# Design Document - Ticket Booking APP

## Contents

# Design Document - Ticket Booking APP

## 1. Understanding Requirement:

Ticket Booking App for simplifying Theatre Owners and Cinema lovers. Implemented Below Functionalities and exposed APIs'.

**User** – CRUD – User creation to on board B2B and B2C users

Two ways user can be created, either user can directly register with their details. In this API, user should mention in user type whether he/she B2B/B2C. Admin can create with roles

**Role** – CRUD - Role should be assigned to any user whether B2B (Partner – PARTNER ROLE) or B2C (end user – CUSTOMER ROLE), ADMIN ROLE has been created for who can initiate all. Role can be created only by Admin. Admin user and Admin Role can be created as part of Initialization Process.

**Login** – Any user can access to get token

**Theatre** – CRUD -Partner can create, as part of this API, user can create Theatre as well as screens associated with the theatre

**Movie** – CRUD – As part of this POST API call, Movie object as well as screening objects also created.

**Screening** – is nothing but list of theatres which are associated with the list of screens

**Search** – En user can search Movies by name and Movies by name and Date

**Booking** – End user can book the ticket

**Payment** – This is implemented by integrating with Razor pay, had to create key and secret. Used my existing credentials.

**Currency Conversion** – For this had to consume their part API , exchange rate host – Have done SSRF check also.

**CRUD** – for all of the above domain/model/entity – there is a place holder for creating all the CRUD operations as well as additional GET APIs'. Due to time crunch, not done for all.

