UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS

Universidad Distrital Francisco José de Caldas

Department of Systems Engineering

Technical Report

Nicolás Guevara Herrán
Samuel Antonio Sánchez Peña
Jorge Enrique Acosta Jiménez

Supervisor: Carlos Andrés Sierra

A report submitted in partial fulfillment of the requirements of
the Universidad Distrital Francisco José de Caldas for the degree of
Master of Science in *Data Science and Advanced Computing*

October 25, 2025

Abstract

This technical report presents the ongoing development of a *Cinema Management System* designed to automate catalog management, screening scheduling, and theater room allocation. The purpose of the project is to build a scalable and secure architecture capable of reducing manual errors and improving operational efficiency in cinema administration. The system follows a microservice-based design composed of two main modules: an authentication service built with Quarkus and Keycloak (using MySQL), and a business service developed with Flask and PostgreSQL. Both services communicate through RESTful APIs and are deployed in containerized environments using CI/CD pipelines.

The report outlines the background, problem definition, and architectural design of the system. It also describes the development methodology, design decisions, and validation plan that will guide the next stages of testing and performance assessment. Although the system is still under active development, the architectural foundation and service interaction have been successfully implemented. Future stages will include load testing, usability analysis, and full deployment in the Azure Cloud environment.

Keywords: microservices, cinema management, Flask, Quarkus, Keycloak

Contents

List of Figures

List of Tables

List of Abbreviations

SMPCS	School of Mathematical, Physical and Computational Sciences

Chapter 1

Introduction

1.1 Background

Cinema and theater management increasingly rely on software systems to automate operations such as movie cataloging, scheduling, and room allocation. Traditional systems—often monolithic or manually operated—tend to generate redundant data, inconsistent schedules, and scalability issues. With the growing adoption of cloud-native technologies, there is a need for modular, maintainable, and secure systems that can support multiple branches or theaters while maintaining centralized control and reliability.

1.2 Problem statement

Current cinema management systems often lack integration between authentication and business operations, leading to security vulnerabilities and poor maintainability. In addition, manual scheduling increases the likelihood of overlapping showtimes or room conflicts. Finally, deployments are rarely automated, which limits scalability and traceability. This project aims to address these issues by designing and implementing a microservice-based solution that separates authentication, business logic, and data management, ensuring secure and consistent operation across environments.

1.3 Aims and objectives

Aim: Develop a modular and secure cinema management system based on microservices that enables efficient catalog and scheduling management.
Objectives:

- Design and implement an authentication module using Quarkus and Keycloak to ensure secure access control.
- Develop a business service in Flask and PostgreSQL to manage movies, rooms, and screenings.
- Establish RESTful communication between services to ensure interoperability and scalability.
- Deploy the system in containerized environments with automated CI/CD pipelines.
- Define a testing and validation plan covering performance, usability, and data consistency.

1.4 Solution approach

The solution follows a microservice architecture with two independent yet interoperable services. The *Auth Service* handles authentication, authorization, and user management through Keycloak, while the *Cinema Service* provides endpoints for catalog and scheduling operations. Each service maintains its own database to ensure autonomy and reduce coupling. Communication occurs through REST APIs secured with access tokens. The system is containerized using Docker and deployed through automated pipelines (GitHub Actions → Azure Container Apps). This architecture allows independent scalability, fault isolation, and a clear separation of concerns, aligning with DevOps and cloud-native best practices.

1.5 Summary of current progress

So far, the architectural design has been completed, both services are functional in local and containerized environments, and integration between Keycloak and Flask has been validated. The next stage involves refining API endpoints, conducting usability and load testing, and implementing monitoring and deployment automation in the Azure cloud.

1.6 Organization of the report

Chapter **??** presents the literature review related to microservice architectures, identity management, and CI/CD practices. Chapter 8 describes the methodology and design process. Chapter **??** outlines the current system implementation and testing plan. Chapter **??** discusses preliminary findings and future work, followed by Chapter **??**, which summarizes the progress made so far and outlines the next development steps. Appendices include technical documentation, configuration details, and API specifications.

