Project-High Level Design
on

Turf Booking System(a sport service app)

Course Name: DevOps Foundation

Institution Name:Medicaps University – Datagami Skill Based Course

Student Name(s) & Enrolment Number(s):

Sr no	Student Name	Enrolment Number
1	Aparna Sharma	EN24CA5030027
2	Harsh Gupta	EN24CA5030057
3	Mimansa Kashyap	EN22IT301054
4	Aditya Bairagi	EN24CA5030009
5	Sandhya Patidar	EN24CA5IT030150
6	Shubhanjati Sharma	EN24CA5030161

Group Name: 12D12

Project Number: DO-50

Industry Mentor Name: Vaibhav sir

University Mentor Name:Akshaya Saxena sir

Academic Year:2024-2026

Table of Contents

1.Introduction

The Turf Booking System is a full-stack web application designed to simplify the process of discovering, scheduling, and managing sports turf reservations. The system provides an intuitive web interface where users can view available turfs, select preferred time slots, and create bookings efficiently. It is built using a modern technology stack with a React frontend for user interaction and a FastAPI backend for handling business logic, API communication, and database operations. The application follows a structured client-server architecture that separates presentation, processing, and data storage layers, ensuring maintainability and scalability. The primary objective of this system is to automate manual booking processes, reduce scheduling conflicts, and provide a reliable platform that can be extended with advanced features such as user authentication, online payments, notifications, and cloud deployment in future versions.

1.1 Scope of the Document

This document describes the design and implementation of the Turf Booking System, a full-stack web application developed to manage turf availability and online slot bookings. It explains the system architecture, major components, data handling methods, and overall workflow. The scope includes both frontend and backend design, database structure, API interactions, and future scalability considerations.

1.2 Intended Audience

The intended audience of this document includes developers, project reviewers, instructors, and stakeholders who need to understand the system design and functionality. It is useful for technical teams responsible for development and maintenance, as well as non-technical readers who require a high-level overview of how the system operates.

1.3 System Overview

The Turf Booking System enables users to view available turfs and reserve time slots through a web interface. The application uses a React-based frontend for user interaction and a FastAPI backend for business logic and data processing. The system ensures efficient

booking management, real-time availability checking, and structured data storage using a relational database.