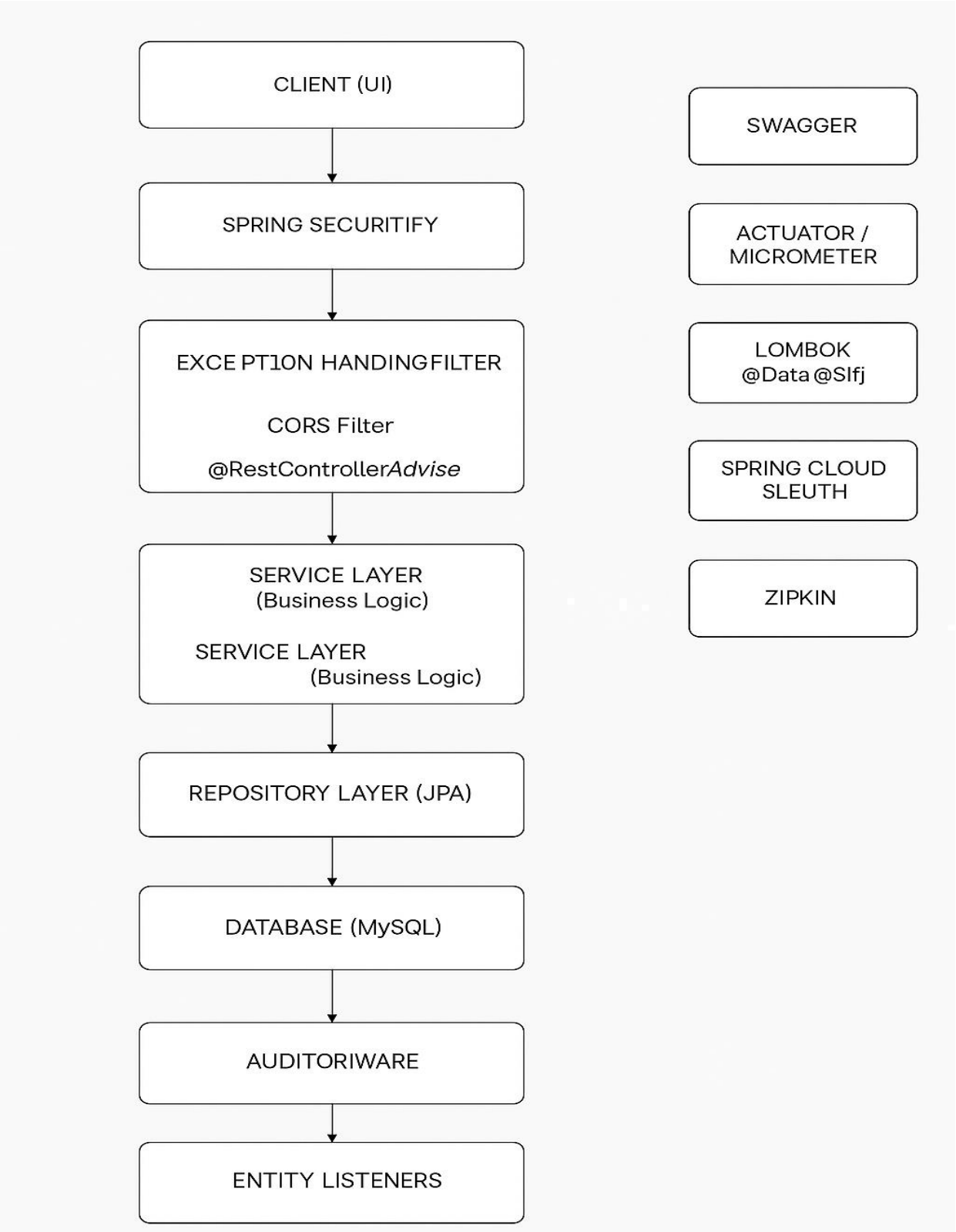
# 2. Users and Roles

There should be minimum there 3 types of users. Admin who can support for initial support and Data uploading, Partners' Theatre owners who can attach their theatres to this