Chapter 2

Literature Review

2.1 Microservice Architecture for Admin Systems

The microservice architecture is a modern approach where a system is decomposed into small, independent services that focus on specific business capabilities (Fowler and Lewis, 2014; Samanta, 2023). This style improves scalability, maintainability, and fault isolation—each service can be updated or deployed independently. In the context of cinema management, this allows modules such as scheduling, user management, and ticket processing to evolve separately, offering operational agility.

Industry systems such as BookMyShow exemplify this structure, adopting service separation and API gateways to unify the interface (Samanta, 2023). However, microservices also bring complexity (e.g., inter-service communication, data consistency), which literature advises to balance with robust DevOps and CI/CD practices (Fowler and Lewis, 2014).

In our project, this architecture supports two major services—`Auth` and `Cinema`—allowing separation of identity from functional features, and enabling independent scaling or maintenance.

2.2 API-Driven Systems and DevOps Practices

API-first design aligns with microservice principles by establishing clear contracts between frontend and backend services (Sharma, 2023). This decoupling permits development and integration across services and clients, like mobile apps or partner platforms.

To manage the operational complexity of distributed systems, modern solutions adopt containerization (e.g., Docker) and CI/CD pipelines for consistent builds and deployments (Debois, 2011). In our system, each service is containerized and deployed via GitHub Actions to Azure, streamlining the delivery process.

DevOps practices—such as infrastructure-as-code and environment parity—ensure rapid iterations and system resilience, even in small teams.

2.3 Identity and Access Management (IAM)

In distributed systems, identity must be handled centrally. Literature supports delegating authentication to specialized identity providers like **Keycloak**, which issues JWT tokens used by microservices for authorization (Hat, 2022).

This approach provides Single Sign-On (SSO), role-based access control, and external identity federation. Keycloak's compliance with OAuth2/OpenID Connect ensures interoperability and security. In our architecture, the `Auth` service integrates with Keycloak, offloading credential handling and simplifying service-level access control.

2.4 Cinema System Design: Industry Comparison

Legacy cinema systems often lacked online access and used manual or on-premise setups (Acharya, 2023). Recent literature and case studies (e.g., BookMyShow) describe platforms that are API-driven, modular, and cloud-deployed (Samanta, 2023). These modern systems feature 24/7 availability, elastic scalability, and real-time booking.

Our project mirrors these principles but at a smaller scale—offering modular services, online administration, and token-based access control. Unlike older monolithic systems, our design simplifies integration and maintenance.

2.5 Critique of the Review

While microservices offer benefits, literature warns of complexity when overused (Fowler and Lewis, 2014). Our system carefully scopes service boundaries to avoid overhead. Many previous works emphasize architecture but lack discussion on DevOps or CI/CD, which our project adopts and documents explicitly.

Security-wise, older systems implement in-house authentication, often neglecting SSO and token standards. Our Keycloak integration reflects best practices supported by modern IAM literature (Hat, 2022).

2.6 Summary

This chapter reviewed microservices, DevOps, IAM, and domain-specific cinema systems. We established that our architecture is aligned with best practices in modularity, automation, and security. By comparing with legacy and modern platforms, the review justifies the project's design decisions and identifies contributions to system maintainability and scalability.

Chapter 3

Background

This section introduces the conceptual and technical background relevant to the project. It covers the context of cinema management platforms as well as the modern architectural paradigms that support scalable and maintainable web systems.

3.1 Domain Overview: Cinema Management Systems

Cinema management systems are specialized software platforms designed to oversee and streamline the operations of movie theaters. These systems typically handle a variety of administrative tasks such as movie scheduling, screening room allocation, seat reservation, ticket pricing, promotions, and real-time occupancy tracking. Additionally, they may include modules for customer relationship management (CRM), inventory control, and reporting.

The digitalization of the entertainment industry has amplified the demand for systems that not only provide operational control but also integrate seamlessly with online platforms, mobile applications, and third-party services. Furthermore, administrative portals must be user-friendly and secure, enabling authorized personnel to manage data efficiently while minimizing operational downtime and human error. For this reason, cinema management systems increasingly adopt modular, API-driven architectures to separate concerns and improve maintainability.

3.2 Web Systems and Modular Architectures

Modern web systems are commonly built on layered or modular architectures that separate core functionalities into distinct services or components. A common approach is the client-server model, where the frontend (typically a web interface or SPA) communicates with backend services via RESTful APIs.

