the user, silently ignores if it already exists
16     await prisma.users.upsert({
17       where: { id: data.user.id },
18       update: {}, // Does not update any existing fields
19       create: {
20         id: data.user.id,
21         email: data.user.email!,
22         created_at: new Date(),
23       },
24     });
25
26     return { success: true, message: "User synchronized" };
27   } catch (err) {
28     console.error("Error when user synchronization:", err);
29     return { success: false, message: "Erreur de synchronisation" };
30   }
31 }
```

Cette logique est utilisée pour éviter les duplications d'utilisateurs, garantir que tous les utilisateurs Supabase ont un équivalent dans la base interne et préparer le terrain pour d'éventuelles relations (listes de films, notamment).

J'ai également intégré une fonctionnalité de suppression de compte qui, pour l'instant, permet de supprimer les données personnelles de l'utilisateur de manière irréversible (listes créées, nom et adresse mail). Le user n'existe plus dans la table users. Cependant pour une suppression totale du user, la procédure doit être effectué par l'admin.

## Vérification d'authentification

La fonction utilitaire **auth()** dans **utils/auth.ts** permet de savoir si un utilisateur a une session active, c'est-à-dire s'il est connecté. Elle est utilisée dans de nombreuses parties de l'application pour valider l'identité de l'utilisateur avant d'exécuter une logique sensible.

```

5  export async function auth() {
6    const supabase = createClient();
7
8    const {
9      data: { session },
10     } = await supabase.auth.getSession();
11
12    return session;
13  }

```

## Vérification du rôle de l'utilisateur

Pour compléter le système de contrôle d'accès, deux fonctions utilitaires permettent d'identifier le rôle de l'utilisateur connecté.

### utils/is-user-admin.ts

```

6  export async function getCurrentUserRole(): Promise<UserRole | null> {
7    const supabase = createClient();
8    const {
9      data: { user },
10     } = await supabase.auth.getUser();
11
12    if (!user) return null;
13
14    const dbUser = await prisma.users.findUnique({
15      where: { id: user.id },
16      select: { role: true },
17    });
18
19    return dbUser?.role as UserRole | null;
20  }
21
22  // check if the user is admin
23  export async function isAdmin(): Promise<boolean> {
24    const role = await getCurrentUserRole();
25    return role === "admin";
26  }

```

La fonction **getCurrentUserRole()** permet de récupérer le rôle (user ou admin) de l'utilisateur actuellement authentifié, en croisant les données de Supabase et de la base de données via Prisma.

- Elle commence par récupérer l'utilisateur connecté via Supabase.
- Ensuite, elle interroge la base de données (**prisma.users**) pour retrouver le rôle associé à l'ID de cet utilisateur.
- Le rôle est retourné typé comme UserRole (un type défini dans l'application pour sécuriser les rôles possibles).

La fonction **isAdmin()** s'appuie sur **getCurrentUserRole()** pour vérifier si l'utilisateur possède le rôle admin. Elle est utilisée pour sécuriser des routes ou des Server Actions critiques, pour conditionner l'affichage de l'interface d'administration et dans le middleware pour restreindre l'accès à certaines sections du site.

## 7. Alignement avec les recommandations d'OWASP

L'**OWASP** (Open Worldwide Application Security Project) propose des ressources et des recommandations pour améliorer la sécurité des applications web. Le OWASP Top 10 liste 10 vulnérabilités les plus critiques. Parmi ces vulnérabilités, certaines concernent la gestion des utilisateurs et l'authentification, par exemple :

- A01 - Contrôle d'accès défaillant
- A02 - Cryptographie défaillante
- A07 - Identification et authentification défaillantes

Mon application prend en compte ces bonnes pratiques grâce à Supabase Auth qui implémente par défaut plusieurs mécanismes de sécurité recommandés par l'OWASP.

Recommandation OWASP	Application à mon projet
<b>Stockage sécurisé des mots de passe</b>	Supabase utilise bcrypt + salt, comme recommandé par l'OWASP
<b>Gestion de session sécurisée</b>	Supabase génère des JWT sécurisés, gérés via cookies HttpOnly
<b>Contrôle d'accès basé sur les rôles</b>	Implémenté avec une logique RBAC : user/admin, via Supabase + Prisma
<b>Protection contre les injections SQL</b>	Prisma ORM avec requêtes typées : protège contre les injections
<b>Vérification par email (MFA simple)</b>	Email de confirmation à l'inscription
<b>Protection des routes sensibles</b>	Middleware Next.js + vérification de session active et rôle utilisateur

## VIII. Déploiement

### 1. Architecture de conteneurisation et orchestration avec Docker Compose

J'ai configuré l'environnement de développement pour maximiser la productivité. Docker est utilisé pour conteneurer l'application et ses dépendances. C'est un outil essentiel pour assurer la cohérence s'il y a plusieurs développeurs qui travaillent sur le projet. La conteneurisation permet d'isoler l'application et ses dépendances dans des environnements reproductibles.

Le Dockerfile utilise une image Node.js 18 Alpine légère avec pnpm comme gestionnaire de paquets, copie les fichiers Prisma pour la génération du client, et expose le port 3000 pour le serveur de développement Next.js.

*Voir Annexe 9 pour le Dockerfile*

Le fichier **compose.yaml** orchestre deux services principaux : l'application Next.js et une base de données PostgreSQL locale. Le service **app** monte le code source en volume pour le développement en temps réel, injecte les variables d'environnement depuis le fichier **.env**, et dépend du service **db**. Le service **db** utilise PostgreSQL 15 Alpine avec persistance des données via un volume Docker nommé **db-data**, garantissant que les données survivent aux redémarrages de conteneurs. Cette configuration permet un environnement de développement complet avec **docker-compose up -d**.

*Voir Annexe 10 pour le fichier compose.yaml*

### 2. Intégration Supabase et gestion des données

Supabase Studio est configuré via **config.toml** pour fournir un environnement de développement local complet incluant l'API REST (port 54331), la base de données PostgreSQL (port 54332), l'interface d'administration Studio (port 54338), et les services en temps réel. Cette configuration permet de développer et tester localement avant le déploiement. La synchronisation entre Prisma et Supabase s'effectue via les commandes **npx prisma migrate dev** pour générer les migrations et **supabase db push** pour synchroniser le schéma, permettant une gestion cohérente des données entre les deux ORM.

### 3. Processus de déploiement

Le déploiement suit un pipeline CI/CD automatisé : chaque merge sur la branche `main` déclenche un déploiement sur Vercel vers <https://queercinema.fr/>. Le processus inclut l'installation des dépendances avec `pnpm install`, l'exécution des tests unitaires Jest et end-to-end Playwright, la génération du build de production avec `pnpm build`, et l'application des migrations Prisma avec `npx prisma migrate deploy`. Les variables d'environnement sont configurées sur Vercel (`DATABASE_URL`, `NEXT_PUBLIC_SUPABASE_URL`, `NEXT_PUBLIC_SUPABASE_ANON_KEY`, `NEXT_PUBLIC_SUPABASE_SERVICE_ROLE_KEY`) et GitHub Actions (`PLAYWRIGHT_USER_EMAIL`, `PLAYWRIGHT_USER_PASSWORD`) pour assurer la continuité entre les environnements de développement et de production.

Le fichier `.env.example` est une réplique du `.env` ou `.env.local`, mais sans les valeurs sensibles, uniquement les clés. Il est utile comme guide pour créer son propre fichier `.env` ou `.env.local`. Voici mon modèle qui comprend les types de variables que j'utilise dans mon `.env` et `.env.local` :

```
1
2 # Database Supabase
3 DATABASE_URL=
4 DIRECT_URL=
5
6 # Database Supabase Key
7 NEXT_PUBLIC_SUPABASE_URL=
8 NEXT_PUBLIC_SUPABASE_ANON_KEY=
9 NEXT_PUBLIC_SUPABASE_SERVICE_ROLE_KEY=
10
11 # Playwright Test end-to-end
12 PLAYWRIGHT_TEST_BASE_URL=
13 PLAYWRIGHT_USER_EMAIL=
14 PLAYWRIGHT_USER_PASSWORD=
15
16 # Contact form Resend Api
17 RESEND_API_KEY=
```

## VI. Tests

### I. Tests unitaires

Les tests unitaires permettent de tester isolément les plus petites unités de code, comme des fonctions ou des méthodes, afin de vérifier qu'elles se comportent correctement. Cela m'aide à détecter et corriger les erreurs, tout en documentant le comportement attendu. Ainsi, mon code devient plus fiable et maintenable. J'ai choisi d'utiliser Jest, un framework de tests JavaScript populaire. Chaque test est exécuté dans un environnement isolé. Jest permet de créer facilement des mocks pour simuler le comportement de dépendances externes.

#### 1. Test de la fonction addMovie

J'ai rédigé une suite de tests unitaires pour vérifier la fiabilité de la fonction addMovie. Elle couvre plusieurs cas : le refus d'accès si l'utilisateur n'est pas administrateur, la validation des données du formulaire, l'ajout réussi d'un film avec stockage de l'image via Supabase, la gestion d'un échec de téléchargement d'image, ainsi que la gestion des erreurs provenant de la base de données Prisma. Les dépendances critiques comme l'authentification, le stockage et la base de données sont simulées via des mocks pour isoler la logique métier de la fonction.