2. System Design

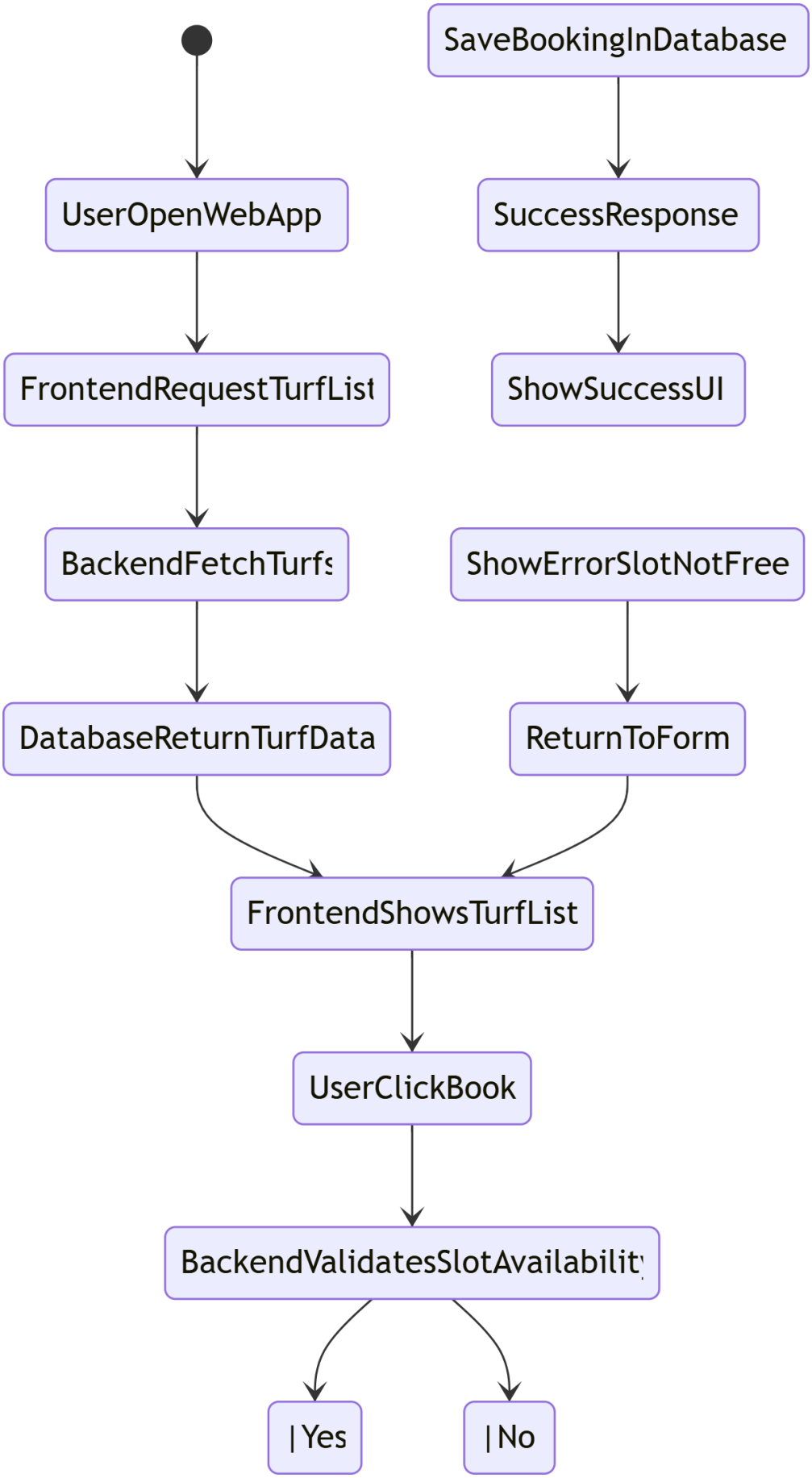
The Turf Booking System is designed using a layered client-server architecture that separates the user interface, application logic, and data storage to ensure scalability, maintainability, and performance. The frontend, developed using modern web technologies, handles user interactions such as viewing turfs, selecting time slots, and submitting booking requests. These requests are sent to the backend through RESTful APIs, where the FastAPI server processes business logic including slot validation, booking creation, and error handling. The system is divided into functional components such as turf management, booking management, and future modules like authentication and payment integration. A relational database stores turf information, booking records, and user details, enabling efficient data retrieval and updates. The design focuses on modularity, security, and flexibility so that additional features, cloud deployment, and scaling strategies can be implemented without major architectural changes.

2.1 Application Design

The application follows a client-server architecture where the frontend communicates with the backend via REST APIs. The frontend handles user interactions, while the backend processes booking logic, validation, and database operations. The design emphasizes modularity, maintainability, and scalability.

2.2 Process Flow

The process begins when a user accesses the application and views the list of available turfs. When a booking request is submitted, the backend validates slot availability, stores the booking in the database, and returns a confirmation response. This structured flow ensures smooth user experience and prevents booking conflicts.

