

Cinema

Name: Muntean Diana-Gratiana

Group: 30236

Table of Contents

[Deliverable 1 3](#_Toc64843130)

[Project Specification 3](#_Toc64843131)

[Functional Requirements 3](#_Toc64843132)

[Use Case Model 3](#_Toc64843133)

[Use Cases Identification 3](#_Toc64843134)

[UML Use Case Diagrams 3](#_Toc64843135)

[Supplementary Specification 3](#_Toc64843136)

[Non-functional Requirements 3](#_Toc64843137)

[Design Constraints 3](#_Toc64843138)

[Glossary 3](#_Toc64843139)

[Deliverable 2 3](#_Toc64843140)

[Domain Model 3](#_Toc64843141)

[Architectural Design 4](#_Toc64843142)

[Conceptual Architecture 4](#_Toc64843143)

[Package Design 4](#_Toc64843144)

[Component and Deployment Diagram 4](#_Toc64843145)

[Deliverable 3 4](#_Toc64843146)

[Design Model 4](#_Toc64843147)

[Dynamic Behavior 4](#_Toc64843148)

[Class Diagram 4](#_Toc64843149)

[Data Model 4](#_Toc64843150)

[System Testing 4](#_Toc64843151)

[Future Improvements 4](#_Toc64843152)

[Conclusion 4](#_Toc64843153)

[Bibliography 4](#_Toc64843154)

# Deliverable 1

## Project Specification

Acest proiect își propune să simuleze o aplicație pentru un cinematograf care poate avea 3 tipuri de utilizatori: client, angajat, administrator, și prezintă, de asemenea, o interfață de conectare în care utilizatorii pot accesa și, în funcție de tipul de cont, se deschid interfețe specifice. Clienții pot vizualiza filmele disponibile și pot cumpăra bilete, angajații pot adăuga/actualiza/șterge/căuta filmele, iar administratorul poate manipula angajații.

## Functional Requirements

Când rulezi aplicația, se deschide interfața de conectare/înregistrare, iar dacă nu ai un cont, poți crea unul și te poți conecta. Dacă te autentifici cu un cont de tip client, se deschide interfața sa specifică, unde filmele disponibile sunt preluate din baza de date și afișate într-o tabelă cu coloane pentru titlu, categorie, dată, preț și oră. Cu contul de angajat, poți vedea inițial toate filmele disponibile și apoi poți căuta unul după ID, insera unul nou, modifica detaliile pentru unul existent și șterge pe cel pe care dorești, iar tabela este actualizată după fiecare operație.

## Use Case Model 1

### Use Cases Identification

Use-Case: Vizualizare filme disponibile

Level: Obiectiv al utilizatorului

Primary Actor: Customer

Main success scenario: Customer vede toate filmele disponibile

Extensions: -

Use-Case: Creare filme

Level: Obiectiv al utilizatorului

Primary Actor: Employee

Main success scenario: Employee va putea adauga filme

Extensions: -

Use-Case: Cautare filme

Level: : Obiectiv al utilizatorului

Primary Actor: Employee

Main success scenario: Employee va putea cauta filme

Extensions: Daca nu exista filme care sa corespunda filtrelor, va fi afissata o tabela goala