```
87  it("should add a successful film", async () => {
88    isAdmin as jest.Mock).mockResolvedValue(true);
89
90    const mockSupabase = {
91      storage: {
92        from: jest.fn().mockReturnValue({
93          upload: jest.fn().mockResolvedValue({
94            data: { path: "path/to/image.jpg" },
95            error: null,
96          }),
97        }),
98      },
99    };
100   (createClient as jest.Mock).mockReturnValue(mockSupabase);
101
102  const mockMovie = { id: 1, title: "Test Movie" };
103  const mockDirector = { id: 1, name: "Test Director" };
104
```

```

105  (prisma.$transaction as jest.Mock).mockImplementation(async (callback) => {
106    (prisma.directors.upsert as jest.Mock).mockResolvedValue(mockDirector);
107    (prisma.movies.create as jest.Mock).mockResolvedValue(mockMovie);
108    (prisma.movies_countries.createMany as jest.Mock).mockResolvedValue({
109      count: 1,
110    });
111    (prisma.movies_directors.create as jest.Mock).mockResolvedValue({
112      id: 1,
113    });
114    (prisma.movies_genres.createMany as jest.Mock).mockResolvedValue({
115      count: 1,
116    });
117    (prisma.movies_keywords.createMany as jest.Mock).mockResolvedValue({
118      count: 1,
119    });
120
121    return await callback(prisma);
122  });
123
124  const formData = new FormData();
125  const mockFile = new File(["dummy content"], "test.jpg", {
126    type: "image/jpeg",
127  });
128
129  formData.append("title", "Test Movie");
130  formData.append("original_title", "Test Original Movie");
131  formData.append("director_name", "Test Director");
132  formData.append("description", "Test description for the movie");
133  formData.append("release_date", "2023");
134  formData.append("runtime", "120");
135  formData.append("country_id", "1");
136  formData.append("genre_id", "2");
137  formData.append("type", "feature");
138  formData.append("keyword_id", "3,4");
139  formData.append("image_url", mockFile);
140
141  await addMovie(formData);
142
143  expect(isAdmin).toHaveBeenCalled();
144  expect(createClient).toHaveBeenCalled();
145  expect(mockSupabase.storage.from).toHaveBeenCalledWith("storage");
146  expect(prisma.$transaction).toHaveBeenCalled();
147  expect(revalidatePath).toHaveBeenCalledWith("/");
148  expect(redirect).toHaveBeenCalledWith("/");
149});

```

## 2. Test de updateMovie

Dans cette suite de tests, j'ai vérifié le bon fonctionnement de la fonction `updateMovie`, qui permet la mise à jour des informations d'un film dans la base de données. J'ai commencé par m'assurer que l'accès à cette fonctionnalité est bien restreint aux administrateurs en simulant différents cas avec la fonction `isAdmin`. J'ai ensuite testé la mise à jour d'un film avec et sans téléchargement d'une nouvelle image via Supabase, en couvrant les cas de succès comme d'échec (upload échoué, erreurs en base). J'ai fait attention à vérifier que toutes les relations (réaliseurs, pays, genres, mots-clés) sont supprimées et recréées proprement lors de la mise à jour. Enfin, j'ai validé que la fonction invalide bien le cache avec

revalidatePath et ferme correctement la connexion Prisma avec \$disconnect. Tous les comportements clés ont été isolés et simulés grâce à des mocks.

```
144  it("devrait mettre à jour un film avec upload d'image", async () => {
145    // Mock isAdmin pour retourner true
146    (isAdmin as jest.Mock).mockResolvedValue(true);
147
148    // Mock Supabase storage upload
149    const mockSupabase = {
150      storage: {
151        from: jest.fn().mockReturnValue({
152          upload: jest.fn().mockResolvedValue({
153            data: { path: "new-image-path.jpg" },
154            error: null,
155          }),
156        }),
157      },
158    };
159    (createClient as jest.Mock).mockReturnValue(mockSupabase);
160
161    // Mock PrismaClient movies.update
162    mockPrismaClient.movies.update.mockResolvedValue({
163      id: "123",
164      title: "Updated Movie Title",
165    });
166
167    const result = await updateMovie(formData);
168
169    // Vérifier que Supabase a été appelé pour l'upload
170    expect(createClient).toHaveBeenCalled();
171    expect(mockSupabase.storage.from).toHaveBeenCalledWith("storage");
172    expect(mockSupabase.storage.from().upload).toHaveBeenCalledWith(
173      expect.stringContaining("newimage.jpg"),
174      mockFile,
175      { cacheControl: "3600", upsert: true }
176    );
177
178    // Vérifier que l'URL de l'image mise à jour est utilisée
179    expect(mockPrismaClient.movies.update).toHaveBeenCalledWith(
180      where: { id: "123" },
181      data: expect.objectContaining({
182        image_url: "new-image-path.jpg",
183      }),
184    );
185
186    expect(result).toEqual({ success: true });
187  });
188
```

Lorsque j'exécute `pn test`, le rapport affiche que les deux fichiers de tests (add-movie.test.ts et update-movie.test.ts) ont passé tous leurs tests (10/10), ce qui indique que toutes les assertions attendues sont respectées. Les tests sont passés parce que chaque cas d'usage, chaque dépendance, et chaque erreur potentielle ont été correctement simulés, testés, et validés par Jest.

J'ai également automatisé les tests avec un fichier `jest.yml` pour l'ajouter à l'intégration continue. Ainsi les tests sont lancés à chaque nouvelle Pull Request, dès que je push une nouvelle version. Cela vérifie si mes modifications ont eu un impact sur le bon fonctionnement des fonctions.

## II. Tests end-to-end

Les tests End-to-End valident le bon fonctionnement de l'application en simulant un parcours utilisateur complet. Playwright est un framework E2E qui permet d'automatiser ces tests sur différents navigateurs, avec ou sans interface graphique. Il facilite le débogage et gère automatiquement les événements complexes (réseau, redirections,...). Il permet de reproduire de manière fidèle le comportement d'un utilisateur réel.

### 1. Test de connexion d'un utilisateur