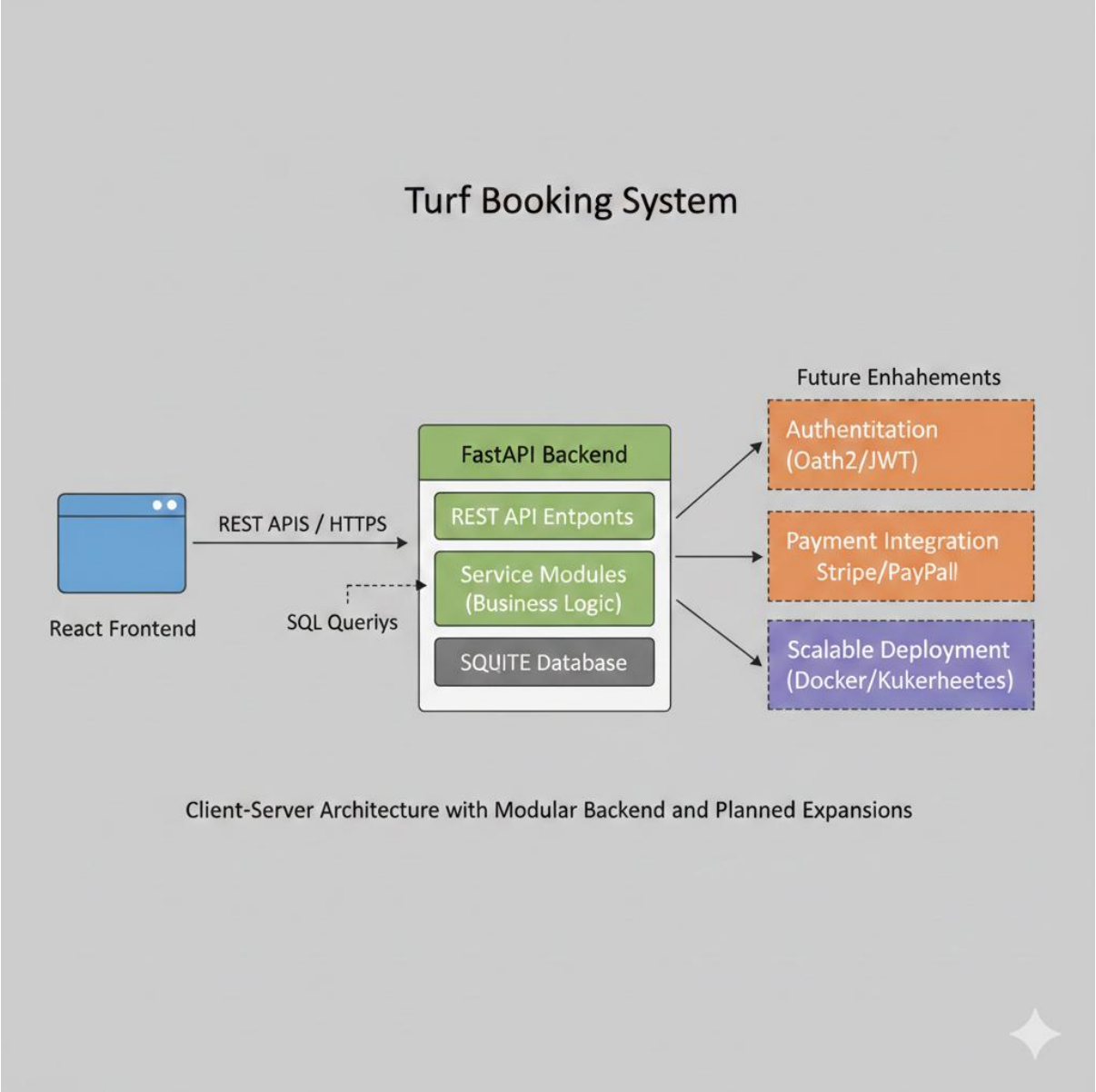

SaveBookingInDatabase
UserOpenWebApp
SuccessResponse
FrontendRequestTurfList
ShowSuccessUI
BackendFetchTurfs
ShowErrorSlotNotFree
DatabaseReturnTurfData
ReturnToForm
FrontendShowsTurfList
UserClickBook
BackendValidatesSlotAvailabilit
| Yes
| No

2.3 Information Flow

Information flows between the user interface, backend services, and database. User inputs such as booking date and time are sent to the backend, processed by service modules, and stored in the database. The backend then sends updated availability information back to the frontend.

2.4 Components Design

The system consists of key components including the user interface, API server, booking service, turf management service, and database layer. Each component has a specific responsibility, enabling separation of concerns and easier maintenance.