app and they can allow the end users to book the tickets online. Customers are the consumers who can search for tickets and book tickets for movies.

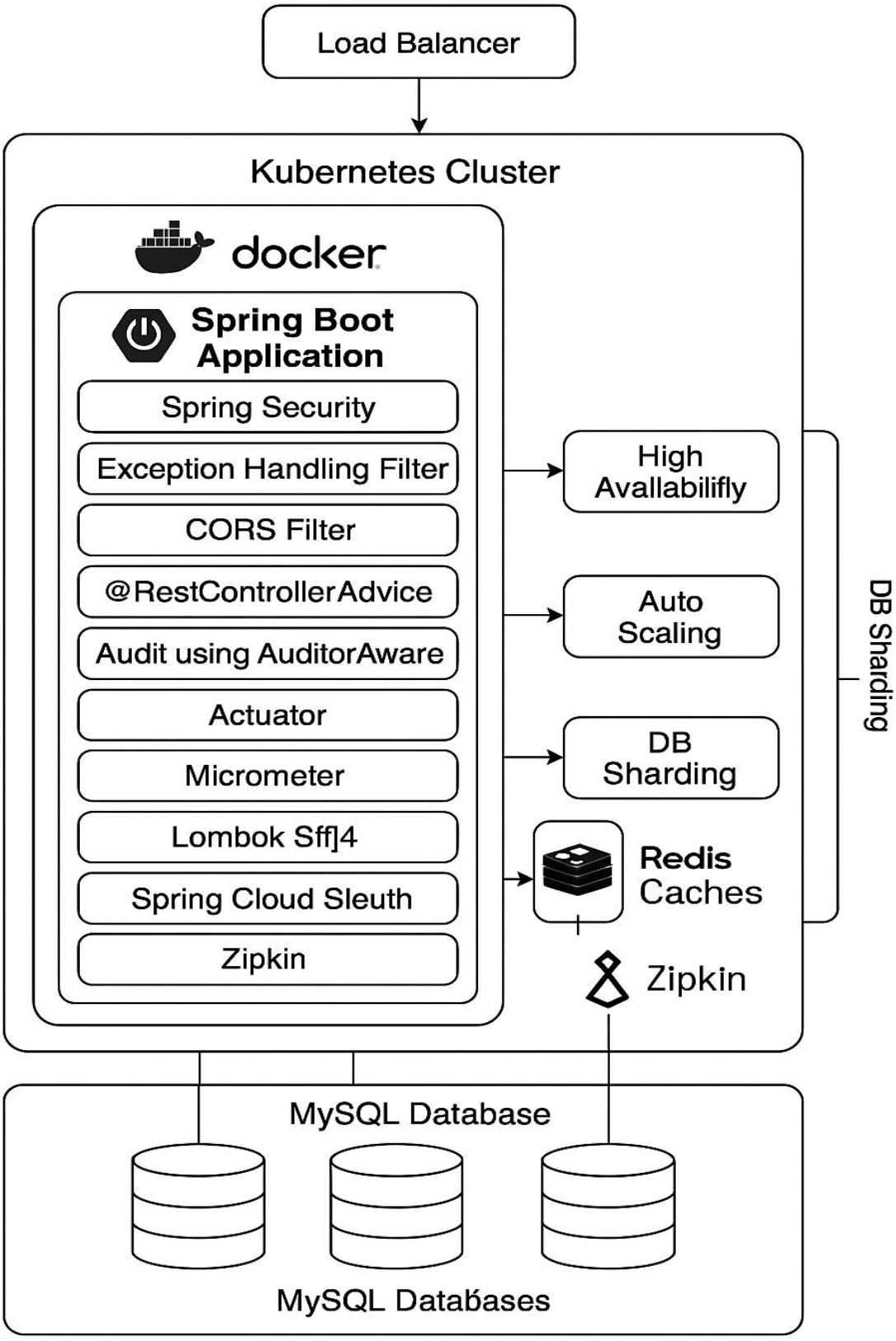
## a. Access Controls – Application Security