```
3  dotenv.config();
4
5
6  const email = process.env.PLAYWRIGHT_USER_EMAIL as string;
7  const password = process.env.PLAYWRIGHT_USER_PASSWORD as string;
8
9  test("login", async ({ page }) => {
10    await page.goto("/login");
11    await page.setViewportSize({ width: 1280, height: 720 });
12
13    await page.getByTestId("email-input").fill(email);
14    await page.getByTestId("password-input").fill(password);
15
16    await page.getByTestId("login-submit-button").click();
17
18    await page.waitForURL("/", { timeout: 10000 });
19
20    const mobileMenu = page.getByTestId("user-menu-trigger-mobile");
21    const desktopMenu = page.getByTestId("user-menu-trigger-desktop");
22
23    if (await mobileMenu.isVisible()) {
24      await mobileMenu.click();
25    } else {
26      await desktopMenu.click();
27    }
28    await expect(page.getByTestId("my-lists-menu-item")).toBeVisible();
29
30    await expect(page.getByTestId("logout-button")).toBeVisible();
31  });
}
```

Ce test vérifie le bon fonctionnement du processus de connexion d'un utilisateur. Il commence par la navigation vers la page de login, où les identifiants sont saisis à l'aide de variables d'environnement. Une fois le formulaire soumis, le test s'assure que l'utilisateur est redirigé vers la page d'accueil. Il vérifie ensuite la présence des éléments du menu utilisateur (version mobile ou desktop) et s'assure que les options "Mes listes" et "Déconnexion" sont bien visibles, confirmant ainsi que l'authentification a réussi.

## 2. Automatisation des tests

En plus des tests en local, j'ai ajouté un fichier playwright.yml qui est un script de configuration pour GitHub Actions qui permet l'exécution automatisée des tests End-to-End (E2E) avec Playwright à chaque mise à jour du code. Les tests sont lancés automatiquement lors d'un push ou d'une pull request. GitHub Actions utilise un environnement Ubuntu récent (ubuntu-latest) pour exécuter les tests. Les tests sont exécutés en utilisant les variables d'environnement avec l'URL de l'application déployée et les identifiants fournis depuis les secrets GitHub pour l'authentification. Les rapports de tests sont générés et stockés sous le nom playwright-report.

Certains tests lancés en local fonctionnent (login et catalogue) mais aucun n'aboutit encore dans le cadre de l'intégration continue que j'ai mise en place. J'ai ajouté des data-testid pour permettre au test d'avoir une localisation stable du composant sur lequel interagir. J'aimerais à terme mettre en plus une meilleure couverture de test pour l'expérience utilisateur et notamment concernant la création et modification de film, ainsi que pour les listes de films.

## III. Tests d'accessibilité, de performance et SEO

Je vérifie régulièrement avec l'outil d'analyse Lighthouse de Google la performance, l'accessibilité et le référencement de chaque page. Il me sert de guide pour cibler les éléments problématiques tout en proposant des moyens d'amélioration et d'optimisation. Voici les résultats d'un test pour la page d'accueil lancé le 04/07/2025 (mobile à gauche et bureau à droite) :



## IV. Tests utilisateurs

Les tests utilisateurs représentent une étape essentielle dans le processus de validation de l'application. Ils permettent de recueillir des retours concrets sur l'expérience réelle des utilisateurs, en identifiant les points forts, les éventuels blocages, et les aspects à améliorer. Au bout de 6 mois de développement, pour

évaluer au mieux l'ergonomie et la fluidité de l'application, j'ai conçu un questionnaire (*voir Annexe 11*) que j'ai distribué auprès de 22 participants.

Les questions ont été pensées pour explorer chaque étape clé de l'expérience utilisateur : de l'accueil sur le site à la navigation, en passant par la recherche de films, l'utilisation des filtres de recherche avancée, la création de compte, l'interaction avec le profil utilisateur, et la gestion des listes personnelles. L'objectif était de comprendre comment les utilisateurs interagissent avec les fonctionnalités, mais aussi de relever leurs impressions spontanées concernant la clarté des interfaces, l'accessibilité des informations, et la facilité d'utilisation.

Le test a permis de mettre en lumière des comportements inattendus, ainsi que des améliorations possibles pour optimiser l'expérience utilisateur. Ces retours sont précieux pour affiner l'application et s'assurer qu'elle répond parfaitement aux attentes des utilisateurs finaux.

Au bout des 12 mois de développement du site, j'ai réalisé une deuxième série de tests utilisateurs auprès de 10 personnes pour vérifier que les problèmes bloquants avaient été surmontés (tels que bugs, lenteur du site, opérations non abouties) suite à mes corrections et que les nouvelles fonctionnalités étaient opérationnelles. J'ai pu vérifier la performance de l'application. D'autres suggestions d'amélioration m'ont à cette occasion été soumises dont notamment des critères de mots de passe, l'ajout de messages explicites concernant la confirmation d'une action, l'ajout de fonctionnalité de création de liste directement depuis une page film. D'autres légers bugs mineurs ont pu être corrigés à cette occasion.

## VII. Documentation

### I. README

J'ai rédigé un README, essentiel pour apporter toutes les clés pour comprendre le fonctionnement de l'application, faciliter son installation, et permettre à d'autres développeurs de contribuer au projet. J'y décris l'objectif de la plateforme, ses principales fonctionnalités ainsi que les technologies utilisées. J'ai également détaillé les étapes d'installation, que ce soit via Docker pour une mise en place rapide ou en développement local plus avancé. Enfin, j'y ai inclus des instructions pour gérer la base de données avec Prisma, lancer les tests unitaires et end-to-end, et une vue d'ensemble de la structure du projet pour faciliter la prise en main par d'autres développeurs.

#### README.md (extrait)

```
39
40  ### Quick Start with Docker
41
42  This is the easiest way to get the application running locally without complex setup.
43
44  #### Prerequisites
45
46  - [Docker](https://www.docker.com/)
47  - [Docker Compose](https://docs.docker.com/compose/install/)
48
49  #### Steps
50
51  1. Clone the repository:
52
53      ```bash
54      git clone https://github.com/apolline-diaz/queer-cinema-database.git
55      cd queer-cinema-database
56      ```
57
58  2. Create environment file:
59
60      ```bash
61      cp .env.example .env.local
62      ```
63
64  3. Start the application with Docker Compose:
65
66      ```bash
67      docker-compose up -d
68      ```
69
70
71  4. Wait for services to be ready:
72
73      ```bash
74      docker-compose ps
75      ```

