Cinema Management System: A Microservice-Based Architecture for Automated Scheduling and Catalog Management

Nicolás Guevara Herrán*, Samuel Antonio Sánchez Peña†, Jorge Enrique Acosta‡
Dept. of Computer Engineering, Universidad Distrital Francisco José de Caldas
Email: *nguevarah@udistrital.edu.co, †samasanchezp@udistrital.edu.co, ‡jeacostaj@udistrital.edu.co

***Abstract*—This paper presents a Cinema Management System designed under a microservice architecture to automate catalog management, scheduling, and room allocation. The system combines a Java-based authentication service (Quarkus + Keycloak + MySQL) with a Python-based business service (Flask + PostgreSQL), both exposed via RESTful APIs. The design emphasizes separation of concerns, secure identity management, and containerized deployment. We describe the architecture, design rationale, and an evaluation plan covering performance, correctness of scheduling (no overlaps), security, and usability. This work-in-progress contributes a reproducible blueprint and metrics for assessing microservice-based administrative systems in the cinema domain.**

***Index Terms*—Cinema management, microservices, REST API, Quarkus, Flask, PostgreSQL, Keycloak, Agile development.**

I. Introduction

Modern cinema and theater operations face increasing demands for digitalization, from automating scheduling and catalog updates to ensuring consistent customer experiences across multiple venues. Manual management often leads to redundant data, overlapping showtimes, and difficulties scaling to new locations. Consequently, there is a growing need for modular, maintainable, and secure information systems [1].

This research introduces a *Cinema Management System* (CMS) designed to automate the main administrative workflows of a cinema complex. The solution integrates microservice design principles to decouple authentication, catalog management, and scheduling tasks into independently deployable modules.

The architecture comprises two interoperating services: (1) an authentication service built with **Quarkus** and secured by **Keycloak**, using MySQL for persistence; and (2) a business service developed in **Flask** with a **PostgreSQL** database, managing movies, showtimes, and screening logistics [2].

This modular approach promotes scalability and fault isolation while enabling continuous integration and deployment (CI/CD) pipelines [3]. The work applies agile methodologies, with functionality defined through user stories and validated through iterative testing [4].

The main contribution of this paper lies in demonstrating how a hybrid Java–Python microservice architecture can effectively manage operational complexity in cinema scheduling, delivering both technical efficiency and improved usability.

II. Methods and Materials

A. System Design Approach

The system was developed under an agile methodology with a strong emphasis on iterative feedback and modular design. Functional requirements were translated into user stories with specific acceptance criteria, enabling regular validation throughout the development process.

At the core of the design is a microservice architecture, where concerns are clearly separated between two main services. The first service is responsible for authentication and user management, implemented in Java using the Quarkus framework and Keycloak as the identity provider. This service interfaces with a MySQL database to store credential and role-related data. The second service, developed in Python using Flask, handles the business logic of the cinema, including movie registration, screening scheduling, and theater room management. It connects to a PostgreSQL database for persistent storage of domain entities.

Communication between these services is facilitated through RESTful APIs over HTTPS, ensuring interoperability and scalability. Each microservice is independently containerized and deployed to a managed cloud environment, with continuous integration and delivery pipelines orchestrated via GitHub Actions.

The design of the Python-based cinema service follows object-oriented principles to encapsulate domain logic in well-defined classes. As shown in Fig. 1, the system includes entities such as *Movie*, *Screening*, *TheaterRoom*, and *Ticket*, each with relevant attributes and methods. Controllers such as *MovieController* and *TheaterRoomController* expose REST endpoints while delegating logic to these domain models. This separation enhances maintainability and testability, especially when validating complex operations like preventing screening overlaps or dynamically updating available seats.