Use-Case: Actuzalizare filme

Level: : Obiectiv al utilizatorului

Primary Actor: Employee

Main success scenario: Employee va putea actualize filme

Extensions: -

Use-Case: Stergere filme

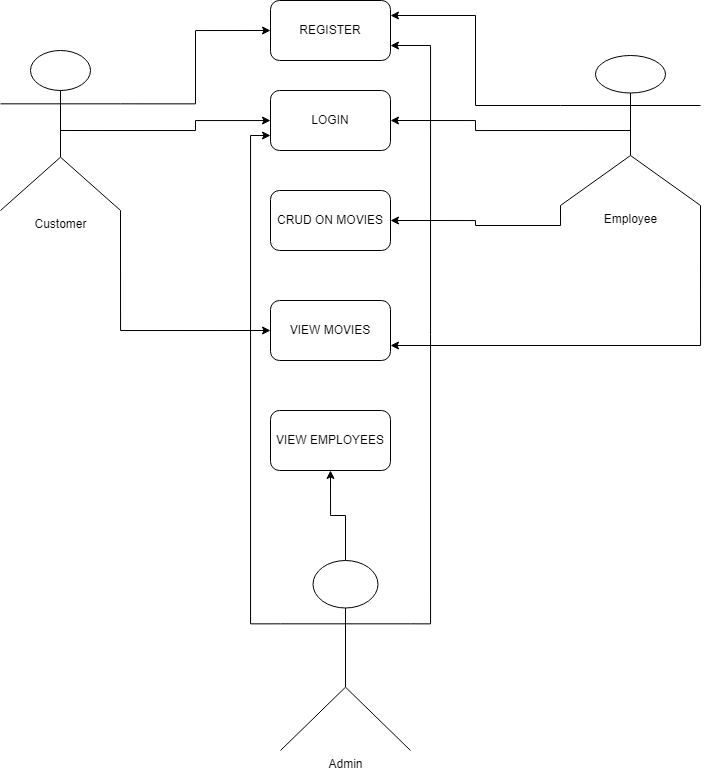
Level: : Obiectiv al utilizatorului

Primary Actor: Employee

Main success scenario: Employee va putea sterge filme

Extensions: -

### UML Use Case Diagrams



## Supplementary Specification

### Non-functional Requirements

1. **Performanță și Timp de Răspuns**: Sistemul ar trebui să ofere timpi de răspuns rapizi pentru interogările utilizatorilor, asigurându-se că afișarea filmelor disponibile și procesarea tranzacțiilor (cum ar fi cumpărarea de bilete) sunt finalizate într-un interval de timp acceptabil. De exemplu, interogările privind disponibilitatea filmelor ar trebui să aibă un timp de răspuns de mai puțin de 2 secunde pentru a asigura o experiență de utilizare lină.

2. **Securitate**: Sistemul trebuie să protejeze datele personale ale utilizatorilor și informațiile de plată în conformitate cu standardele de securitate actuale, inclusiv criptarea datelor sensibile și utilizarea de protocoale sigure pentru transmiterea datelor. Măsuri de autentificare și autorizare ar trebui, de asemenea, implementate pentru a preveni accesul neautorizat.

3. **Scalabilitate**: Sistemul ar trebui să fie proiectat pentru a susține un număr mare de utilizatori simultani și să poată scala resursele în funcție de cerere. Acest lucru este crucial în orele de vârf când mulți utilizatori încearcă să cumpere bilete pentru premiere sau evenimente speciale.

4. **Ușurință de Utilizare și Accesibilitate**: Interfața utilizatorului ar trebui să fie intuitivă și ușor de navigat pentru toate categoriile de utilizatori, inclusiv persoanele cu dizabilități. Ar trebui să fie compatibilă cu cititoarele de ecran și să respecte standardele de accesibilitate web (WCAG).

### Design Constraints

1. **Limbajul de Programare**: Sistemul va fi dezvoltat folosind Java, profitând de Spring Framework pentru crearea unei aplicații web responsive și scalabile. Această restricție se datorează utilizării răspândite a Java în medii enterprise și bibliotecilor sale robuste pentru gestionarea securității, operațiilor de bază de date și serviciilor web.

2. **Managementul Bazelor de Date**: MySQL va fi utilizat ca sistem de gestionare a bazelor de date datorită fiabilității sale, scalabilității și suportului pentru interogări complexe necesare pentru gestionarea listelor de filme, conturilor de utilizatori și tranzacțiilor cu bilete.

3. **Dezvoltarea Interfeței Utilizator**: Interfața utilizatorului va fi dezvoltată cu React.js pentru a asigura o experiență dinamică și receptivă pe partea clientului. Această alegere se bazează pe arhitectura sa bazată pe componente, care facilitează dezvoltarea componentelor de interfață utilizator reutilizabile.

4. **Protocoale de Securitate**: Toate transmisiile de date între client și server trebuie să fie criptate folosind protocoalele TLS/SSL. În plus, datele utilizatorilor și parolele stocate în baza de date trebuie să fie hash-uite pentru a îmbunătăți securitatea.

5. **Mediu de Implementare**: Sistemul va fi containerizat folosind Docker, facilitând mediile consistente de dezvoltare, testare și producție. Acest lucru asigură că aplicația rulează fără probleme în diferite medii de calcul.

## Glossary

• **Tipuri de Utilizatori**: Se referă la rolurile distincte din cadrul sistemului: client, angajat și administrator, fiecare cu diferite drepturi de acces și funcționalități.

• **Listare Filme**: O înregistrare în baza de date care conține informații despre un film, cum ar fi titlul, categoria, datele de proiecție, prețurile și orele.

• **Tranzacție Bilet**: Procesul prin care un client achiziționează un bilet pentru o proiecție de film. Include selectarea filmului, orei, procesarea plății și emiterea unui bilet digital.

• **Protocoale de Securitate**: Standarde și algoritmi utilizați pentru a cripta datele și comunicațiile, inclusiv TLS/SSL pentru securitatea rețelei și algoritmii de hash pentru stocarea informațiilor sensibile.

• **Containerizare**: Utilizarea containerelor Docker pentru a împacheta aplicația cu toate dependențele sale, asigurând consistența în diferitele etape de dezvoltare și implementare.

# Deliverable 2

## Domain Model

Entități Principale:

1. Utilizator

* Atribute: ID, Nume, Email, Parolă, Rol (Client, Angajat, Administrator)
* Relații:
* Clientul poate face rezervări.
* Angajatul poate gestiona filmele.
* Administratorul poate gestiona utilizatorii și filmele.

1. Film

* Atribute: ID Film, Titlu, Categorie, Dată Lansare, Durată, Preț Bilet
* Relații:
* Filmele pot fi vizualizate de către clienți.
* Filmele sunt gestionate de angajați.

