Cinema Management System

Design Document

Cihan CAN, Umut AVCI, Samet AVCI, Mumtaz ERDOGAN

February 15, 2025

Contents

1 Introduction

1.1 Purpose

The Cinema Management System (CMS) is a backend system designed to manage various aspects of a cinema, including movie scheduling, seat reservations, and customer subscriptions. This document provides a detailed overview of the system architecture, design decisions, and implementation details.

1.2 Scope

The CMS provides functionalities such as:

- Movie and session scheduling
- Seat reservation and availability management
- Customer subscription and authentication
- REST API with HATEOAS, filtering, pagination, and caching
- Integration with PostgreSQL and Docker

2 System Overview

2.1 Architecture

The CMS follows the **Hexagonal Architecture**, ensuring separation of concerns:

- **Domain Layer:** Business logic and core entities
- **Application Layer:** Services managing business operations
- **Infrastructure Layer:** Database persistence and API exposure

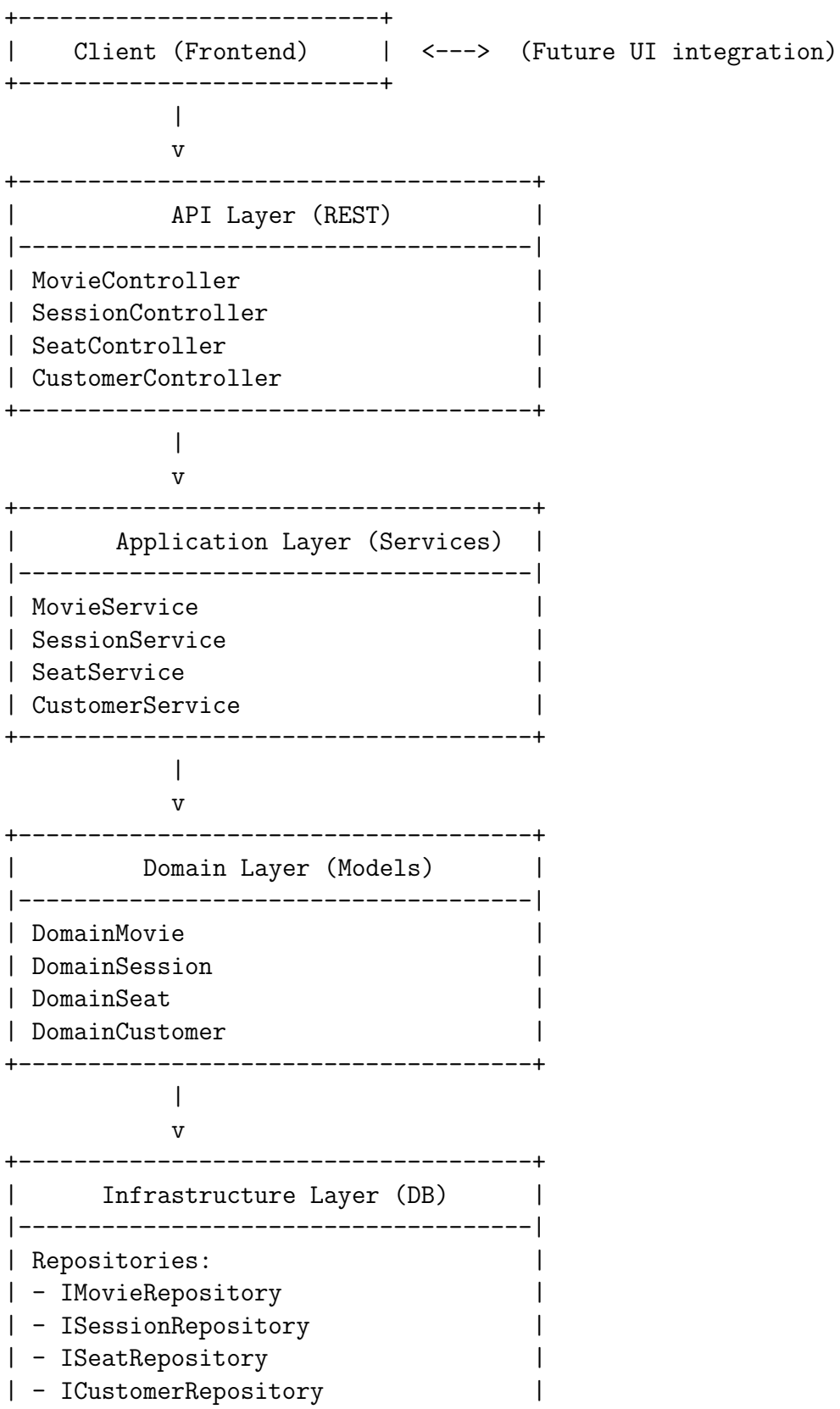
2.2 Technology Stack

- **Programming Language:** Java (Spring Boot)
- **Database:** PostgreSQL (JPA/Hibernate)
- **API:** RESTful API
- **Build Tool:** Maven
- **Containerization:** Docker & Testcontainers
- **Testing:** JUnit, Mockito, Testcontainers

3 System Architecture

3.1 Component Diagram

The high-level architecture is structured as follows:



```
| Persistence Entities                |
| Database (PostgreSQL)               |
+-------------------------------------+
```

3.2 Modules & Components

3.2.1 Domain Layer

Contains core business logic and models:

- **DomainMovie:** Represents movie details
- **DomainHall:** Represents a cinema hall
- **DomainSession:** Manages session scheduling
- **DomainSeat:** Handles seat availability
- **DomainCustomer:** Stores customer details

3.2.2 Application Layer

Handles business logic and transactions:

- **MovieService:** CRUD operations for movies
- **SessionService:** Manages movie sessions
- **SeatService:** Tracks seat availability
- **CustomerService:** Handles customer data

3.2.3 Infrastructure Layer

- **Controllers:** Expose REST API endpoints (e.g., MovieController, SessionController)