One prominent design paradigm is the microservices architecture. Instead of developing a single, monolithic backend, the system is decomposed into multiple loosely coupled services, each responsible for a specific business capability. These services communicate via HTTP or messaging protocols and are often containerized using platforms like Docker to facilitate independent deployment, scaling, and fault isolation.

Such modularity allows teams to work in parallel on different parts of the system, increases flexibility in choosing technology stacks, and enhances overall system resilience. This approach is particularly advantageous for administrative platforms, such as cinema management systems, where multiple stakeholders interact with different modules (e.g., movie scheduling, seating configuration, sales analytics) with varying performance and availability requirements.

Chapter 4

Objectives

The primary aim of this project is to design, develop, and deliver a fully functional, modular web-based cinema management application that demonstrates the integration of software engineering principles, full-stack development, and modern DevOps practices. The application leverages a microservice architecture composed of two independent backends and a frontend, and showcases the practical use of Java and Python for different service domains.

Objectives:

1. Analyze system requirements and translate them into well-structured user stories supported by a user story map; apply Scrum methodology throughout the project lifecycle to guide incremental development and ensure continuous delivery.
2. Develop two backend services (authentication and business logic) using Java and Python respectively, following object-oriented and modular design principles; ensure the system meets functional requirements through automated unit, acceptance, and performance testing.
3. Integrate the system into a DevOps pipeline by implementing CI/CD with GitHub Actions and deploying containerized services using Docker Compose; document the architecture, development process, and testing strategy comprehensively.

Chapter 5

Scope

This project focuses on the development of a modular cinema management system composed of multiple backend services. These include components responsible for core business logic, user authentication and authorization, and integration layers that support secure access and interoperability. The system is designed using a layered architecture and exposes RESTful APIs for interaction with external clients and administrative tools.

The scope covers system analysis, backend development using different technologies, API design, automated testing (unit and acceptance), containerization of services, and the setup of CI/CD pipelines for continuous integration and deployment. Identity and access management is handled through the integration of an external authentication provider configured as part of the system's infrastructure.

Excluded from the scope are the development of a frontend user interface, payment system integrations, mobile application support, and advanced reporting or analytics. The focus is strictly on backend service orchestration, modular architecture, and DevOps practices aimed at building a scalable and maintainable system foundation.

Chapter 6

Assumptions

The development of the cinema management system relies on several assumptions to simplify the implementation and maintain clarity within the project's limited scope:

- The application targets administrative use only; customer-facing interfaces such as public booking, ticket sales, or seat selection are not included.
- User roles are limited to two predefined types: *Administrator* and *Customer*, with basic role-based access control handled by an external identity provider (Keycloak).
- Authentication, user promotion, and user activation/deactivation are managed externally and exposed through an integration layer without requiring custom identity logic within the core application.
- All business operations are simplified to CRUD actions on movies and screenings. Complex scheduling features such as overlapping conflict resolution or theater optimization are not implemented.
- Movie and screening data are assumed to be entered and maintained manually by administrators, and there is no automated synchronization with external content providers.
- The system is designed to run in a local or containerized environment. Scalability, multi-tenant support, or cloud-native features are beyond the intended scope.

Chapter 7

Limitations

The development of the cinema management system was subject to several limitations that influenced design decisions, implementation depth, and the overall scope of the project. These constraints are outlined below to contextualize the results and delimit expectations:

- **Budget Constraints:** The project was developed with no dedicated funding. As such, no paid infrastructure, premium software tools, or third-party APIs were used. All components were restricted to free, open-source, or community editions.
- **Time Constraints:** Given the academic nature of the project, development was carried out within a fixed semester timeframe. This limited the depth of testing, documentation, and refinement of certain features.
- **Reduced Feature Set:** To keep the system manageable, features such as payment processing, seat reservation, user notifications, and reporting dashboards were excluded.
- **No Real-World Data:** All testing and validation were conducted using synthetic or sample data, as access to actual cinema operational data was not available.
- **Limited Cloud Infrastructure:** While the system is deployed to a cloud environment, infrastructure configuration is minimal and oriented toward academic validation. The project does not include high-availability setups, automated scaling, or load balancing.
- **Simplified Role Management:** Although Keycloak enables advanced role hierarchies and realm configurations, the implementation limited itself to two basic roles to avoid additional configuration complexity.