To have an access control in place, we should have application security to have Authentication and Authorization of resources. This is being implemented using **Spring Security. This is explained in detail under technologies used section.**

## 3. Logical Architecture



## 4. System Design



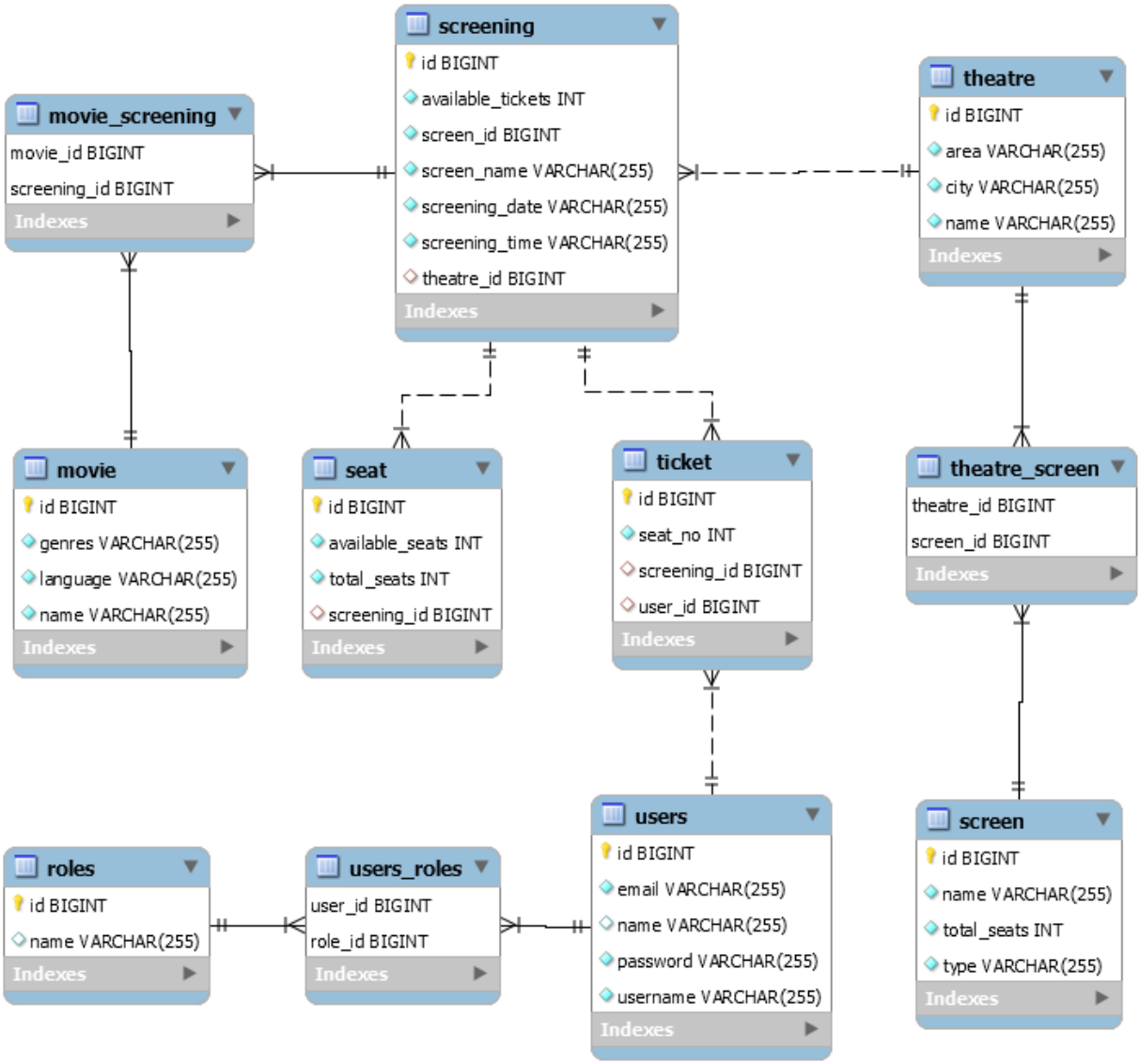