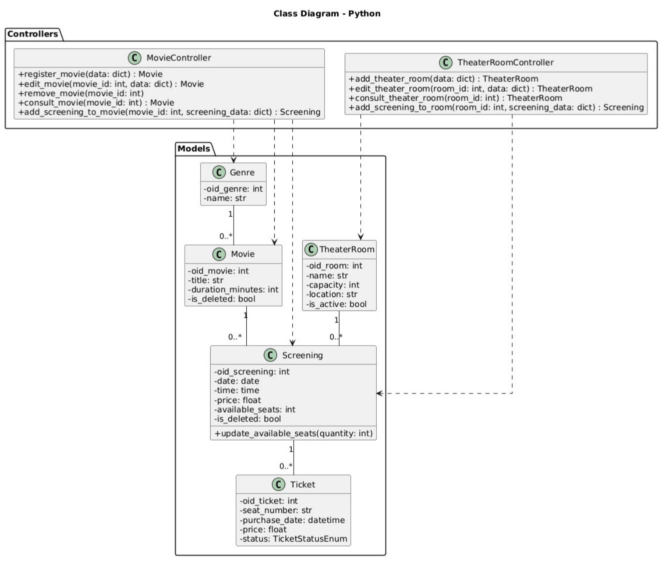
Class Diagram - Python
Controllers
MovieController
+register_movie(data: dict) : Movie
+edit_movie(movie_id: int, data: dict) : Movie
+remove_movie(movie_id: int)
+consult_movie(movie_id: int) : Movie
+add_screening_to_movie(movie_id: int, screening_data: dict) : Screening
TheaterRoomController
+add_theater_room(data: dict) : TheaterRoom
+edit_theater_room(room_id: int, data: dict) : TheaterRoom
+consult_theater_room(room_id: int) : TheaterRoom
+add_screening_to_room(room_id: int, screening_data: dict) : Screening
Models
Genre
-oid_genre: int
-name: str
1
0..*
Movie
-oid_movie: int
-title: str
-duration_minutes: int
-is_deleted: bool
TheaterRoom
-oid_room: int
-name: str
-capacity: int
-location: str
-is_active: bool
1
0..*
1
0..*
Screening
-oid_screening: int
-date: date
-time: time
-price: float
-available_seats: int
-is_deleted: bool
+update_available_seats(quantity: int)
1
0..*
Ticket
-oid_ticket: int
-seat_number: str
-purchase_date: datetime
-price: float
-status: TicketStatusEnum

Fig. 1. Class diagram of the Python-based Cinema Service.

This object-oriented approach also supports extensibility. For example, introducing pricing tiers or multi-language catalog descriptions would involve minimal changes to existing entities. However, this structure assumes synchronous communication and consistent state across services, which may introduce limitations in distributed scenarios where eventual consistency is required.

B. User Stories

The functional requirements were captured through user stories aligned with Scrum methodology:

- **HU1 – Movie Registration:** As an administrator, I want to register a movie or play with title, genre, and duration so that it becomes available in the catalog.
- **HU2 – Movie Consultation:** As a user, I want to view available movies or plays so that I can choose which one to attend.
- **HU3 – Movie Modification:** As an administrator, I want to edit movie or play information so that the catalog remains updated.
- **HU4 – Movie Deletion:** As an administrator, I want to remove unavailable movies or plays so that the database remains consistent.
- **HU5 – Screening/Theater Room Registration:** As a cinema administrator, I want to register screenings by specifying date, time, theater room, and movie so that I can schedule and organize movie projections in the system.
- **HU6 – Movie-Screening Relationship:** As a cinema administrator, I want to associate a movie with multiple screenings, so that the system correctly manages the schedule and users see all options available.
- **HU7 – Screening Query:** As a cinema user, I want to see all screenings for a movie with times, rooms, and availability, so that I can choose the most convenient option.
- **HU8 – Screening Deletion:** As a cinema administrator, I want to delete screenings with confirmation and ticket handling, so that the schedule stays accurate and users aren't misled.

C. System Architecture

The reference architecture (Fig. 2) follows a layered microservice model. The frontend (React) interacts with an API Gateway that routes requests to both microservices. Each service manages its own database instance, ensuring data independence and scalability.