76
77
78
79
80
81
```

```
81
82 5. Initialize the database:
83
84  ````bash
85 docker-compose exec app npx prisma migrate dev --name init
86  ```
87
88  ````bash
89 docker-compose exec app npx prisma generate
90  ```
91
92 6. Access the application:
93
94  Main App: http://localhost:3000
95
```

## II. API

Dans l'architecture de mon application, j'utilise exclusivement les Server Actions de Next.js 14 pour interagir avec ma base de données et effectuer les opérations côté serveur. Contrairement à une API REST traditionnelle, les Server Actions ne sont pas exposées via des endpoints HTTP accessibles (comme /api/movies). Elles sont directement appelées depuis les composants côté serveur, ce qui optimise les interactions avec la base de données tout en évitant les aller-retours HTTP. Par conséquent, il n'est pas possible de documenter ou de tester ces actions avec des outils comme Swagger ou Postman, car il n'existe pas d'URL publique pour interroger ces fonctions. Pour les rendre testables, il faudrait créer des routes API dédiées qui encapsulent les appels aux Server Actions.

## VIII. Veille

La veille technologique est une composante essentielle de mon activité de développeuse. Elle me permet de rester à jour sur les évolutions rapides du développement web, de faire des choix techniques éclairés et d'optimiser la qualité, la performance et la maintenabilité de mes projets. Depuis juillet 2024, dans le cadre de la création de mon application, j'ai intégré une démarche de veille régulière et structurée, centrée sur mon stack technologique : Next.js 14, TypeScript, Prisma, Supabase, PostgreSQL, TailwindCSS, Recharts, et React Hook Form.

Je consacre environ une à deux heures par semaine à la veille, réparties entre lecture, tests, sauvegarde de ressources et mise à jour de mon code si nécessaire. Ma veille repose sur plusieurs sources :

**Newsletters spécialisées** : je me suis abonnée à **NEXT.JS Weekly** (<https://nextjsweekly.com/>), une newsletter qui compile les meilleurs articles, outils et projets Next.js, me permettant de découvrir rapidement les nouvelles fonctionnalités et bonnes pratiques. J'utilise également **TLDR Web Dev** (<https://tldr.tech/webdev>), une newsletter quotidienne qui synthétise l'actualité du développement web en quelques minutes de lecture, couvrant les frameworks, outils et tendances émergentes.

**Plateformes communautaires** : discussions sur Reddit (r/reactjs, r/webdev), Dev.to, Stack Overflow, ainsi que GitHub où les discussions dans les issues et pull requests permettent de suivre les problématiques techniques et les solutions apportées par la communauté.

Je suis également des développeurs comme Melvynx qui crée du contenu pour apprendre avec Nextjs et Benjamin Code qui est spécialisé en front-end, qui partagent leurs expériences et bonnes pratiques.

## **IX. R&D / Innovation : Containerisation et environnement de développement local**

### **I. Description du besoin d'information**

Dans le cadre du développement de mon application, j'ai eu besoin de mettre en place un environnement de développement local containerisé pour assurer la cohérence entre les environnements de développement, test et production. L'objectif était de pouvoir lancer facilement mon application Next.js avec une base de données PostgreSQL locale via Supabase, tout en garantissant la reproductibilité de l'environnement.

#### **Problème initial avec Docker Desktop sur macOS :**

Lors de l'installation et de l'utilisation de Docker Desktop sur mon Mac, j'ai été confronté à un problème critique qui compromettait mon workflow de développement. Au démarrage de mon Mac, une popup non-supprimable apparaissait systématiquement, rendant l'utilisation de la machine difficile. Tentant de résoudre le problème en désinstallant puis réinstallant Docker Desktop depuis la dernière version disponible, j'ai été confronté à un message d'erreur inquiétant : **"Docker will damage your computer"**.

Ce problème s'est avéré être un bug connu de compatibilité entre Docker Desktop et certaines versions de macOS, particulièrement problématique sur les puces Apple Silicon. Même après une installation réussie, l'alerte de sécurité persistait à chaque lancement de l'application, identifiant à tort le fichier Docker comme malveillant. Cette situation créait un environnement de développement instable et peu fiable.

