**Software Architecture Document**

**TICKETWAVE**

|  |
| --- |
| **Date : 18th September 2024** |
| **Version : 1.0** |
| **State : On-going** |
| **Author : Anh Huynh** |

#### Version History

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Version** | **Date** | **Author(s)** | **Changes** | **State** |
| 0.0 | 18/9/2024 | Anh Huynh | Making SAD | Draft |
| 1.0 | 3/10/2024 | Anh Huynh | First initial version | On-going |
| 1.1 | 9/10/2024 | Anh Huynh | Adjust C1 and C3 based on teacher’s feedback | On-going |

Contents

[**1. Introduction 4**](#_Toc42673512)

[**2. System Context (C1) 5**](#_Toc42673519)

[**3. Containers and tech choices (C2) 6**](#_Toc42673525)

[**4. Components (C3) 8**](#_Toc42673525)

# Introduction

Ticketwave is a ticket management system designed to provide users with an efficient way to browse events, book tickets, and manage their bookings. The platform supports multiple user roles such as regular users, sales managers, and administrators. The backend is developed using Spring Boot, the frontend uses React, and MySQL serves as the database. This architecture adheres to SOLID principles to ensure maintainability, scalability, and flexibility for future enhancements.

# System Context Diagram (C1)

## A diagram of a sales manager Description automatically generatedDiagram

## Key Actors and External Systems:

* **User (Person):** Interacts with the system to browse events, book tickets, and manage bookings.
* **Sales Manager (Person):** Creates and manages event listings, and tracks ticket sales.
* **Admin (Person):** Manages users, roles, and event approvals.
* **Email/Notification System (External System):** Sends booking confirmations, updates, and reminders to users.

## Interactions:

* **Users** interact with the system through a frontend interface to browse events, book tickets, and manage their profiles.
* **Sales Managers** interact with the system to manage events and monitor sales.
* **Admins** oversee user management and event approvals.
* **The Email System** receives booking confirmations and notifications related to bookings, and event updates, from the system and sends them to users.

# Containers and tech choices (C2)

## A diagram of a computer Description automatically generatedDiagram

## Containers

* **Frontend (React Application)**:
* Provides the main user interface for users, sales managers, and administrators to interact with the system, and communicates with the backend via RESTful APIs.
* **React** was selected for the frontend because of its **component-based architecture**, which encourages reusability and separation of concerns:
* **KISS (Keep It Simple, Stupid)** is applied as React helps keep the UI code simple and reusable, allowing quick updates to individual components without affecting the whole system.
* **Backend (Spring Boot Application)**:
* Processes API requests from the frontend, manages business logic (such as ticket booking and event management), ensures role-based access control, and communicates with the database for CRUD operations and the email system for notifications.
* **Spring Boot** was chosen because it provides a **modular, layered architecture** that aligns with **SOLID principles**:
* It supports **dependency injection**, making it easy to follow the **Single Responsibility Principle (SRP)** by breaking the application into manageable layers (controllers, services, repositories).
* Spring Boot simplifies **REST API** creation, testing, and security through its built-in tools.
* **MySQL Database**:
* Manages persistent data storage for users, events, and tickets.
* **MySQL** was chosen for the database because of its reliability and strong support for **transactional integrity**:
* MySQL integrates seamlessly with **Spring Boot JPA**, which helps implement persistence layers while reducing boilerplate code, adhering to the **DRY (Don't Repeat Yourself)** principle.
* **Email/Notification System**
* Sends booking confirmations and event-related updates to users via email.

# Components (C3)

## A diagram of a company Description automatically generatedDiagram

## Components

* **Controller Layer**: Contains the REST controllers that handle HTTP requests from the frontend. Controllers validate the incoming data and pass it to the appropriate service for processing.
  + Example: *TicketController* handles ticket-specific requests like ticket booking, viewing, and cancellations*.*
* **Service Layer**: Implements the business logic for different entities (User, Event, Ticket) and interacts with repositories to manage data.
  + Example: *TicketServiceImpl* handles business logic for ticket bookings, cancellations, and viewing.
* **Repository Layer**: Manages communication with the database, implements CRUD operations and interacts with the database to retrieve or store entity data.
  + Example: *TicketRepositoryImpl* handles the persistence and retrieval of ticket data from the database, ensuring all ticket-related actions are stored or retrieved correctly.
* **Mapper Layer**: Transform data between DTOs (Data Transfer Objects) and Domain Entities. This ensures that data is appropriately structured when moving between layers, especially between the service and controller layers.
  + Example: *TicketMapper* transforms ticket-related DTOs into ticket domain entities and vice versa.
* **Domain Layer**: Represents the core business entities in the system, such as User, Event, and Ticket. These entities are managed by the service and repository layers.
* **Configuration Layer**: Manages application-wide settings, such as security (authentication and authorization), CORS configurations, database initialization, and global exception handling. It ensures that the application operates securely and that system-wide configurations are properly applied.
  + Example: *WebConfig* manages CORS settings to ensure that the frontend can communicate with the backend.

## Design Principles

* **SOLID Principles:**
  + **Single Responsibility Principle (SRP)**: Each class/component has a well-defined responsibility. For instance, TicketController only handles API requests related to ticket management, while TicketServiceImpl handles the business logic for ticket-related operations.
  + **Open/Closed Principle (OCP)**: The system is open for extension but closed for modification. For example, new controllers or services can be added without altering existing functionality.
  + **Interface Segregation Principle (ISP)**: Interfaces such as TicketService and TicketRepository ensure that components only expose what is necessary for their responsibilities, adhering to the principle of not forcing unnecessary dependencies.
  + **Dependency Inversion Principle (DIP)**: The controllers depend on abstractions (interfaces) instead of concrete implementations. For example, TicketController interacts with the TicketService interface, allowing for flexibility and easy testing.
* **KISS (Keep It Simple, Stupid):**
  + The architecture avoids unnecessary complexity. Each layer has a clearly defined responsibility, keeping the design simple and focused.
* **DRY (Don’t Repeat Yourself):**

Business logic is centralized in the service layer to avoid duplication. Mappers are used to standardize conversions between DTOs and domain models.