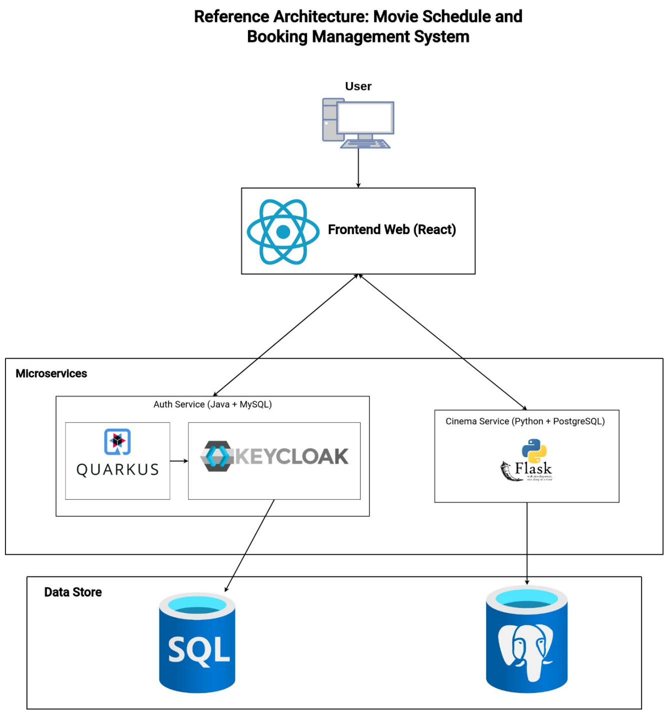
Reference Architecture: Movie Schedule and Booking Management System
User
Frontend Web (React)
Microservices
Auth Service (Java + MySQL)
QUARKUS
KEYCLOAK
Cinema Service (Python + PostgreSQL)
Flask
Data Store
SQL

Fig. 2. System Architecture of the Cinema Management Platform.

Finally, the deployment diagram (Fig. 3) shows the architecture deployed on Microsoft Azure. The CI/CD pipeline uses GitHub Actions to build and push Docker images to the Azure Container Registry, automatically deploying them to Container Apps. The frontend operates in Azure Web Apps, connecting via HTTPS to the gateway that routes requests to each microservice.

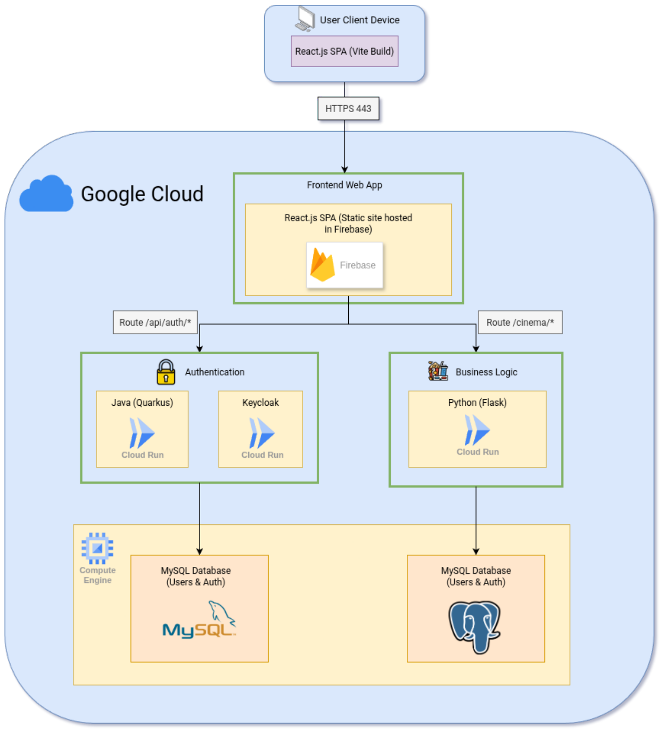

Fig. 3. Deployment on Azure Cloud Infrastructure.

III. Conclusions

References

[1] M. Fowler and J. Lewis, "Microservices: a definition of this new architectural term," *martinfowler.com*, 2015. [Online]. Available: https://martinfowler.com/articles/microservices.html

[2] M. Grinberg, *Flask Web Development: Developing Web Applications with Python*. O'Reilly Media, 2018.

[3] P. Debois, "DevOps: A Software Revolution in the Making," in *Agile Conference*, IEEE, 2011.

[4] K. Schwaber and J. Sutherland, *The Scrum Guide*, Scrum.org, 2020. [Online]. Available: https://scrumguides.org

[5] Red Hat, "Keycloak Documentation," *Red Hat Developer*, 2022. [Online]. Available: https://www.keycloak.org/documentation.html

[6] PostgreSQL Global Development Group, "PostgreSQL 15 Documentation," 2023. [Online]. Available: https://www.postgresql.org/docs/15/