## II. Solution trouvée et mise en oeuvre

### 1. Recherche de solutions et adoption d'OrbStack

Face à cette problématique, j'ai fait des recherches pour explorer plusieurs sources. J'ai consulté les issues sur GitHub liées à Docker Desktop, où de nombreux développeurs Mac décrivaient des problèmes similaires aux miens. Parallèlement, j'ai étudié la documentation technique afin de comparer différentes alternatives à Docker Desktop sur macOS. Enfin, j'ai parcouru des forums spécialisés comme Reddit et Stack Overflow pour comprendre quelles solutions la communauté adoptait face à ces difficultés.

Mes recherches m'ont conduite à découvrir OrbStack, une alternative moderne à Docker Desktop spécialement optimisée pour macOS. Cette solution tire parti des API natives de macOS au lieu d'une virtualisation lourde, ce qui améliore les performances. Il consomme moins de mémoire. Le démarrage des conteneurs est quasi instantané, ce qui accélère mon workflow. De plus, OrbStack offre une compatibilité complète avec l'API Docker, sans nécessiter de modifications dans mon code existant. Enfin, son interface est conçue spécialement pour macOS, ce qui facilite la gestion des conteneurs.

### 2. Configuration de l'environnement de développement et intégration avec Supabase Studio

Une fois OrbStack installé, j'ai configuré un environnement de développement intégré qui me permet de lancer mon application Next.js directement dans un conteneur avec hot-reload, ce qui facilite grandement le développement en temps réel. J'ai également déployé PostgreSQL localement en utilisant les images officielles, assurant ainsi une base de données stable et isolée. Pour gérer cette base de données de manière visuelle, j'ai intégré Supabase Studio, ce qui m'offre un contrôle intuitif sur les données locales. Cette intégration me permet de visualiser et modifier facilement les données de test via une interface proche de celle utilisée en production, de tester mes requêtes SQL directement sur les données locales, et de gérer les migrations de manière cohérente entre les environnements local et production, tout en facilitant le débogage des problèmes liés à la base de données. Enfin, j'ai configuré les variables d'environnement afin que mon application pointe vers cette instance locale, garantissant la cohérence et la fluidité de l'ensemble du système.

## Conclusion

Queer Cinema Database est un projet que j'ai développé avec l'ambition de répondre à des besoins réels face à un manque de référencement des œuvres audiovisuelles LGBTQI+. Face aux limites des plateformes généralistes, j'ai voulu proposer une solution spécialisée, permettant une exploration précise des thématiques queer au sein du cinéma.

L'application, développée avec Next.js 14, TypeScript et une base PostgreSQL via Supabase, repose sur une architecture moderne et sécurisée. L'intégration de Prisma pour la gestion des données et l'authentification de Supabase garantissent une expérience utilisateur fluide et conforme aux standards de sécurité actuels.

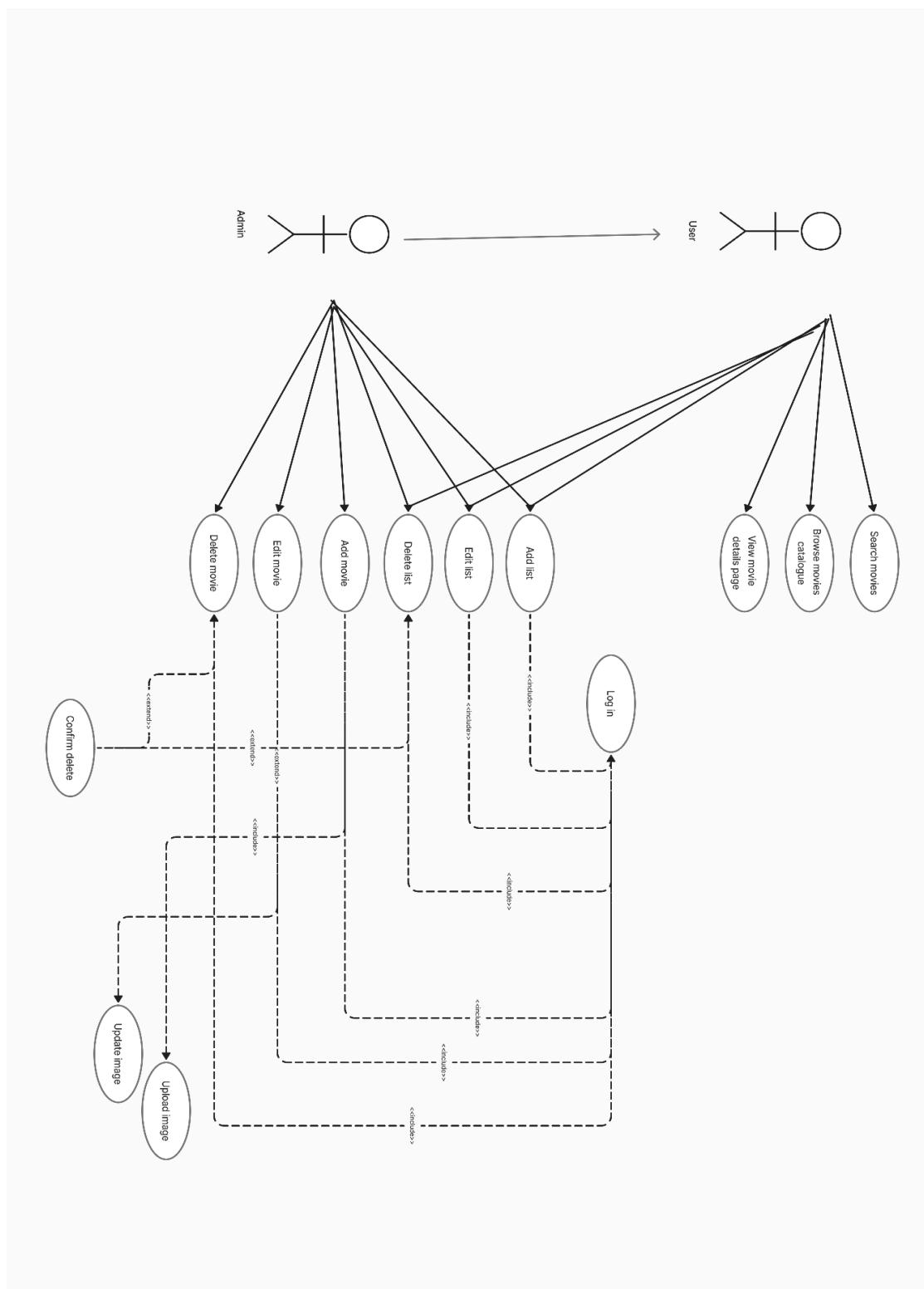
Ce projet m'a permis de mobiliser et d'approfondir les compétences clés du référentiel RNCP : développement sécurisé, conception organisée en couches, et préparation au déploiement sécurisé avec Docker, Jest et Playwright. La phase de R&D autour de la containerisation avec OrbStack a aussi renforcé ma capacité à résoudre des problématiques concrètes liées à l'environnement de développement.

J'ai tenté d'avoir une approche rigoureuse sur les objectifs de qualité, avec des tests automatisés, le respect du RGPD et des normes d'accessibilité, ainsi que des considérations d'éco-conception numérique.

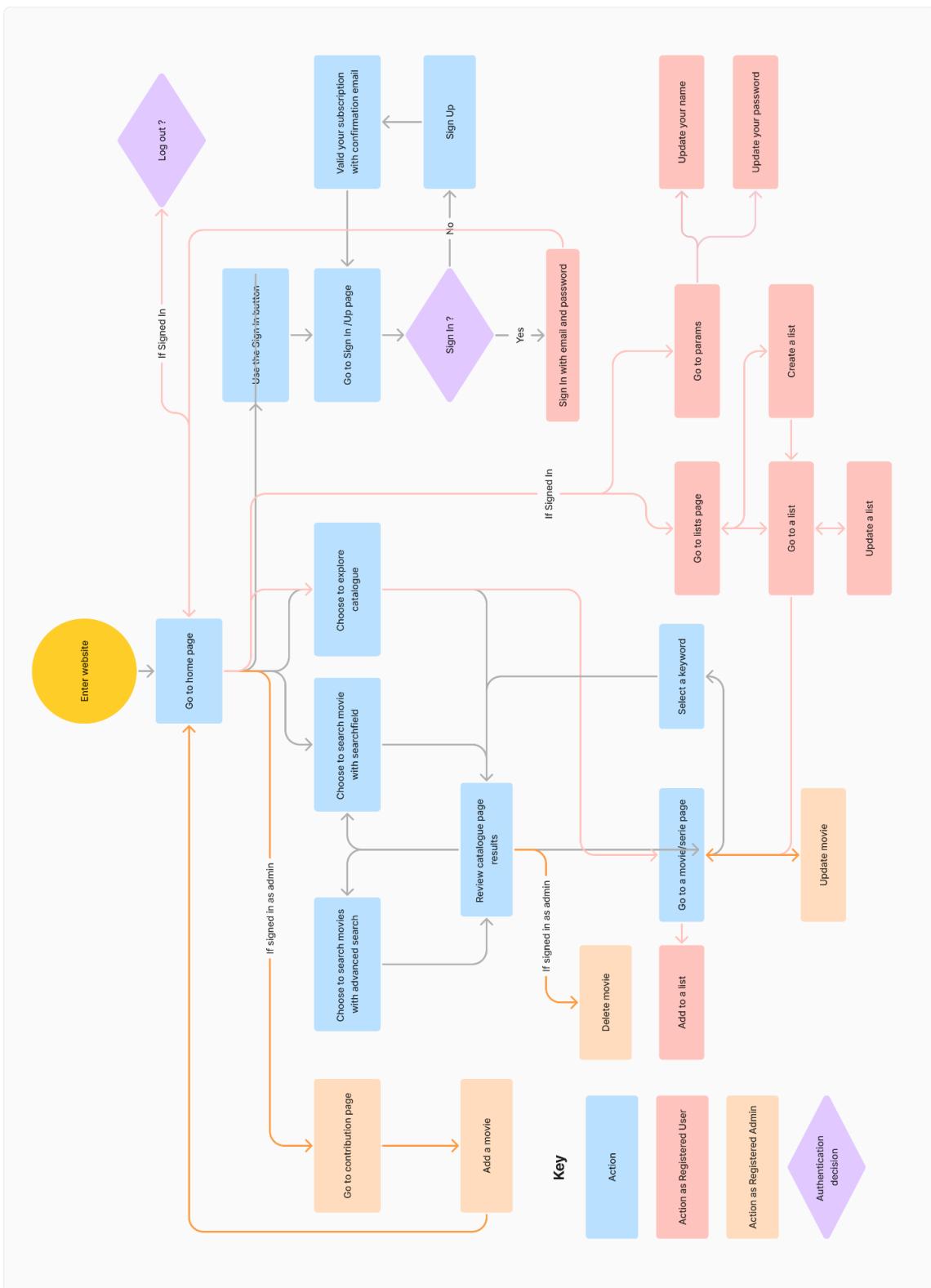
Plusieurs axes d'amélioration restent à explorer, notamment l'optimisation des performances via du cache et du lazy loading, le développement d'une dimension collaborative avec contributions utilisateurs, et la migration vers une infrastructure plus fiable pour les médias (Scaleway). La maintenance et la compatibilité technique, notamment entre Edge Runtime et Prisma, feront aussi l'objet d'un travail approfondi.

# Annexes

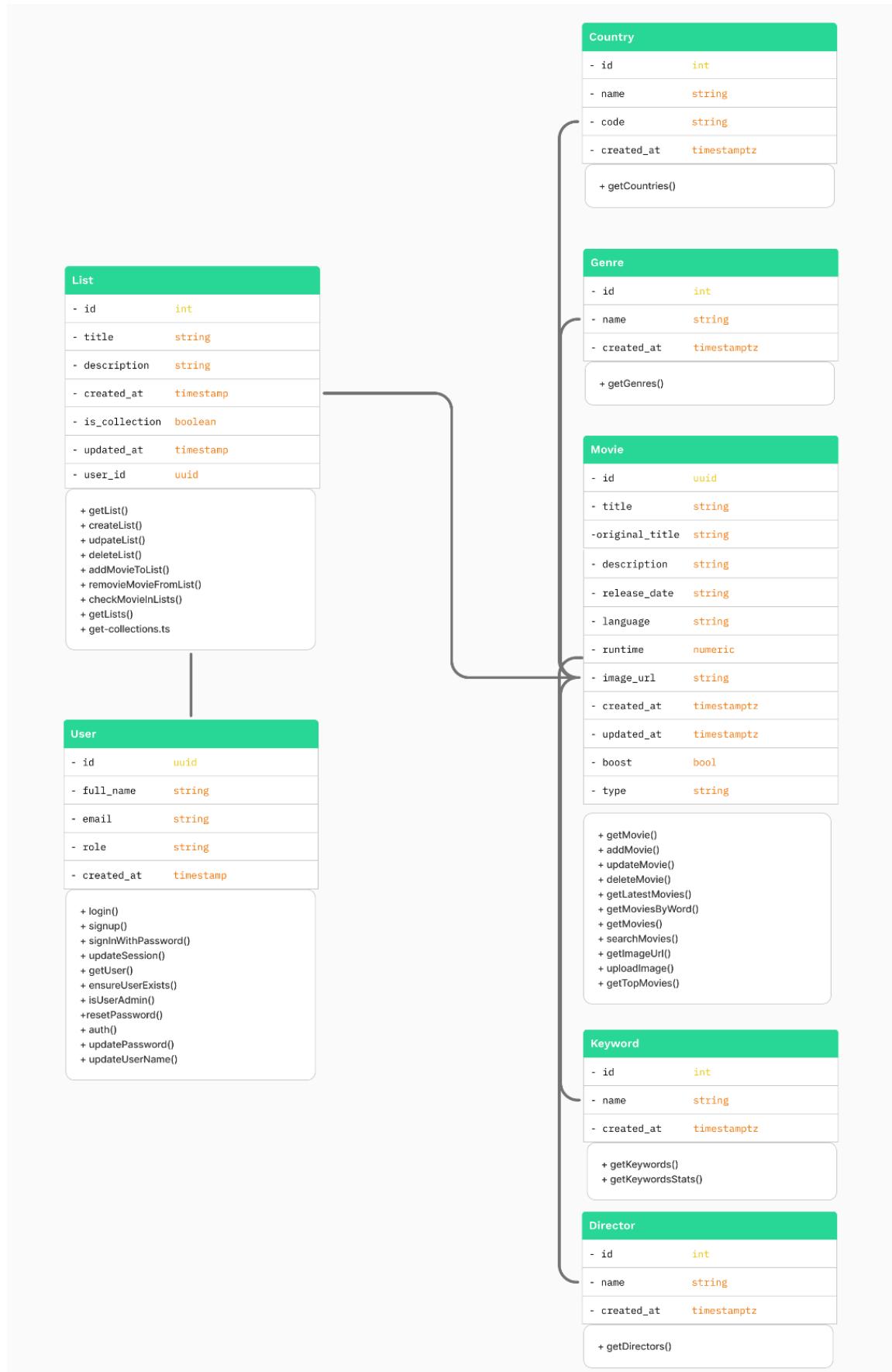
## Annexe 1 : Diagramme des cas d'utilisation



## Annexe 2 : Diagramme d'activité

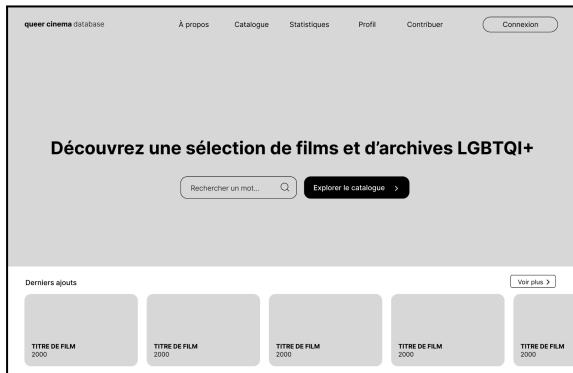


## Annexe 3 : Diagramme de classes.

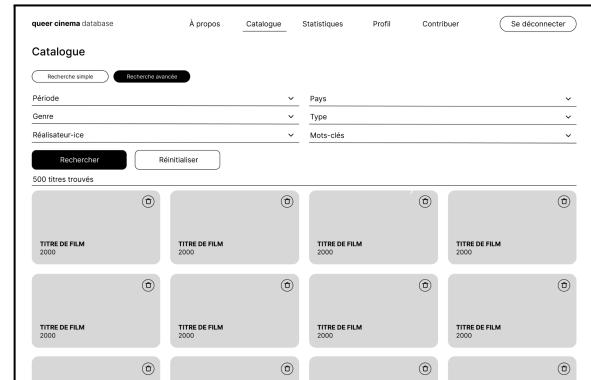