Chapter 8

Methodology

This chapter outlines the methodology proposed for the design, implementation, and evaluation of the cinema management system. The project adopts an agile and modular approach, emphasizing iterative development, separation of concerns, and automation for testing and deployment. It integrates diagrams to illustrate both technical design and business flow, ensuring that the methodology is transparent and reproducible.

8.1 Development Approach

An agile methodology based on Scrum principles has been adopted. Key practices include:

- **User Story Mapping:** Functional requirements are organized using a user story map to define core use cases for authentication, user management, movie management, and screening scheduling.
- **Sprints and Deliverables:** The project is structured into short sprints, each aiming to deliver deployable components. Sprint planning, reviews, and retrospectives guide progress and support incremental improvements.
- **Version Control and CI/CD:** GitHub is used for source control. Continuous integration and deployment pipelines are configured via GitHub Actions to automate testing, building, and deployment of services.

8.2 Architecture and Technologies

The system is designed as a modular microservices architecture. It consists of three core components: a React.js frontend, a Python Flask service for cinema business logic, and a Java Quarkus service for user and role administration through Keycloak. Each service is isolated by responsibility and communicates over HTTPS.

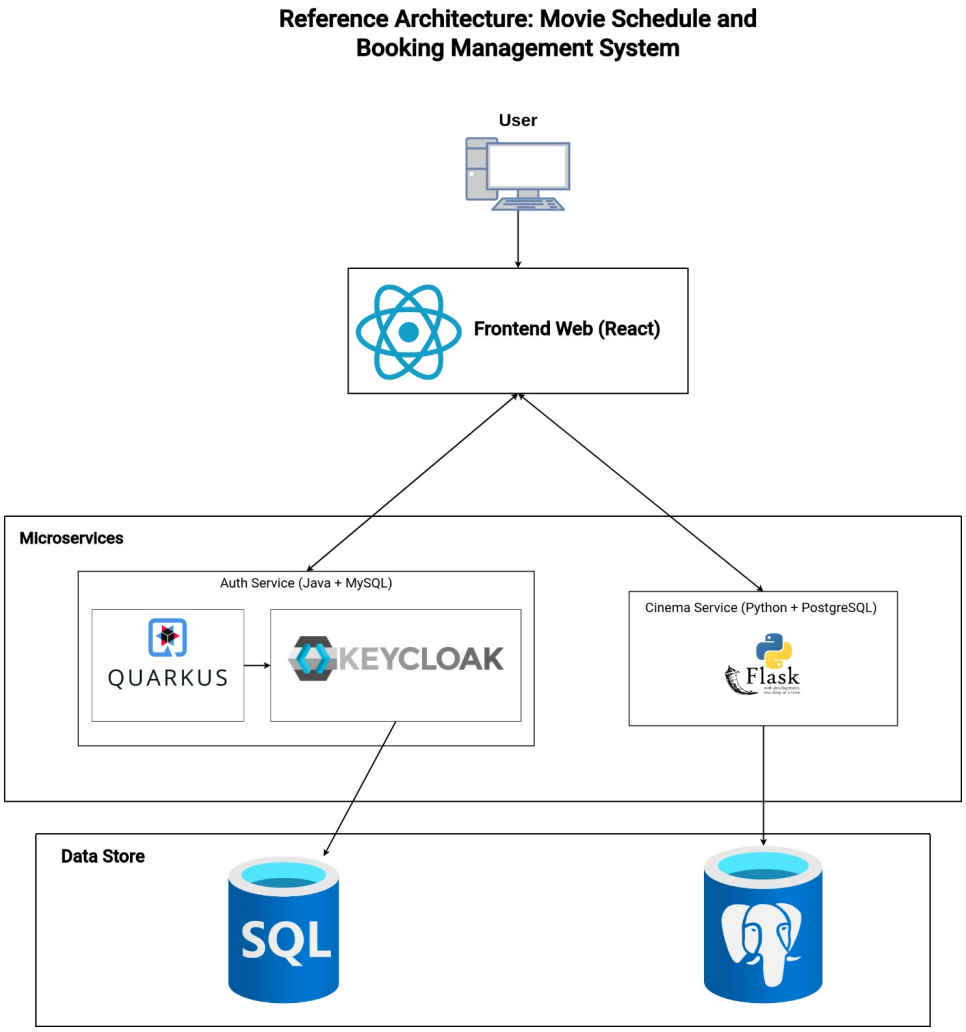

Figure 8.1: System architecture overview with technology stack.