## 5. ERD

Primarily, User, Role Entities are required to capture the User Details and apply Security Mechanisms.

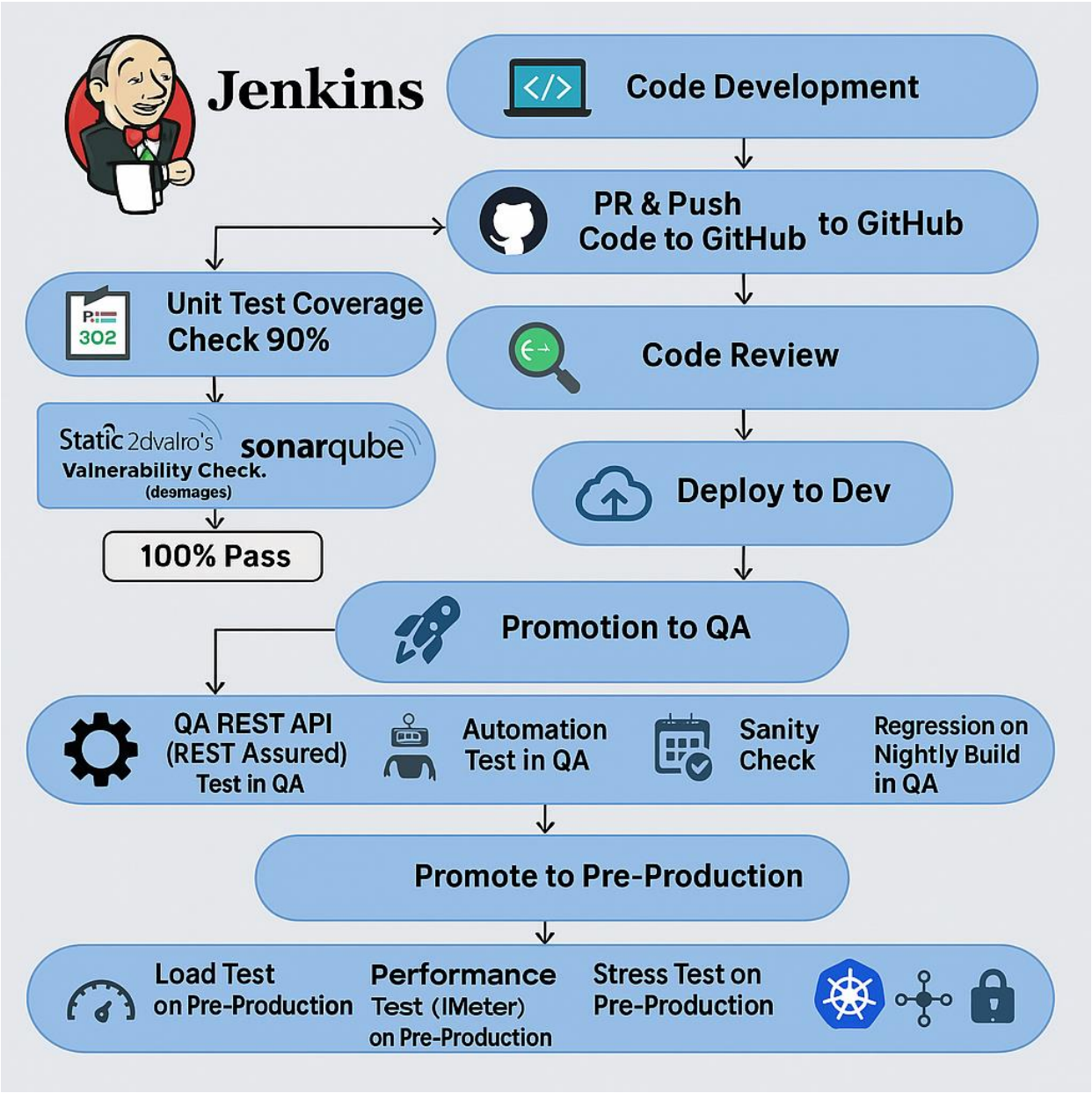
Theatre, Screen and Seat Objects are required capture the Partner/ Provider/ Theatre Owners Input.

