<Company Name>
<Company Name>

<Moga Diana>
Supplementary Specification

Version <1.0>

Revision History

Date	Version	Description	Author
27/05/2025	1.0	Made all the document entirely.	Moga Diana

Table of Contents

Supplementary Specification

1. Introduction

This Supplementary Specification document outlines additional system requirements that are not fully captured in the use-case model of the Movie Management Application (MovieHub). It addresses legal and regulatory constraints, quality attributes such as usability, reliability, performance, and supportability, as well as environmental and design constraints.

MovieApp is designed to enable administrators to manage detailed information about movies, directors, and actors, while providing end users with capabilities to register, log in, browse movie data, leave reviews, maintain personalized watchlists, and receive notifications. This document details the supplementary aspects required to ensure a robust, secure, and high-quality system beyond the functional requirements.

2. Non-functional Requirements

The following sections define key system quality attributes, expressed in terms of quality attribute scenarios to describe the expected system behaviors under various conditions.

2.1 Availability

Quality attribute definition:
System should be available and ready to use almost all the time without unexpected downtime.

Source of stimulus:
Users trying to access the app.

Stimulus:
User opens the app or requests data (like movie list).

Environment:
Normal operating conditions (server running).

Artifact:
The whole MovieHub system (frontend and backend).

Response:
System responds and shows requested data.

2.2 Performance

- Quality attribute definition:

The app should be fast and responsive.

- Source of stimulus:

User clicking or searching.

- Stimulus:

User requests to view movies or submit reviews.

- Environment:

Normal network and server load.

- Artifact:

Backend API and frontend UI.

- Response:

Show requested data quickly; save user input without delay.

- Response measure:

Page loads under 3 seconds; API calls under 500 ms.

- Tactics:

Use efficient database queries; caching; limit data size sent.

2.3 Security

- Quality attribute definition:

Protect user data and restrict access properly.

- Source of stimulus:

Users and attackers.

- Stimulus:

User tries to log in or access restricted features.

- Environment:

Connected via internet with HTTPS.

- Artifact:

Authentication system and database.

- Response:

Allow access only to authorized users; block invalid attempts.

- Response measure:

No data leaks; passwords stored hashed; access control enforced.

- Tactics:

Use JWT tokens, password hashing, role checks, and HTTPS.

2.4 Testability

- Quality attribute definition:

System should be easy to test and maintain quality.

- Source of stimulus:

Developers running tests.

- Stimulus:

Executing automated unit or integration tests.

- Environment:

Test environment with sample data.

- Artifact:

Codebase and test suites.

- Response:

Tests run smoothly, catch errors, and help fix bugs.

- Response measure:

High test coverage; quick test execution.

- Tactics:

Write modular code; use mocking frameworks; isolate tests.

2.5 Usability

- Quality attribute definition:

Users should find the app easy and pleasant to use.

- Source of stimulus:

End users (both admins and regular users).

- Stimulus:

User navigates through the app or performs actions like adding to watchlist.

- Environment:

Desktop.

- Artifact:

User interface (frontend).

- Response:

Clear layout, easy navigation, and feedback on actions.

- Response measure:

User satisfaction surveys; low error rates in usage.

- Tactics:

Use consistent UI design; responsive layouts; clear buttons and messages.

3. Design Constraints

This section lists the fixed choices and rules that must be followed while building the MovieHub. These are decisions that cannot be changed easily because they guide how the system is designed and developed.

3.1 Software Languages and Frameworks

- The frontend must use **React** with **TypeScript** to build the user interface. This choice ensures components are reusable and types help catch errors early.
- The backend must use **Spring Boot** (Java framework) for building APIs and handling business logic.
- For security, **Spring Security** with **JWT (JSON Web Tokens)** is used to manage user authentication and access control.

3.2 Architectural Pattern

- The application follows a **four-tier architecture**:
  - Presentation Tier (frontend UI)
  - Application Tier (backend services and logic)
  - Data Tier (database and data management)
  - Testing Layer (for unit and integration testing)

3.3 Database

- The system uses **PostgreSQL**, a relational database, for storing all data including movies, users, reviews, and watchlists.
- The database is managed using **pgAdmin** as the administrative tool.

3.4 Object-Relational Mapping (ORM)

- The backend uses **Spring Data JPA with Hibernate** for interacting with the database through Java objects, simplifying data operations.
- **JPA Annotations** are used to define how data models map to database tables.

3.5 Notifications

- The system uses an **Observer Pattern** to send email notifications to users when a movie they are watching is released.
- Email sending is implemented with **Spring Mail** and **JavaMailSender**.

3.6 Code and Development Tools

- **Lombok** is used in the backend to reduce repetitive code such as getters, setters, and constructors.
- Data exchange between frontend and backend uses **DTOs (Data Transfer Objects)** for clear and secure data transfer.

3.7 Testing Tools

- Unit tests use **JUnit 5** and **Mockito** to check the business logic in isolation.
- Integration tests use **Spring Boot Test framework** and **MockMvc** to simulate real HTTP requests and verify API behavior.
- Test data is separated using **TestPropertySource** and **fixture JSON files** to make tests reliable and repeatable.
- Tests clean up after themselves using transactional rollbacks to keep the database consistent.

3.8 Process Constraints

- All development must follow **layered architecture principles**, separating concerns into models, controllers, services, and repositories.
- Passwords must be hashed before storage, and no plaintext passwords should ever be saved.