8.3 Deployment Strategy

The deployment environment leverages Google Cloud. Services are containerized and deployed independently using Cloud Run. Firebase hosts the static frontend, and secure routing is established for user-facing APIs and internal service calls.

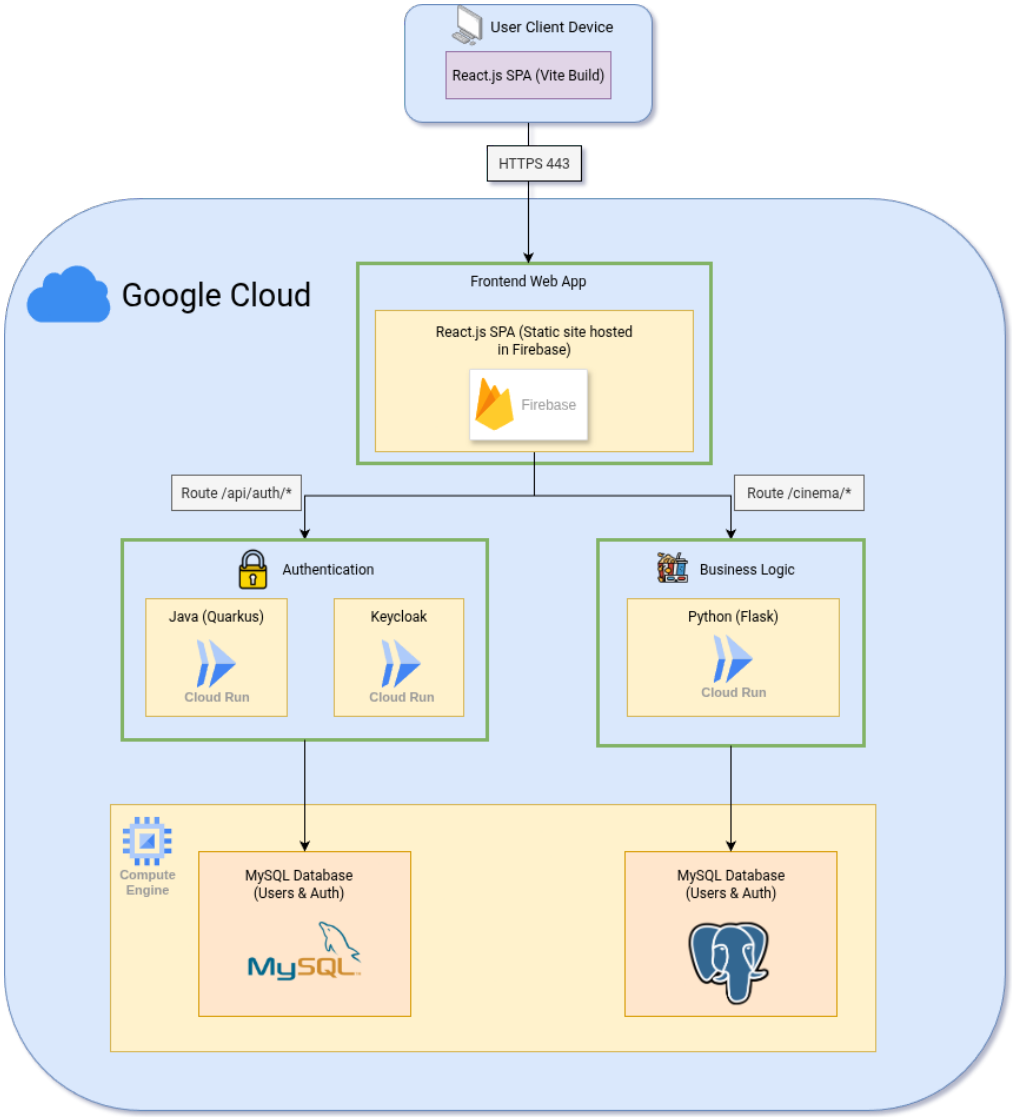

Figure 8.2: Deployment diagram for services hosted on Google Cloud.