1. Proiecție

* Atribute: ID Proiecție, ID Film, Dată, Ora, Sala
* Relații:
* Mai multe proiecții pot fi asociate unui film.
* Proiecțiile au loc în săli diferite.

1. Rezervare

* Atribute: ID Rezervare, ID Utilizator, ID Proiecție, Număr Bilete
* Relații:
* O rezervare este făcută de un utilizator.
* O rezervare este asociată unei proiecții.

1. Sală

* Atribute: ID Sală, Număr Locuri, Facilități
* Relații:
* O sală găzduiește multiple proiecții.

Relații:

* Utilizator - Rezervare: One-to-Many (un utilizator poate avea mai multe rezervări, o rezervare aparține unui singur utilizator)
* Film - Proiecție: One-to-Many (un film poate avea mai multe proiecții, o proiecție aparține unui singur film)
* Proiecție - Rezervare: One-to-Many (o proiecție poate avea mai multe rezervări, o rezervare aparține unei singure proiecții)
* Sală - Proiecție: One-to-Many (o sală poate găzdui mai multe proiecții)



## 

## Architectural Design

### Conceptual Architecture

Arhitectura Layer oferă un echilibru între complexitate și beneficiile de performanță, scalabilitate și flexibilitate, fiind potrivită pentru o aplicație dinamică și cu multiple puncte de interacțiune cu utilizatorul, cum ar fi un sistem de rezervări pentru un cinema.

Pentru aplicația de cinema, recomand utilizarea arhitecturii bazate pe microservicii datorită următoarelor avantaje:

1. Presentation Layer (Stratul de Prezentare)

Rol: Acest strat gestionează interacțiunea cu utilizatorii.

Tehnologii: REACT

1. Business Logic Layer

Rol: Conține regulile și procesele de afaceri ale aplicației. Se iau decizii: ce se întâmplă cu datele și cum sunt ele procesate.

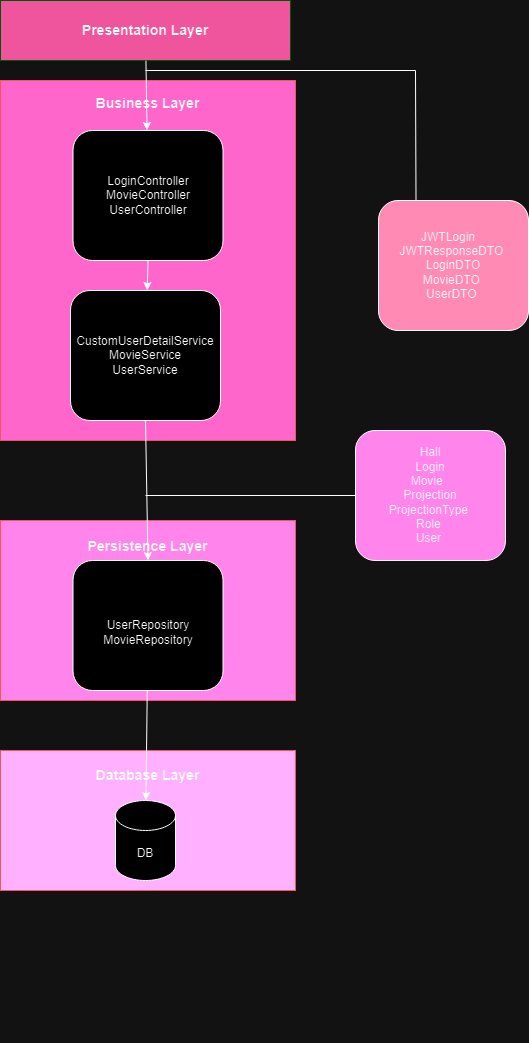
În MVC: Reprezintă „Controller”-ul, care interpretează datele trimise de utilizator și decide acțiunile ce trebuie întreprinse.