There should be interface for End User/ Consumer to consume the services, those interface entity is Ticket Booking.

Whenever new movie release that needs to be deployed or screened for that Movie and Screen Objects are required/



## 6. CI & CD (Integration and Deployment Diagram)




**Code Development**

- Developers write code.
- Create Pull Request (PR) to GitHub.

**PR Checks**

- **Unit Test Execution**
    - Framework: JUnit/TestNG
    - **Coverage Goal:** ≥ 90%
- **Code Review**

- Manual review via GitHub or Bitbucket

**Static Code Analysis & Security Checks**

- Tool: **SonarQube**
- Check: **100% pass** (quality gate, no vulnerabilities/blockers)

**Build & Deploy to Dev**

- Build code using Maven/Gradle
- Deploy to **Dev environment**

**Promotion to QA**

- After smoke/sanity check in Dev

**QA Testing**

- **API Testing** – REST Assured
- **Automation UI Tests** – Selenium/Cypress
- **Sanity Check** – Quick validation
- **Regression Testing (Nightly)** – Full test suite
- **Goal:** 100% pass rate

**Promotion to Pre-Prod**

- Only if QA tests are successful

**Pre-Production Testing**

- **Load Testing** – JMeter
- **Performance Testing**
- **Stress Testing**

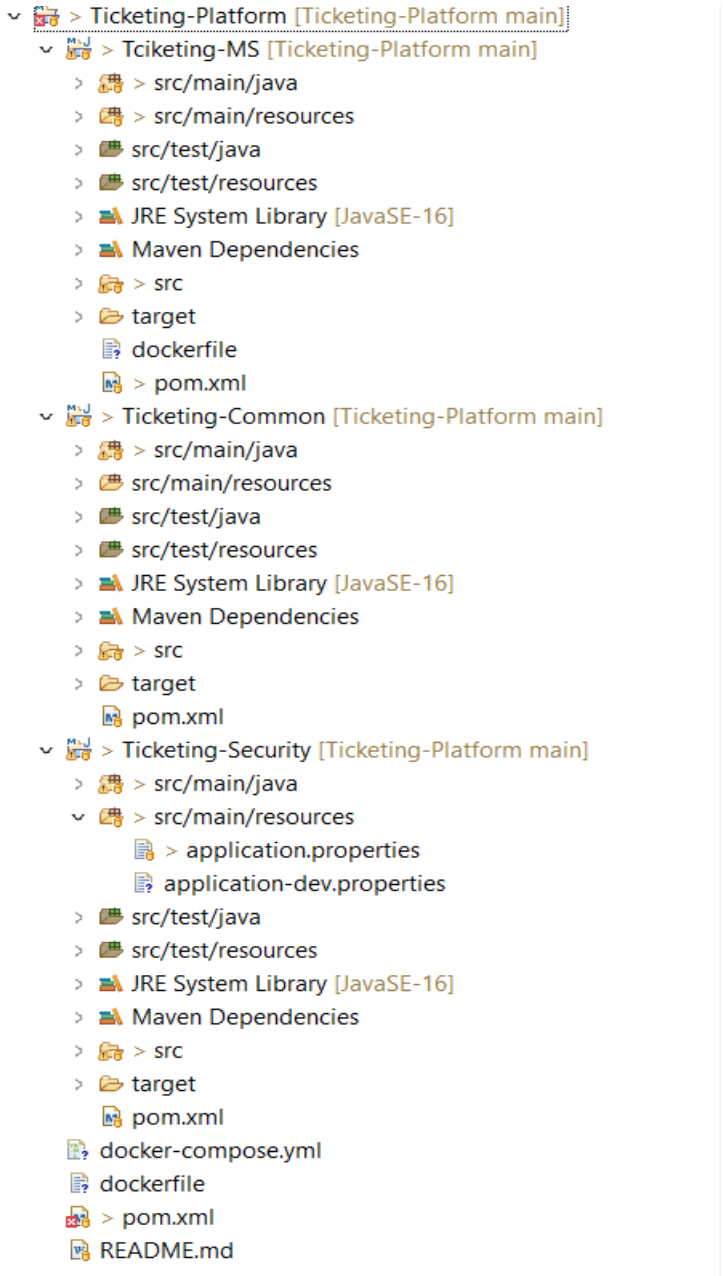
**Production Deployment**

- Deployment using **Kubernetes**
- Use of **Load Balancer**
- **Secure Networking** – TLS, firewalls, access control

# 7. Project Structure

Its Multi maven module projects, Modules are

a. **Common** Module which has Exception Handling Mechanism and Audit Configurations
b. **Security** Which has application Security explained in detail in Technologies Used Section
c. **MS – Microservice** All the transaction APIs' written here



# 8. Technologies Used

- Programming Language
  **Java** - To build highly scalable SAAS product, Java is a best choice Java + Spring Boot apps scale well in a containerized setup using Docker & K8s.

- Framework

  **Spring and Spring Boot** – Spring provides vast number of libraries to avoid spending write boiler plate code, Spring boot is very much suitable for Micro Services. Spring web module helps to build rest services seamlessly

- Back End Integration

  **JPA** is a java specification used to integrate DB, most of the DB operations can be done with less query writing effort. Since JPA is an ORM we can avoid wring queries. Query with value injection can be avoided

- Database

  **MySQL**, RDBMS is recommended for ticketing apps, since the app required structured data and relationship between entities. And Transactions (Data Integrity) and Concurrency Control can be also achieved

- Application Security
  **Spring Security** customizable framework for authentication, authorization, and protection against common attacks like CSRF, session fixation, clickjacking, etc.

It's used to generate **JWT** by username and password, once the user login the token can be generated once, that token can be used for every stateless transaction, it will help to secure REST API. JWT is generated with encryption, salting mechanism, HMAC key to prevent hack. Password Encoder used to mask the password while storing in DB.

**RBAC** can be done by applying @PreAuthorize("hasRole('ADMIN')"), it can be applied for whole Apis exposed from the class or at method level

**CSRF** (Cross-Site Request Forgery) disabled in security config. Prevents **malicious sites** from making requests **on behalf of an authenticated user**.

**CORS** (Cross-Origin Resource Sharing) Filter applied to filter the request. Controls which **origins (domains)** are allowed to make requests to your server.

**SSRF** To avoid SSRF attack implemented validating URL to validate the third-party API Which is Currency exchange provider API, Is that API secured, loop back

- **Transaction**
  Spring provided **@Transactional** can be used to rollback in case of when multiple entities persistence required sequentially. if everything inside succeeds, the transaction is committed. If something fails or throws an exception, the transaction is rolled back.

- **Auditing**

  Automatic Tracking of Entity Changes - No need to manually set created At, updated At, created By, etc. every time — JPA Auditing handles it *automagically*. With help of SecurityContextHolder + JWT, and we get full user audit trails for free. Tracking timestamps and

users helps detect: Unauthorized changes, Out-of-order updates, Stale data during conflict resolution

- **Exception Handling**
  it's done globally by two ways by utilizing spring provided annotation @RestControllerAdvice and using filter, we can avoid boilerplate code by not doing at method level.
  **Global Exception Handling**: Handle all exceptions in one place, reducing redundancy and complexity.
  **Customize Response Format:** Return a consistent error response structure across all controllers.
  **Cleaner Controllers:** Avoid cluttering the controller methods with try-catch blocks for common exceptions.
  **Enhanced Readability**: Global exception handling improves the clarity and organization of error handling logic.
  **Improved Maintenance**: Modify error response logic in one place (in @RestControllerAdvice) rather than in every individual controller.

  @RestControllerAdvice **does Controller-Level Errors**: mainly concerned with exceptions that happen **within controllers**
  Filter helps **General Error Logging**- want to log all errors globally, even those that happen outside the controller layer, a filter is more suitable.