## Annexe 4 : Wireframes

### Page d'accueil



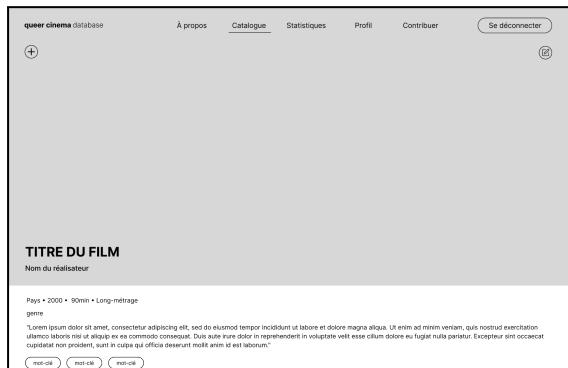
### Page catalogue avec recherche avancée



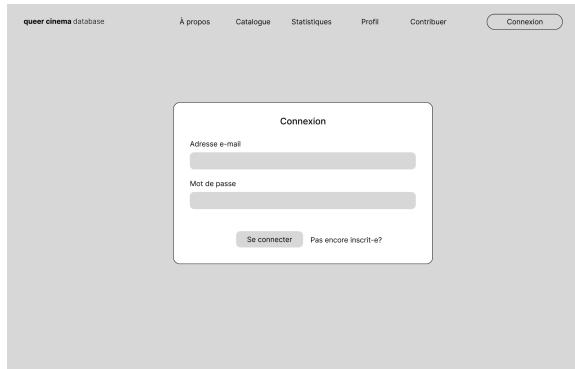
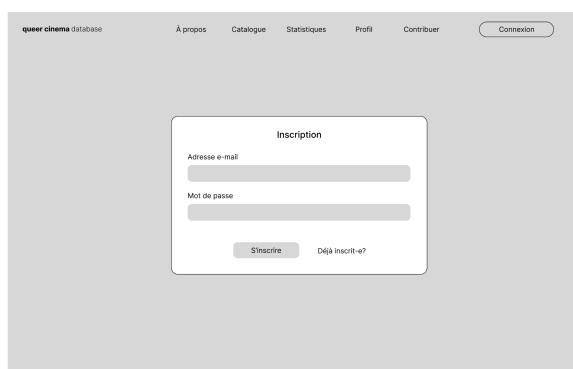
**Si l'utilisateur est admin:** un bouton avec une icône "poubelle" est visible sur chaque card pour supprimer le film.

### Page d'un film

**Si l'utilisateur est admin:** un bouton avec une icône “ modifier” lui permet d'être renvoyé à une page de formulaire pour modifier du film.



### Page d'inscription et Page de connexion



## Page de profil avec la liste des listes de films et Page d'une liste

## Page de création et page de modification d'une liste

## Page de profil avec suppression de liste

## Page d'ajout de film (réservé à l'admin uniquement)

## Page de modification de film (formulaire pré-rempli réservé à l'admin uniquement)

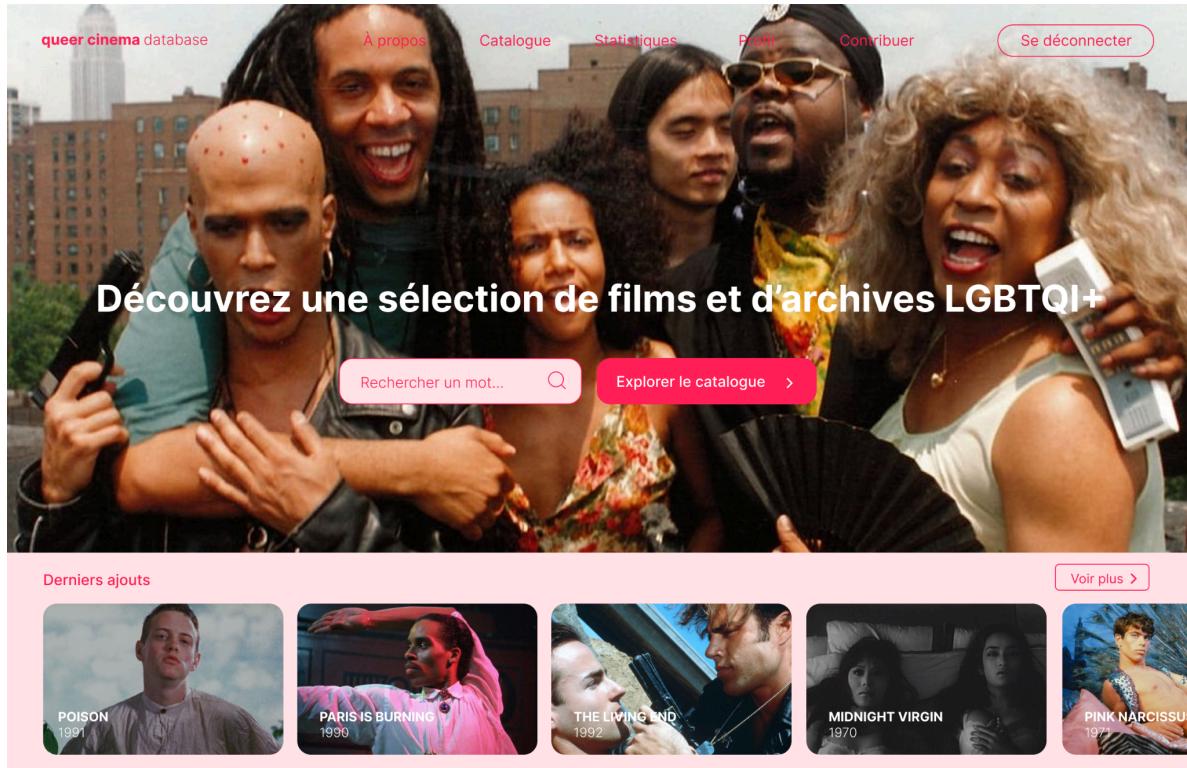
The screenshot shows a form for editing a movie. On the left, there's a sidebar with fields for 'Titre', 'Réalisateur·ice', 'Image actuelle', 'Image URL', 'Télécharger une nouvelle image', and 'Synopsis'. The main area contains fields for 'Année de sortie', 'Pays', 'Durée (en minutes)', 'Langue', 'Type', 'Genre', 'Mots-clés', and two dropdown menus for 'Genre' and 'Mots-clés'. At the bottom right are 'Annuler' and 'Enregistrer les modifications' buttons.

## Page du catalogue avec suppression de film

The screenshot shows a catalog page with a search bar and filters for 'Pays', 'Type', and 'Mots-clés'. Below the filters, there's a message: 'Voulez-vous vraiment supprimer cet élément ? Cette action est irréversible.' with 'Annuler' and 'Confirmer la suppression' buttons. A grid of movie cards is visible, each labeled 'TITRE DE FILM' and '2000'. A modal dialog box is overlaid on the page.

## Annexe 5 : Maquettes

### Page d'accueil



### Page d'inscription et page de connexion

La maquette montre deux pages adjacentes. La page de gauche, intitulée "Inscription", contient des champs pour "Adresse e-mail" et "Mot de passe", ainsi qu'un bouton "S'inscrire" et un lien "Déjà inscrit-e?". La page de droite, intitulée "Connexion", contient des champs pour "Adresse e-mail" et "Mot de passe", ainsi qu'un bouton "Se connecter" et un lien "Pas encore inscrit-e?". Les deux pages partagent un menu en haut avec les options "queer cinema database", "À propos", "Catalogue", "Statistiques", "Profil", "Contribuer" et "Connexion".

Page de catalogue

Page d'un film

queer cinema database

À propos Catalogue Statistiques Profil Contribuer Se déconnecter

### Catalogue

Recherche simple Rechercher avancée

Période Pays

Genre Type

Réalisateur·ice Mots-clés

Rechercher Réinitialiser

500 titres trouvés

queer cinema database

À propos Catalogue Statistiques Profil Contribuer Se déconnecter

**GO FISH**  
Rose Troche

Etats-Unis • 1994 • 83min • Long-métrage

comédie

Max est une jeune lesbienne branchée qui a du mal à trouver l'amour. Une amie lui fait rencontrer Ely, qui plait à Max, mais Ely est mal fagotée, cassinerre et plus âgée. Elles n'ont pas grand-chose en commun. Mais peut-être apprendre à regarder au-delà des apparences ?

mon œuvre mot œuvre mot œuvre

## Page de profil

queer cinema database

À propos Catalogue Statistiques Profil Contribuer Se déconnecter

### Mes listes

Créer une nouvelle liste +

## Page de création de liste

queer cinema database

À propos Catalogue Statistiques Profil Contribuer Se déconnecter

### Créer une liste de films

Titre

Description

Sélection des films

Créer la liste

## Page d'ajout de film

queer cinema database

À propos Catalogue Statistiques Profil Contribuer Se déconnecter

### Ajouter un film au catalogue

Titre **DEALS**

Réalisateur·ice Argie Robinson

Image actuelle

Image URL <data:01204802238840.wsdp>

Télécharger une nouvelle image Choisir un fichier Aucun fichier choisi

Synopsis

Recrutés par le gouvernement américain pour leur capacité unique à mentir, tricher et se battre, Any, Max, Jovet et Dominique rejoignent une académie clandestine d'agents secrets connue sous le nom de D.E.B.S.

Année de sortie 2004 Pays Etats-Unis

Durée (en minutes) 91 Langue Anglais

Type Long-métrage

Genres

comédie action

Mots-clés

Non romane romance

Annuler Enregistrer les modifications

## Page de modification de film

queer cinema database

À propos Catalogue Statistiques Profil Contribuer Se déconnecter

### Ajouter un film au catalogue

Titre