1. Data Access Layer

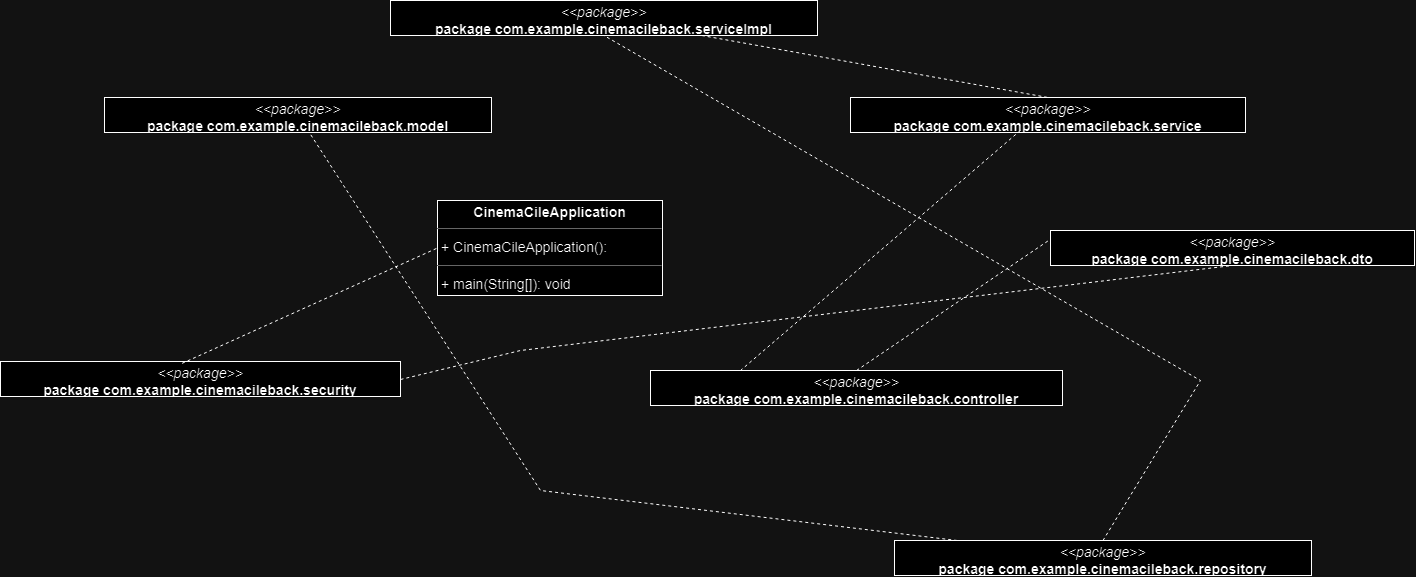
Rol: Stratul de acces la date facilitează comunicarea între business logic și sursele de date, cum ar fi bazele de date sau serviciile web.

1. Model Layer

Rol: Definește structura datelor utilizate în aplicație, inclusiv schema bazei de date și modelele de obiecte utilizate în logica de afaceri.

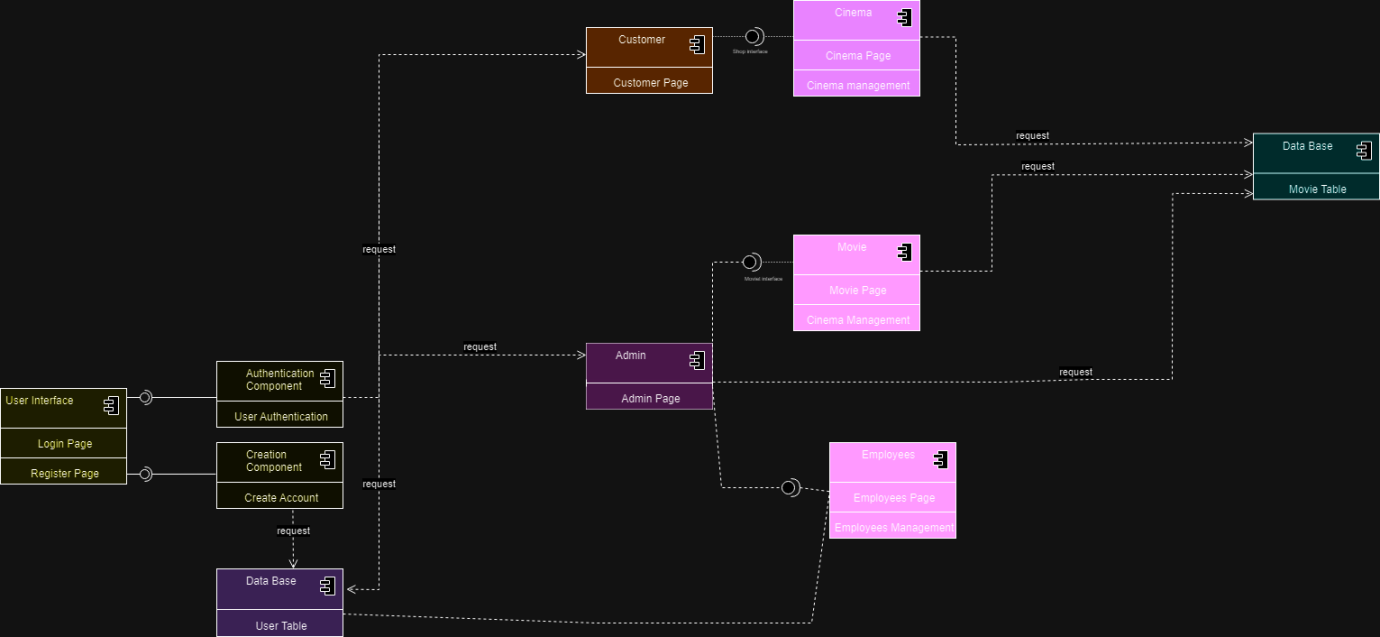


### Package Design

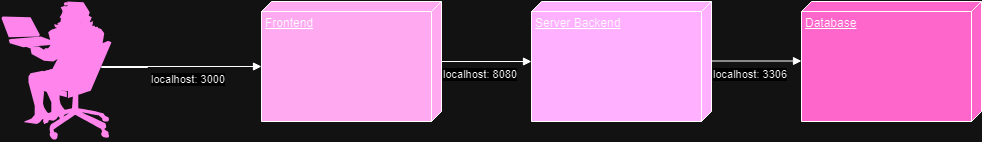


### Component and Deployment Diagram

Component Diagram:



Deployment Diagram:



# 

# Deliverable 3

## Design Model

### Dynamic Behavior

Diagrama de secventa LOGIN

Diagrama de secvență ilustrează pașii succesivi ai interacțiunilor dintre actorii implicați în procesul de autentificare

Participanți: User și Sistemul

Mesaje:

* User trimite o cerere HTTP GET către sistem pentru autentificare, oferind adresa de e-mail și parola.
* Sistemul primește cererea și extrage parametrii (adresa de e-mail și parola).
* Sistemul caută în baza de date persoana asociată cu adresa de e-mail furnizată.
* Sistemul trimite un răspuns HTTP adecvat în funcție de rezultatul verificării.

Secvențe de viață: Verificarea existenței persoanei în baza de date

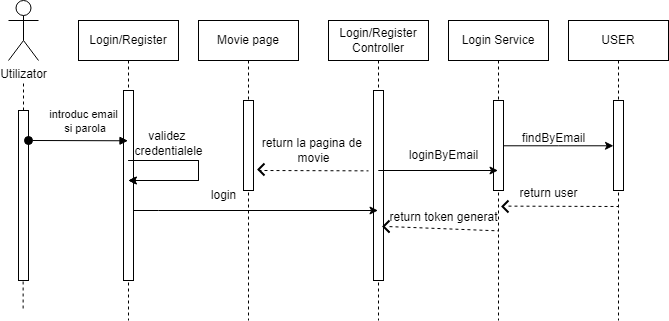


Diagrama de Comunicare LOGIN

Diagrama de comunicare prezintă interacțiunile dintre obiectele implicate în procesul de autentificare.

Obiecte:

* Controller-ul pentru autentificare
* Repozitoriu pentru persoane (PersonRepository)

Mesaje:

* Cererea HTTP GET trimisă către controller-ul pentru autentificare.
* Interogarea bazei de date pentru a identifica persoana asociată cu adresa de e-mail.
* Răspunsurile HTTP returnate către client

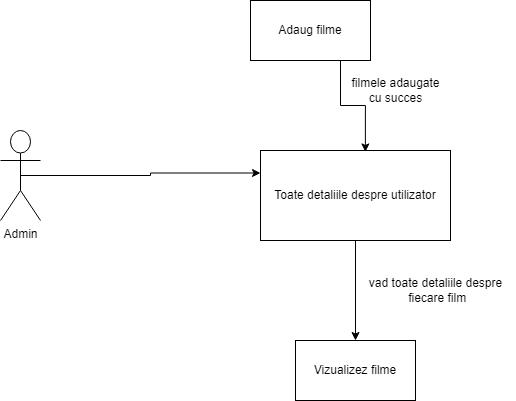


Diagrama de secventa View movies

Pentru partea de Frontend pasii sunt urmatorii :

1. Ma loghez ca si User
2. Intru pe pagina de View Movies
3. Vizualizez lisa de filme
4. Apar detaliile despre filme

Pentru partea de Backend pasii sunt urmatorii :

1. Autentific si returnez tokenul JWT
2. Returnez lista de filme pt a vizualiza userul
3. Returnez filmele in lista
4. Returnez detaliile despre filme