- **Logging**
  Lombok @Slf4j used for logging to avoid boiler plate code of getting logger instance from logger factory

- **Monitoring**

  **Spring Boot Actuator** Exposes endpoints to check application health, metrics, environment, thread dumps, and more.

  **Micrometer** Integrated with Spring Boot Actuator. It acts as a facade for various monitoring systems (Prometheus, Datadog, New Relic, etc.). Use to gather detailed metrics (CPU, memory, GC, etc.).

  **Prometheus + Grafana - Prometheus** scrapes metrics exposed by Spring Boot. **Grafana** visualizes them.

**Distributed Tracing & Logging -Spring Cloud Sleuth + Zipkin**

Adds tracing IDs to logs. Works with Zipkin for distributed tracing visualization.

- Documentation **Swagger** (now under the **Open API** spec umbrella) auto-generates API docs and a UI from code.
- **Cloud** – this can be bundled as a **docker** image it can be pushed into any docker registry and deployed in any Kubernetes cluster.
- **Cache – Redis**

  Redis used for Caching

# 9. Design Principles, Patterns and Other Standards Used

**Spring, Spring Boot and JPA used. By Utilizing all this below design principles and Design Patterns Used**

**Design Principles:**

1. **Separation of Concerns (SoC)**

Spring Boot + JPA encourages breaking your application into layers:

- Controller Layer (web/API logic)
- Service Layer (business logic)
- Repository Layer (data access logic)
- Entity Layer (data model)

  This helps avoid mixing concerns and makes the code easier to test and maintain.

2. **Inversion of Control (IoC)**

   Spring's IoC container handles object creation and wiring:

Developer focus on what needs to be done, not how objects are created.

3. **Domain-Driven Design (DDD) Principles**

Using JPA entities aligns well with DDD: @Entity classes represent domain objects.

4. **Convention Over Configuration**

   Spring Boot minimizes boilerplate:

- Auto-configuration of Data Source
- Auto scan of @Entity and @Repository
- Default REST and actuator paths

5. **Single Responsibility Principle (SRP)**

Each class or component typically has one reason to change:

1. Controller → handles HTTP requests
2. Service → handles business rules
3. Repository → handles DB queries

Easier to test, debug, and refactor.

6. **DRY (Don't Repeat Yourself)**

- Spring Data JPA reduces boilerplate with auto-generated queries.
- Spring Boot uses properties or YAML for externalized config.
- Common logic is encapsulated in services or utilities.

**Design Patterns:**

1. **Dependency Injection (DI)**
   Spring injects dependencies automatically, making testing and maintenance easier.

2. **Repository Pattern**

JPA implements the repository pattern under the hood: This keeps data access logic out of your service or controller.

3. **Builder Pattern**

Used in: Creating complex objects like DTOs or Entities
Purpose: Simplifies object creation, especially for immutable classes.

4. **Factory Pattern**

Used in: Bean creation, dynamic repository or service creation
Purpose: Encapsulates object creation logic.

Spring beans are often created via @Bean or auto-scanned

5. **Template Method Pattern**

Used in: JpaRepository
Purpose: Skeleton of an algorithm in a method, letting subclasses fill in specifics. JpaRepository provides generic CRUD, and you define specifics via method names or overrides.

6. **DTO (Data Transfer Object) Pattern**

Used in: Controller → Service → View layers
Purpose: Separates domain models from what's exposed externally.

7. **Adapter Pattern**

Used in: Mapping between domain models and persistence models
Purpose: Converts interface of one class to another.

**Standards**

1. **Modularity & Scalability**

Using @Component, @Service, and @Repository annotations modularizes your app. Each module can be swapped, tested, or scaled independently.

2. **Transaction Management**

**@Transactional** Ensures data consistency without manually managing transactions.

# 10. Project Release Plan and Estimation

**Project Phases & Tasks**

**Phase 1: Requirements & Architecture Design**

- **Define Project Scope & Requirements**
  - Gather functional and non-functional requirements (1-2 weeks).