8.4 Database Design

Each backend manages its own database schema tailored to its responsibilities. The Python service manages movies, screenings, and theater rooms. Relationships are defined to ensure referential integrity and support key domain constraints.

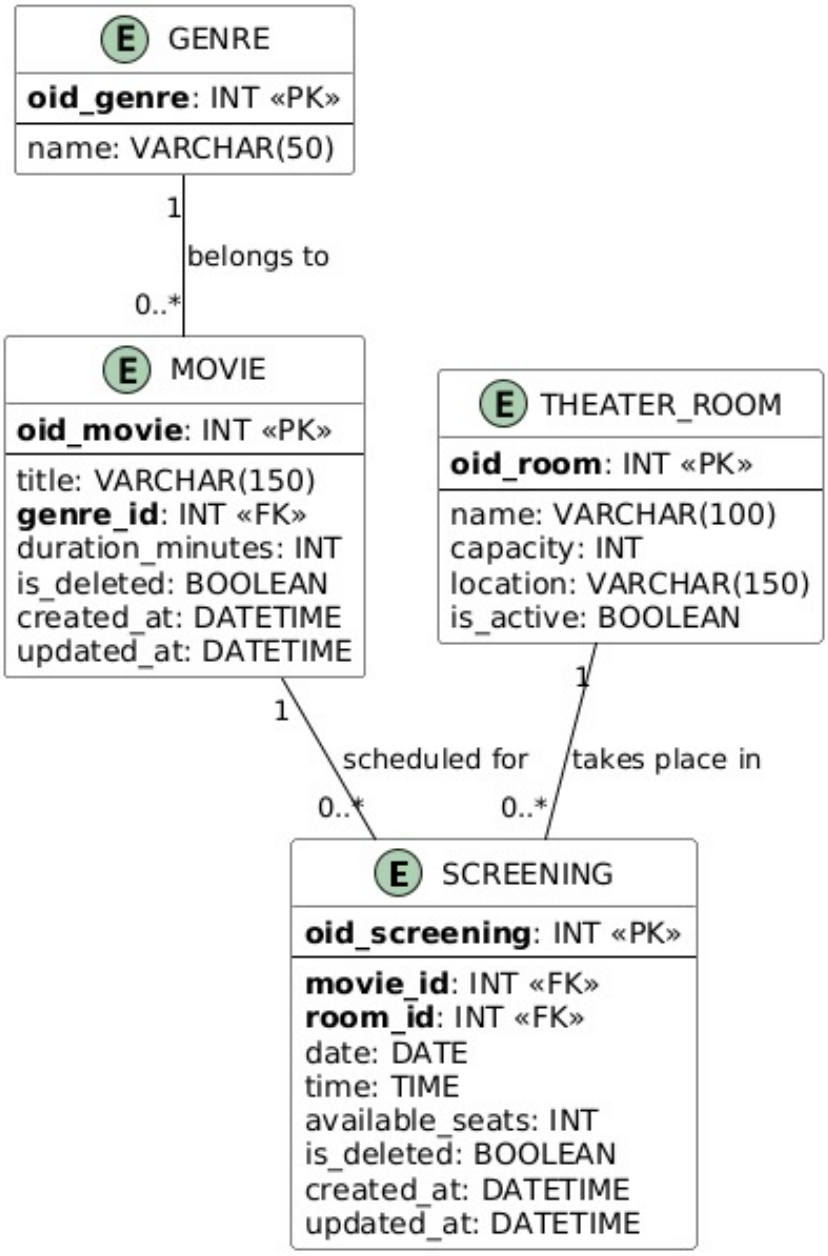

Figure 8.3: Entity-relationship diagram of the cinema management database.

8.5 Object-Oriented Design

The Python service uses object-oriented principles to separate domain logic and API layers. Controllers process incoming requests, delegate logic to services, and manipulate entity classes.