- **Repositories:** Provide database access (e.g., IMovieRepository, ISeatRepository)
- **Persistence Entities:** Maps domain models to the database

4 Database Schema

4.1 Tables & Relationships

- **1 Movie** can have **n Sessions**
- **1 Hall** can have **n Sessions**
- **1 Session** can have **n Seats**
- **1 Customer** can have **n Reservations**

5 API Design

5.1 REST Endpoints

HTTP Method	Endpoint	Description
GET	`/api/movies`	Fetch all movies
GET	`/api/movies/{id}`	Get movie details
POST	`/api/movies`	Add a new movie
PUT	`/api/movies/{id}`	Update movie details
DELETE	`/api/movies/{id}`	Delete a movie
GET	`/api/sessions`	List movie sessions

Table 1: API Endpoints Overview

6 Security & Authentication

- **Authentication:** JWT-based authentication
- **Authorization:** Role-based access control (Admin, Customer)
- **Security Measures:** Input validation, CSRF protection

7 Deployment & Testing

7.1 Docker Configuration

The system runs inside a Docker container. The configuration is as follows:

Listing 1: Docker Compose Configuration

```
version: '3.8'

services:
  database:
    image: postgres:15
    container_name: postgres_db
    restart: always
    environment:
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: admin
      POSTGRES_DB: cms
    ports:
      - "5433:5432"
    volumes:
      - pg_data:/var/lib/postgresql/data

  app:
    build: .
    container_name: cms_app
    restart: always
```

```
    depends_on:
      - database
    environment:
      SPRING_DATASOURCE_URL: jdbc:postgresql://database:5432/cms
      SPRING_DATASOURCE_USERNAME: admin
      SPRING_DATASOURCE_PASSWORD: admin
    ports:
      - "8080:8080"
    command: ["java", "-jar", "app.jar"]

volumes:
  pg_data:
```

7.2 Testing Strategy

- **Unit Tests:** Service and repository testing (JUnit, Mockito)
- **Integration Tests:** API and database testing (Testcontainers)
- **Performance Testing:** Stress tests for concurrent bookings

8 Conclusion

The Cinema Management System is designed with scalability and maintainability in mind. The use of REST API, Hexagonal Architecture, and Docker makes it a modern and efficient solution for cinema management.