- Identify key features: authentication, user roles, API services, caching, auditing, etc.
- Estimated Effort: 40 hours

- **System Architecture Design**
  - Design high-level architecture (Spring Boot, JPA, MySQL, Redis, Spring Security).
  - Define the database schema and relationships.
  - Design security and exception handling structure.
  - Plan for Redis sharding and caching strategies.
  - Estimated Effort: 40 hours

**Phase 2: Initial Setup & Framework Configuration**

- **Spring Boot Setup**
  - Set up Spring Boot project with Spring JPA, Spring Security, and MySQL.
  - Configure application properties (e.g., MySQL connection, Redis connection).
  - Estimated Effort: 16 hours

- **Database Design & JPA Setup**
  - Implement JPA entities, relationships, and repositories.
  - Define auditing and timestamp management.
  - Implement auditing with AuditorAware.
  - Estimated Effort: 24 hours
- **Redis Cache Integration**
  - Set up Redis caching and sharding mechanism.
  - Integrate Redis with Spring Boot for cacheable services.

- Estimated Effort: 24 hours

**Phase 3: Implement Core Features**

- **Spring Security Configuration**
  - Implement JWT-based authentication and authorization.
  - Set up roles and permission-based access control.
  - Implement CORS configuration for cross-origin requests.
  - Estimated Effort: 32 hours
- **Exception Handling**
  - Implement global exception handling with @RestControllerAdvice.
  - Customize error messages and response structures.
  - Estimated Effort: 16 hours
- **Auditing Setup**
  - Implement AuditorAware for automatic user tracking.
  - Integrate auditing into entity classes.
  - Estimated Effort: 16 hours

**Phase 4: Develop APIs & Core Business Logic**

- **API Development**
  - Design and develop RESTful APIs (CRUD operations, authentication, etc.).
  - Implement validation, exception handling, and error responses.
  - Estimated Effort: 80 hours
- **Service Layer Logic**
  - Implement core business logic and service methods.
  - Integrate caching, auditing, and security into services.
  - Estimated Effort: 40 hours

**Phase 5: Testing**

- **Unit Testing**

  - Write unit tests for service and repository layers.
  - Test with JUnit and Mockito.
  - Estimated Effort: 32 hours

- **API Testing**
  - Write integration tests for REST APIs (Spring Test, MockMvc).
  - Test endpoints for expected behaviors, edge cases.
  - Estimated Effort: 40 hours

- **QA Automation Testing**
  - Implement automated testing with Postman or similar tools.
  - Automate functional API tests.
  - Estimated Effort: 40 hours

- **Load, Performance, and Stress Testing**
  - Set up load testing with tools like JMeter or Gatling.
  - Run performance tests to identify bottlenecks.
  - Conduct stress testing to assess scalability.
  - Estimated Effort: 40 hours

**Phase 6: Deployment & Monitoring**

- **Deployment Setup**
  - Set up deployment pipelines (CI/CD using Jenkins, GitHub Actions, or similar).
  - Configure deployment environments (dev, staging, prod).
  - Estimated Effort: 24 hours

- **Monitoring & Logging Setup**
  - Integrate with monitoring tools (Prometheus, Grafana, ELK stack).
  - Set up logging for error tracking and system performance.
  - Estimated Effort: 16 hours

**Phase 7: Post-Deployment & Maintenance**

- **Post-Deployment Monitoring & Bug Fixing**
    - Monitor logs and system health post-deployment.
    - Address any critical issues or bugs.
    - Estimated Effort: 24 hours

**Total Estimated Effort:**

- Development & Testing: ~400 hours (10 weeks)
- Post-Deployment & Maintenance: ~40 hours (1 week)

**Time & Effort Breakdown per Area:**

1. Backend Development (Spring Boot, JPA, Security, Redis, Auditing, etc.): 60% of the total effort (~240 hours)
2. Testing & QA (Unit tests, integration tests, load/performance testing): 20% of the total effort (~80 hours)
3. Deployment & Monitoring: 10% of the total effort (~40 hours)
4. Project Management, Documentation, and Communication: 10% of the total effort (~40 hours)

**Risks & Considerations**

- Redis Sharding Complexity: Ensure the sharding logic is well-architected and scalable.
- Security: Pay close attention to proper configuration of JWT, role-based access control, and secure communication (e.g., SSL/TLS).
- Performance: Redis caching and load balancing need to be tested under stress to ensure system scalability.
- Integration with MySQL: MySQL might be a limiting factor in terms of performance if the system scales heavily. Plan for database optimizations and partitioning strategies.