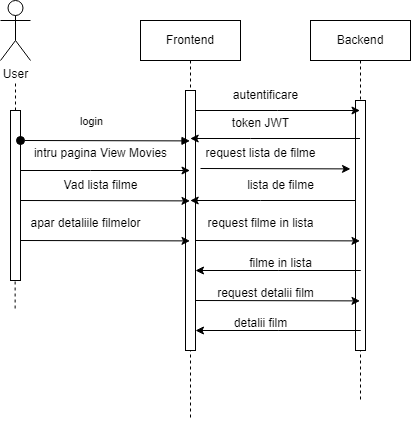
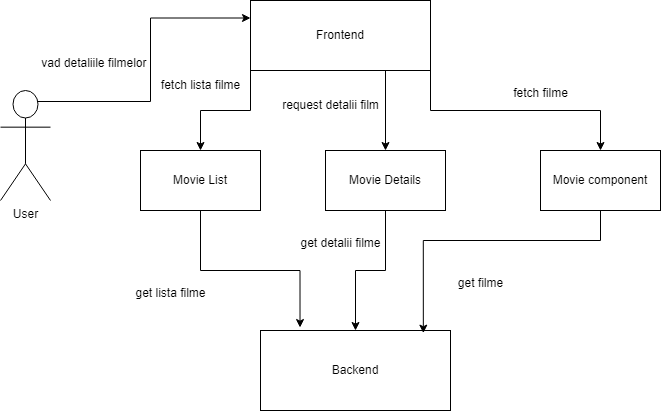
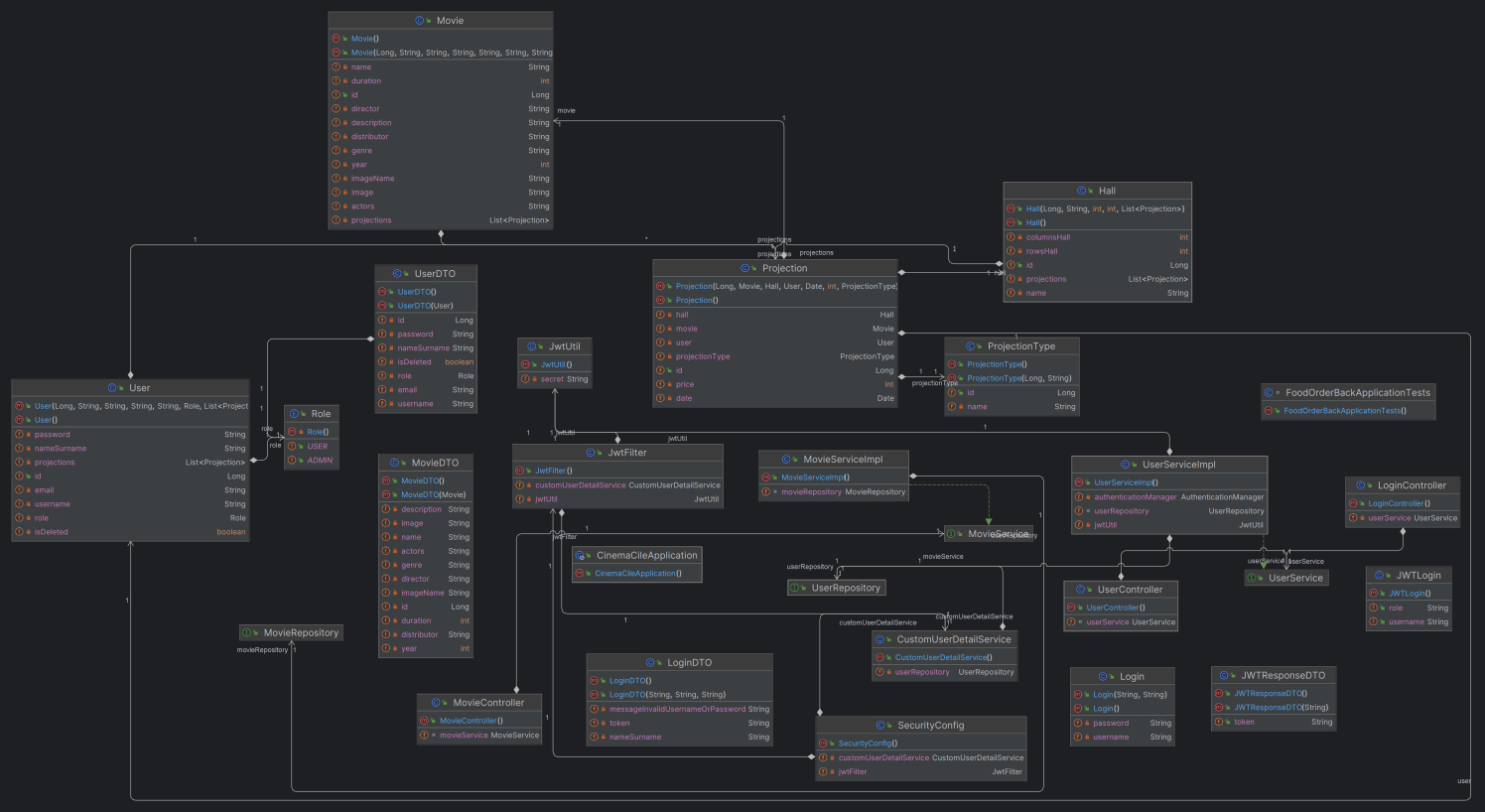


Diagrama de Comunicare View Movies



### Class Diagram



Avem urmatoarele GoF:

1. **Singleton Pattern**

Observație:

* Clasa JwtUtil este un bun candidat pentru patternul Singleton deoarece gestionează operațiuni legate de tokenuri JWT și ar trebui să existe o singură instanță a acestei clase pentru a centraliza și securiza manipularea tokenurilor.

1. **Factory Method Pattern**

Observație:

* Clasa UserServiceImpl poate folosi Factory Method pentru crearea obiectelor de tip User sau UserDTO, în funcție de cerințe.

- De asemenea, metodele din clasele Controller (cum ar fi MovieController sau UserController) ar putea folosi un Factory Method pentru a instanția și configura diverse DTO-uri (Data Transfer Objects) sau alte obiecte necesare.