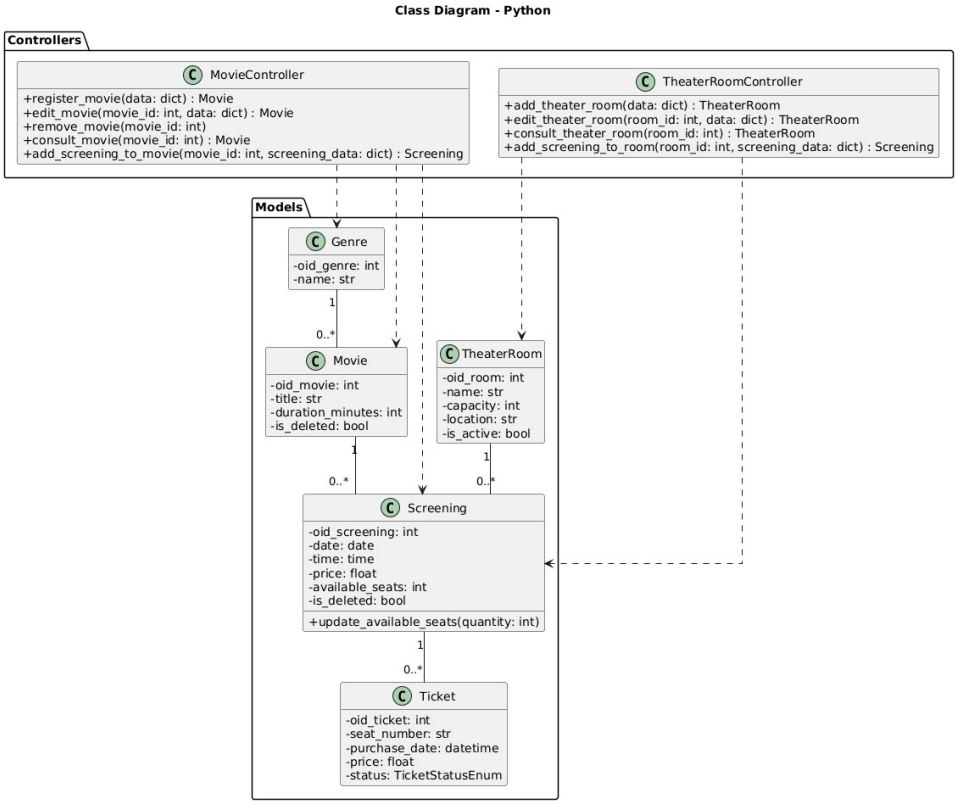

Figure 8.4: Class diagram for the Python Flask service.

The Java service handles Keycloak integration and exposes administrative endpoints for managing users and roles. It encapsulates the Keycloak Admin API and provides abstraction over authentication flows.

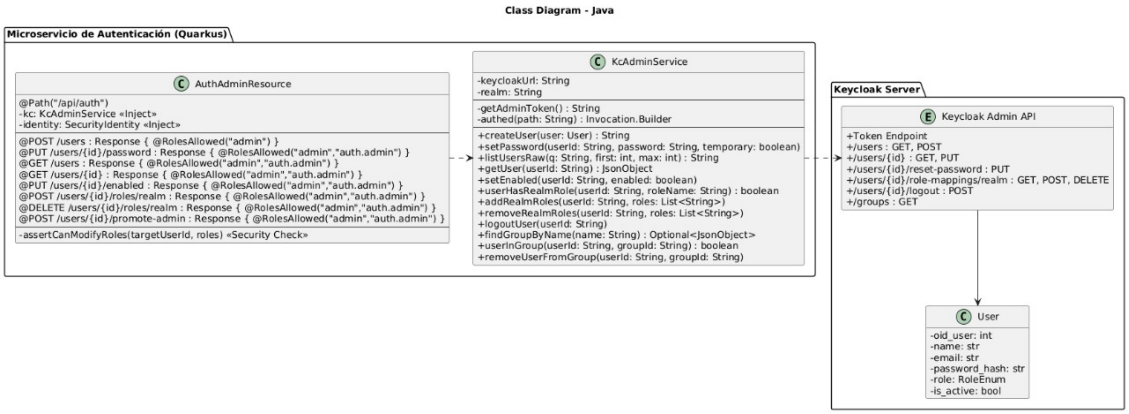

Figure 8.5: Class diagram for the Java Quarkus service managing Keycloak.