Réalisateur·ice

Synopsis

Année de sortie

Pays

Durée (en minutes)

Type

Genre

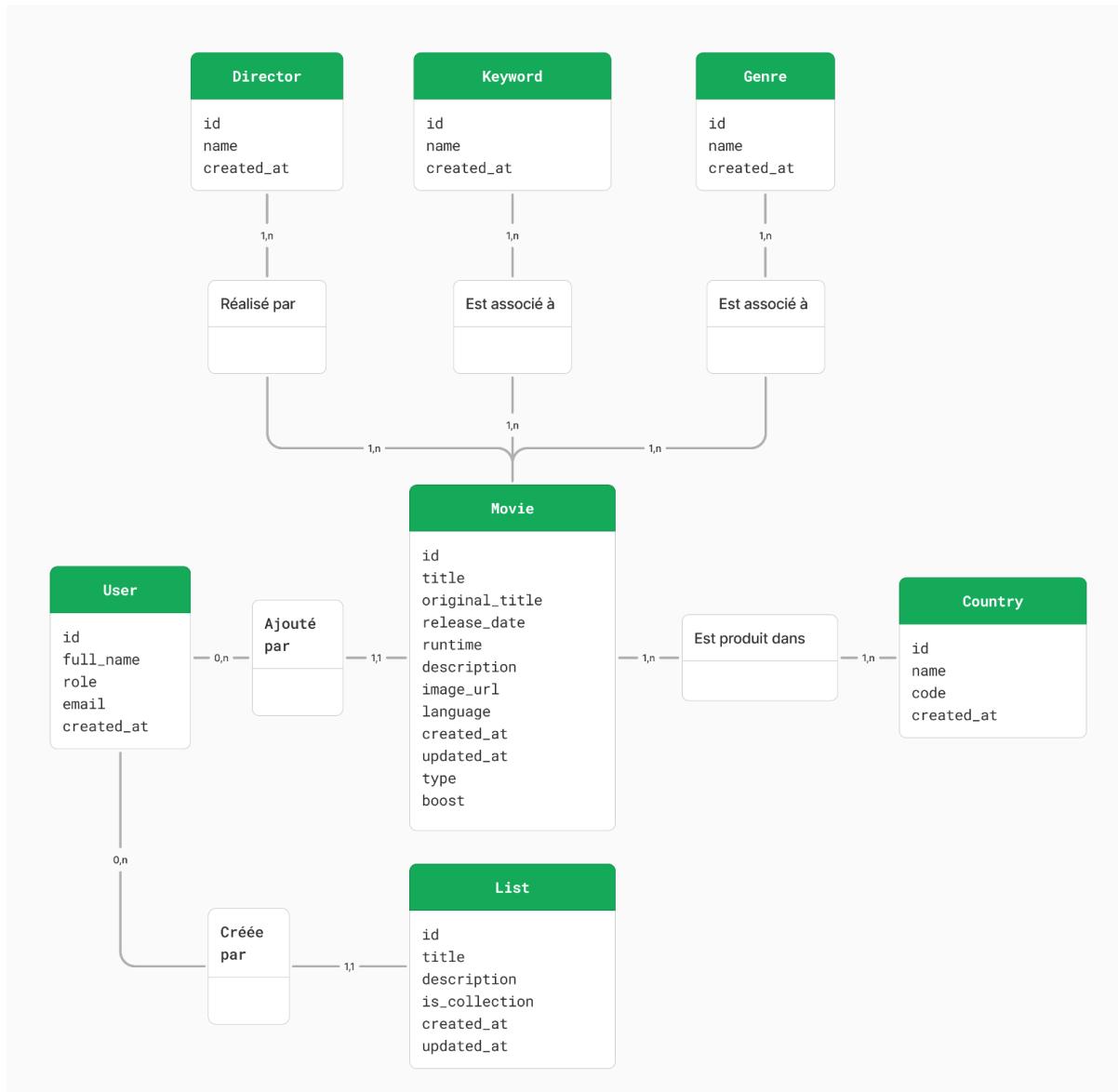
Mots-clés

Image

Choisir un fichier Aucun fichier choisi

Annuler Ajouter le film

## Annexe 6 : Modèle conceptuel de données



## Annexe 7 : Modélisation logique de données



## Annexe 8 : Registre des traitements de données

### Liste des traitements

<b>Coordonnées du responsable de l'organisme (responsable de traitement ou son représentant si le responsable est situé en dehors de l'UE)</b>	Nom : Diaz Code postal : 93310	Prénom : Apolline Ville : Le Pré-Saint-Gervais	Adresse : 11 rue Béranger Téléphone : 621799104	Adresse mél : apolline.diaz@gmail.com
<b>Coordonnées du représentant (responsable de traitement ou son représentant si le responsable est situé en dehors de l'UE)</b>	Nom : Diaz Code postal : 93310	Prénom : Apolline Ville : Le Pré-Saint-Gervais	Adresse : 11 rue Béranger Téléphone : 621799104	Adresse mél : apolline.diaz@gmail.com
<b>Coordonnées du délégué à la protection des données (DPO)</b>	Nom : Code postal :	Prénom : Ville :	Société (si DPO externe) : Téléphone :	Adresse : Adresse mél :
Identification du traitement		Finalité du traitement		Données sensibles ?
Nom du traitement	N° / RÉF	Date de création de la fiche	Dernière mise à jour de la fiche	Oui/non
Gestion des utilisateurs et authentification	1	03/07/2025	03/07/2025	Stockage des adresses e-mail des utilisateurs pour l'authentification, identification et support en cas de bugs Non

## Registre des traitements

Fiche de registre								1
Description du traitement								
Nom du traitement	Gestion des comptes utilisateurs et authentification							
N° / RÉF	1							
Date de création du traitement	03/07/2025							
Mise à jour du traitement	03/07/2025							
Acteurs								
Responsable du traitement	Apolline DIAZ	11 rue Béranger	93310	Le Pré-Saint-Gervais	France	0621799104	apolline.diaz@gmail.com	
Délégué à la protection des données	Non désigné							
Société du DPO (si celui-ci est externe)	N/A							
Finalité(s) du traitement effectué								
Finalité principale	Gestion des comptes utilisateurs pour authentification, identification, support et contact							
Sous-finalité 1	Support technique (ex : identifier un utilisateur en cas de bug)							
Sous-finalité 2	Envoi d'emails liés au compte (ex : notifications, reset mot de passe)							
Sous-finalité 3	Gestion des sessions et sécurisation de l'accès							
Catégories de données personnelles concernées			Description			Durée de conservation		
État civil, identité, données d'identification, images...	Adresse mail, identifiant unique			Jusqu'à suppression du compte utilisateur				
Informations d'ordre économique et financier (revenus, situation financière, situation fiscale)	Non collecté			N/A				
Numéro de Sécurité Sociale (ou NIR)	Non collecté			N/A				
Données de connexion (IP, logs, etc.)	Collectées via Supabase (logs d'accès, IP)			Selon politique Supabase / logs temporaires				
Catégories de personnes concernées			Description			Précisions		
Utilisateurs	Personnes inscrites sur l'application			Utilisateurs authentifiés				
Destinataires			Type de destinataire			Précisions		
Administrateur	Responsable du traitement			Gestion et administration				
Supabase	Sous-traitant			Base de données et authentification				
Vercel	Hébergeur			Hébergement et déploiement				
Mesures de sécurité			Type de mesure de sécurité			Précisions		
Authentification sécurisée	Contrôle d'accès			Supabase Auth utilise des tokens JWT sécurisés avec expiration et rafraîchissement automatique				
Chiffrement des données	Sécurité des données			Données stockées chiffrées au repos et en transit via TLS, grâce aux infrastructures sécurisées de Supabase				
Hébergement sécurisé	Hébergement sécurisé			Hébergement sur Vercel avec certificats SSL/TLS, protection contre les attaques DDoS et mise à jour continue				
Accès restreint	Gestion des droits d'accès			Seuls l'administrateur a accès aux données sensibles, avec authentication				
Transferts hors UE		Destinataire	Pays	Type de Garanties		Liens vers la documentation		
	Supabase	Etats-Unis	Clauses Contractuelles Types (CCT)	<a href="https://supabase.com/privacy">https://supabase.com/privacy</a>				
	Vercel	Etats-Unis	Clauses Contractuelles Types (CCT)	<a href="https://vercel.com/legal/privacy-policy">https://vercel.com/legal/privacy-policy</a>				

## Annexe 9 : Dockerfile