1. **Decorator Pattern**

Observație:

- În cazul în care funcționalitățile legate de utilizatori (User) și filme (Movie) trebuie extinse dinamic cu comportamente suplimentare, patternul Decorator poate fi aplicat. De exemplu, dacă dorim să adăugăm dinamic comportamente de auditare sau logging, putem folosi Decorator.

1. **Observer Pattern**

Observație:

- Dacă există necesitatea de a notifica diferite componente ale sistemului despre anumite evenimente (ex. actualizarea informațiilor unui utilizator sau a unui film), patternul Observer poate fi implementat. De exemplu, un sistem de notificări poate fi adăugat pentru a trimite emailuri sau mesaje atunci când datele sunt modificate.

1. **Facade Pattern**

Observație:

- Clasele UserServiceImpl și MovieServiceImpl pot fi considerate fațade care simplifică interacțiunea cu sistemul. Ele oferă o interfață unificată pentru operațiuni complexe care implică mai multe clase din backend (ex. UserRepository, MovieRepository).

1. **Strategy Pattern**

Observație:

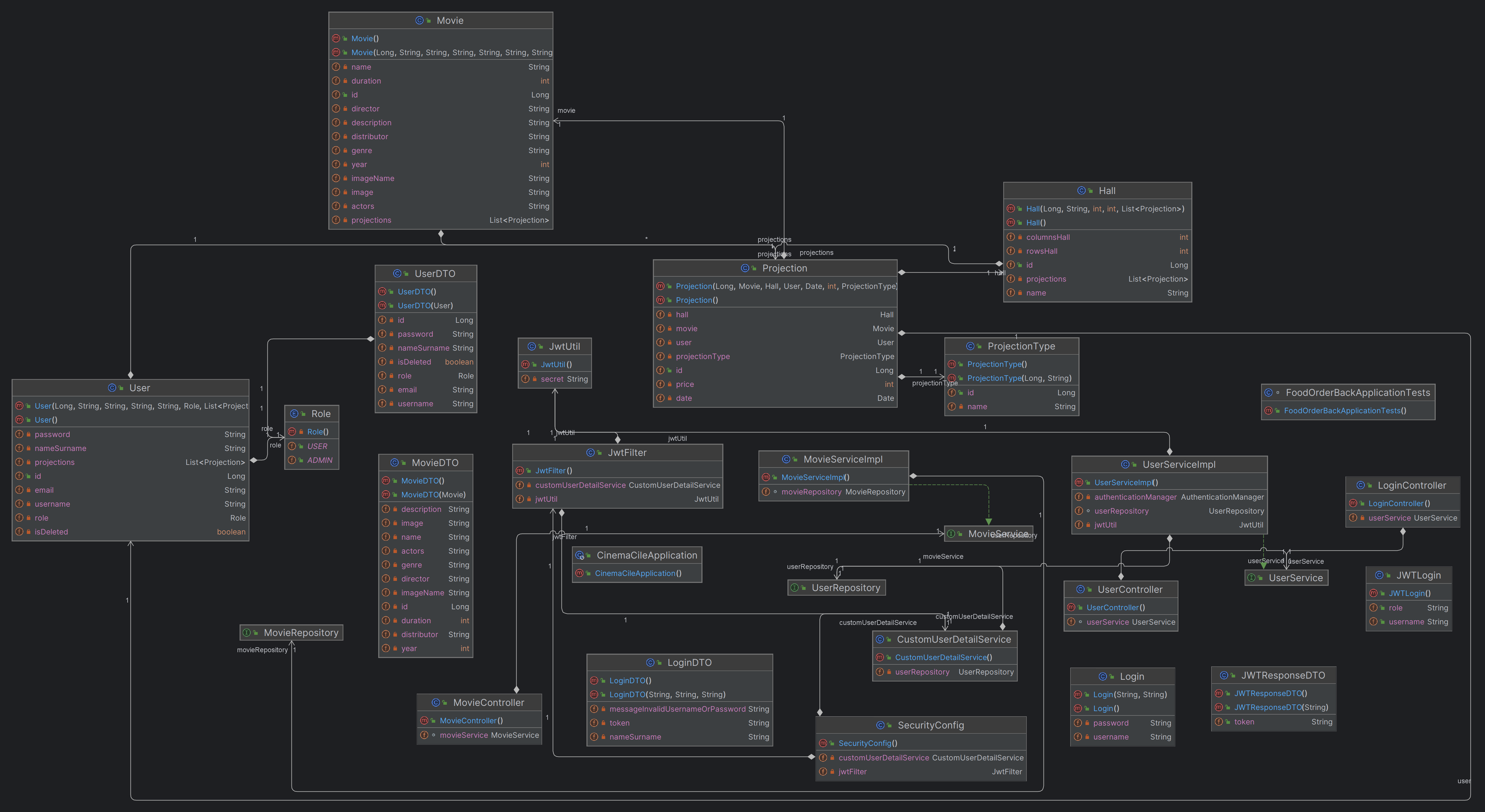
* Patternul Strategy ar putea fi utilizat pentru a gestiona autentificarea și autorizarea în diferite moduri. De exemplu, clasa SecurityConfig poate configura diverse strategii de autentificare (ex. prin JWT, OAuth, etc.).