8.6 Business Process Modeling

To support user-centered design, a high-level process model was created. This outlines the interactions between customers, cinema staff, and content providers when reserving and managing screenings.

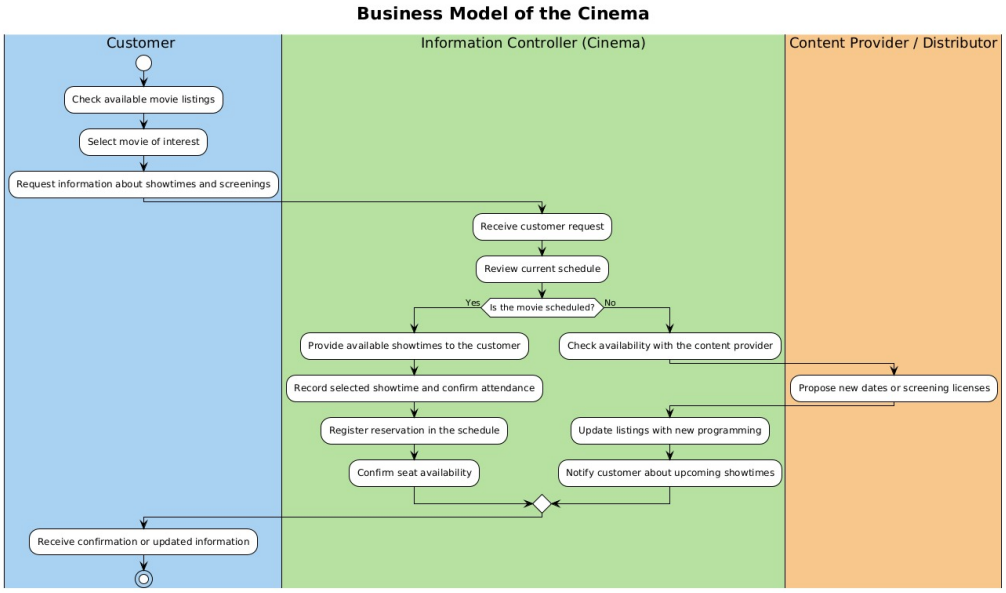

Figure 8.6: Business process model of cinema screening management.

8.7 Testing and Validation

The following testing strategy is planned to ensure robustness and correctness:

- **Unit Testing:** Functional modules will be tested using `pytest` for Python and `JUnit` for Java.
- **Integration Testing:** Containerized services will be tested together using tools like Postman and cURL to validate interactions.
- **Acceptance Testing:** Cucumber will be used to define and validate Gherkin scenarios based on user stories.
- **Performance Testing:** JMeter will simulate concurrent requests to evaluate latency and response stability.

8.8 Deployment and Infrastructure

The deployment strategy includes the following steps:

- **Containerization:** All services are containerized using Docker, ensuring environment consistency.
- **Local Orchestration:** Docker Compose will be used in development for multi-service orchestration and testing.
- **Cloud Deployment:** CI/CD pipelines push containers to Google Cloud Run. The frontend is hosted in Firebase.
- **Secure Communication:** All endpoints are exposed over HTTPS, and private services communicate securely within the cloud network.

This methodology promotes modularity, traceability, and reproducibility, while aligning with industry practices for secure and scalable web application development.

References

Acharya, K. (2023), 'Web based cinema seat allocation system – case study kathmandu plaza'. Internship Report, SSRN ID: 4841209.

Debois, P. (2011), 'Devops: A software revolution in the making', *Agile Conference (IEEE)* .

Fowler, M. and Lewis, J. (2014), 'Microservices: a definition of this new architectural term'. (accessed Oct 2025).
URL: *https://martinfowler.com/articles/microservices.html*

Hat, R. (2022), 'Keycloak documentation'. (accessed Oct 2025).
URL: *https://www.keycloak.org/documentation.html*

Samanta, P. (2023), 'Online movie ticket booking platform – system design (e.g. bookmyshow)'. (accessed Oct 2025).
URL: *https://medium.com/@psamanta*

Sharma, R. (2023), 'Best practices for ci/cd in microservices architecture'. (accessed Oct 2025).
URL: *https://dev.to/ravisharma/best-practices-ci-cd-microservices*