```
1 # Step 1: Base image for the build
2 FROM node:18-alpine AS base
3 WORKDIR /app
4
5 RUN npm install -g pnpm
6
7 # Copy the necessary files for dependency installation
8 COPY package.json package-lock.json ./prisma/
9 COPY prisma/ prisma/
10
11 # Install dependencies
12 RUN pnpm install
13
14 # Copy the rest of the code
15 COPY . .
16
17 # Expose the application port
18 EXPOSE 3000
19
20 # Command to start the application
21 CMD ["pnpm", "dev"]
```

## Annexe 10 : Fichier compose.yml

```
2 services:
3   app:
4     build:
5       context: .
6       container_name: next-app
7
8     ports:
9       - "3000:3000"
10    depends_on:
11      - db
12    env_file:
13      - .env
14    environment:
15      - DATABASE_URL=${DATABASE_URL}
16      - NEXT_PUBLIC_SUPABASE_ANON_KEY=${NEXT_PUBLIC_SUPABASE_ANON_KEY}
17      - NEXT_PUBLIC_SUPABASE_SERVICE_ROLE_KEY=${NEXT_PUBLIC_SUPABASE_SERVICE_ROLE_KEY}
18    volumes:
19      - ./app
20    command: ["pnpm", "dev"]
21
22 db:
23   image: postgres:15-alpine
24   container_name: postgres-db
25   restart: always
26   ports:
27     - "5432:5432"
28   volumes:
29     - db-data:/var/lib/postgresql/data
30   environment:
31     POSTGRES_USER: postgres
32     POSTGRES_PASSWORD: Virg024&
33     POSTGRES_DB: postgres
34   volumes:
35     db-data:
```

## Annexe 11 : Résultats du questionnaire pour le test utilisateurs (extraits)

### 6. Comment trouvez-vous la page d'un film ?

Rendez-vous sur une page de film et évaluez la clarté et la précision.

tip top, c'est 100% clair et précis!! j'imagine que c'est

### 7. Comment se passe la création de votre compte

Créez un compte et décrivez si l'expérience est simple.

NB: après la création vous recevrez un mail de confirmation.

### 8. Comment trouvez-vous votre espace profil ?

Connectez-vous et visitez votre profil. Notez vos impressions.

j'ai dû m'y reprendre à deux fois pour créer un compte quand j'ai essayé de me connecter après ma première connexion.

je m'attendais à un profil plus personnalisable : chaque utilisateur pourra appeler ou appeler cette page mes listes plutôt que l'onglet connexion ne se rabat pas alors que la page

C'est là où j'ai le plus de retours à faire ^^

Au début, je me suis inscrite et je ne savais pas si r Idéalement, il faudrait informer l'utilisateur lors de la connexion.

Ce n'est pas vraiment une page de profil, mais une liste de films.

Sur mobile, il n'y a pas le bouton connexion par contre il y a un bouton déconnexion.

Mais cette page de liste est très bien pour pouvoir consulter les films.

Lorsqu'on clique sur le bouton déconnection, il ne suffit pas de cliquer sur le bouton de déconnection.

Prévoir une partie suppression de compte également.

Les pages sont très claires avec les hashtags qui sont placés dans les titres.

Dans la création de compte, peut-être rajouter un champ pour la confirmation de l'email.

Tant qu'on a pas confirmé notre adresse email, le nom et la date de naissance doivent être renseignés.

tout est ok, une frustration sur un extrait vidéo.

la création du compte est claire. En revanche, il n'est pas très intuitif.

Le profil est pratique avec la liste.

Super

Si jamais il y a des infos à rajouter y'a ces autres idées :

- langue(s) parlée(s) du film

- autre titre (pour que soient référencés les films qu'on a vus)

Expérience claire, le mail de confirmation était dans la boîte de réception.

Pour l'instant c'est tout vide mais c'est très clair et ça fonctionne.

Peut-être qu'on aurait envie d'avoir la micro possibilité de modifier les informations.

Je trouve que les mots-clés restent un peu trop générés.

Le mail de confirmation est allé dans les spams. Et

Il n'y a que <creer une nouvelle liste>. C'est un peu dommage.

### 9. Comment s'est passée la création et la modification d'une liste de films ?

Créez une liste de films, modifiez-la, puis indiquer les difficultés rencontrées.

### 10. Sauriez-vous ajouter un film à votre liste depuis une recherche ?

Ajoutez directement un film à une liste depuis une recherche.

Si vous n'avez pas trouvé comment faire, indiquez.

### 11. Quel est votre ressenti général sur le site ?

Donnez votre avis global : points forts, points à améliorer.

hyper simple!! ce serait peut-être chouette de pouvoir choisir l'ordre

j'ai cherché un film depuis la page d'accueil et j'ai eu du mal à trouver le bouton de recherche.

très chouette, j'ai déjà repéré des films que j'ai envie de voir. Les points forts : le projet en lui-même, la base de données, la facilité d'utilisation.

points à améliorer : c'est encore une fois hyper perso, mais je n'ai pas encore testé toutes les fonctionnalités.

bon courage bb <3

La création, la modification sont simples et plutôt fluides.

J'adore beaucoup beaucoup beaucoup le design, b

Je ne peux pas créer de liste vide, il faut forcément qu'il y ait au moins un film.

J'ai trouvé le bouton, mais effectivement je ne sais pas où il est.

Les améliorations selon moi seraient sur la partie recherche.

Il me manque une info comme quoi l'enregistrement est terminé.

Par contre une fois trouvé, c'est tout bon et c'est pratique.

Et quelques petits trucs comme les messages d'info.

Ce serait cool si j'avais un bouton supprimer la liste

A dispo si tu as des questions sur les retours que je te ferai.

C'est facile à faire et intuitif.

Je clique sur les trois lignes en haut à gauche de la page.

C'est une bonne idée ce projet car au travers de la recherche, on peut trouver des films intéressants.

J'aime le fait de proposer du contenu LGBTQIA+, de

La création d'une liste fonctionne. L'onglet recherche je me suis déconnecté et n'ai pas réussi à me reconnecter.

je me suis déconnecté et n'ai pas réussi à me reconnecter.

J'aimerais pouvoir proposer un média en tant qu'utilisateur.

Le moteur de recherche de l'accueil ne fonctionne pas.

L'input recherchez un film dans le profil/création de liste.

Le manque de contenu vidéos manque pour ma part.

Sans cette explication je n'aurais pas trouvé ! Donc

Très facile et intuitif

Idéalement ça donnerait envie soit d'avoir accès au

Grâce à l'outil de recherche, je vais pouvoir trouver rapidement ce que je recherche.

ET / OU

ça donnerait envie d'avoir un bouton sur chaque vignette.

Création de liste, facile. Après, je me demande si je suis

J'ai trouvé le bouton assez rapidement mais j'ai du mal à le trouver.

Je reviendrai, c'est sûr. C'est un travail colossal. J'aime beaucoup le design.