Motivația alegerii patternurilor:

* Singleton pentru JwtUtil: Asigură că există o singură instanță care gestionează operațiunile cu tokenuri JWT, centralizând controlul și securitatea.
* Factory Method pentru DTO-uri și User: Simplifică crearea obiectelor complexe și permite extinderea facilă a sistemului fără a modifica codul existent.
* Decorator pentru extinderea funcționalităților: Permite adăugarea dinamică de noi comportamente (ex. logging, audit) fără a modifica clasele existente.
* Observer pentru notificări: Asigură actualizarea componentelor dependente în mod automat atunci când au loc schimbări relevante, îmbunătățind coeziunea și reducând cuplajul.
* Facade pentru servicii: Simplifică interacțiunea cu subsistemele complexe, oferind o interfață clară și ușor de utilizat.
* Strategy pentru autentificare și autorizare: Permite schimbarea dinamică a algoritmilor de autentificare și autorizare fără a modifica codul care utilizează aceste strategii.

## Data Model

Modelul de date, implementat în MySQL, include multiple relații de tip Many to many, în special între tabelul „utilizator” și tabelele „roluri”, „lista de filme”. Alte exemple de astfel de relații includ relația dintre „lista filme” și „filme”.



# System Testing

Testarea Unitară și Adnotările:

**Testarea unitară** verifică cele mai mici componente ale unei aplicații, precum metodele sau clasele, pentru a asigura corectitudinea logicii și a detecta erorile devreme în procesul de dezvoltare.

**Adnotările** marchează metodele sau clasele în timpul testării unitare:

* @BeforeEach: execută o metodă înainte de fiecare test.
* @Test: marchează metodele de testare.
* @Mock: desemnează obiectele simulate folosite în teste.

Construirea unui Test:

1. Inițializare și Pregătire: Inițializează obiectele și resursele necesare, de exemplu, cu MockitoAnnotations.openMocks(this) în metoda setUp().
2. Definirea Scenariului de Test: Stabilește parametrii de intrare și așteptările pentru metoda testată.
3. Execuția Testului: Rulează metoda de testat și obține rezultatul.
4. Validarea Rezultatului: Verifică dacă rezultatul corespunde așteptărilor folosind aserțiuni.
5. Curățarea Resurselor: Eliberează resursele utilizate pentru a nu interfera cu alte teste.

Importanța Testării Unitare:

Testarea unitară este esențială în dezvoltarea software-ului, oferind o metodă eficientă de a verifica funcționalitatea codului și de a preveni erorile. Testele unitare automate permit detectarea și remedierea rapidă a erorilor, conducând la un cod mai fiabil și mai ușor de întreținut.

# Future Improvements

Proiectul ar putea fi îmbunătățit prin adăugarea funcționalității de subtitrări la componenta de redare a filmelor. Subtitrările ar putea fi preluate automat prin REST de la servicii precum opensubtitles.org.

Funcționalitatea de căutare ar trebui adăugată odată cu creșterea bazei de date a filmelor. Pentru acces mai facil, ar putea fi implementate liste care să conțină filmele favorite.

De asemenea, ar fi util un sistem care să colecteze feedback-ul privind media vizionată.

# Conclusion

Proiectul a reușit să își atingă obiectivele prin proiectarea și implementarea unui backend robust de tip REST API pentru distribuirea conținutului media, combinat cu un frontend atractiv realizat în ReactJS.

Utilizând MySQL ca sistem de gestionare a bazelor de date, a fost creat un model de date scalabil și eficient pentru a stoca informații despre filme și entități legate de utilizatori. REST API-ul oferă o interfață intuitivă pentru utilizatori, permițându-le să efectueze operațiuni precum crearea și gestionarea listelor de media, accesarea informațiilor detaliate despre filme și multe altele.

Pe parcursul procesului de dezvoltare, au fost respectate cele mai bune practici în ingineria software, inclusiv designul modular și reutilizarea codului. Metodologiile de testare utilizate, cum ar fi testarea unitară cu JUnit, asigură fiabilitatea și corectitudinea codului, iar principiile RESTful facilitează interoperabilitatea și integrarea ușoară cu alte sisteme.

Pe viitor, proiectul ar putea îmbunătăți experiența utilizatorilor prin adăugarea funcționalității de subtitrări, implementarea unei funcții de căutare, introducerea listelor de favorite, integrarea unui sistem de feedback.

# Bibliography

* **React**  : <https://reactjs.org/docs/getting-started.html> , <https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs>
* **Spring :** <https://spring.io/projects/spring-framework>
* **Unit testing :** <https://aws.amazon.com/what-is/unit-testing/>
* **My SQL** : <https://www.oracle.com/mysql/what-is-mysql/>,

<https://www.mysql.com/about/>

* **Web Sockets :** <https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API>
* **MVC :** <https://www.geeksforgeeks.org/mvc-framework-introduction/>