Client-Server Architecture with Modular Backend and Planned Expansions

2.5 Key Design Considerations

Important design considerations include data validation, prevention of double booking, security, performance, and scalability. The system is designed to support future features such as authentication, payment integration, caching, and deployment in cloud environments.

2.6 API Catalogue

The API catalogue defines endpoints used for system communication, such as retrieving turf lists, creating bookings, and viewing booking details. These APIs follow REST principles and ensure consistent interaction between frontend and backend.

3. Data Design

The data model of the Turf Booking System defines how information is structured, stored, and related within the database. The system primarily consists of key entities such as Turfs, Bookings, and Users. Each turf entity contains details like turf name, location, price, and available time slots, while the booking entity stores information including user name, booking date, start time, end time, and the associated turf ID. A relationship is established where one turf can have multiple bookings, ensuring proper tracking of slot availability and preventing scheduling conflicts. The user entity, planned for future implementation, will manage authentication and booking history. The data model is designed using relational principles to maintain data integrity, support efficient queries, and allow future expansion such as payment records, notifications, and analytics without significant structural changes.

3.1 Data Model

The data model defines entities such as Turfs, Bookings, and Users. Relationships are structured so that each turf can have multiple bookings, and each booking contains user details, time slots, and dates.

3.2 Data Access Mechanism

Data is accessed through backend services using database queries and ORM techniques. This approach abstracts database operations, improves security, and simplifies data handling within the application.

3.3 Data Retention Policies

The system stores booking data for operational and reporting purposes. Retention policies ensure that historical booking information is preserved while allowing future implementation of archiving or cleanup strategies for old records.

3.4 Data Migration

Data migration refers to transferring database structures or records when upgrading the system or moving to a different database environment. The design allows smooth migration through schema versioning and backup strategies.

4. Interfaces

The Dockerized Sports Rust Service provides clearly defined interfaces that enable communication between users, the application, and supporting services. These interfaces ensure smooth interaction between different system components while maintaining security and portability across environments.

4.1 API Interface

The application exposes a RESTful API built using Rust, which allows clients such as web applications, mobile apps, and API testing tools to interact with the service. Communication takes place over HTTP or HTTPS protocols, and all data is transferred in JSON format to ensure compatibility with modern systems. The API typically runs on port 8080, which can be configured using environment variables to support different deployment environments. The API follows REST principles, meaning each request is independent and contains all necessary information. Since the service is containerized, the API behaves consistently across development, testing, and production setups. This consistency eliminates environment-related errors and ensures that all team members experience the same system behavior.

4.2 Database Interface

The service connects to a relational database managed inside a Docker container. The database used in this project is PostgreSQL, which provides reliable and structured data improving security and flexibility. Docker Compose ensures that the database container starts before the application container, maintaining proper dependency order. This interface guarantees secure, stable, and efficient data management within the containerized environment.

4.3 Container and Deployment Interface

Containerization and orchestration are handled using Docker and Docker Compose. Docker packages the Rust application along with its dependencies into a lightweight container, ensuring portability. Docker Compose manages multiple services, including the application and database containers, by defining them in a single configuration file. It also creates a shared network and manages volumes for persistent storage. This interface ensures that the entire system can be started with a single command and behaves identically on every machine. It simplifies deployment, enhances team collaboration, and supports scalable infrastructure.

5. State and Session Management

The Dockerized Sports Rust Service is designed using a stateless architecture, which is considered a best practice for container-based applications. This design ensures flexibility, scalability, and fault tolerance.

5.1 Application State

The application does not store any persistent state within the container itself. All business data, including sports records and related information, is stored in the PostgreSQL database. Because containers are temporary by nature, this approach ensures that restarting or recreating containers does not result in data loss. Docker volumes are used to persist database data beyond the lifecycle of individual containers. This separation of application logic and data storage ensures reliability and supports easy scaling of the service.

5.2 Session Management

If authentication is implemented, the system uses token-based authentication mechanisms such as JWT. In this approach, session information is stored on the client side rather than on the server. Each request contains the authentication token, and the server validates it independently. This eliminates the need for server-side session storage and supports horizontal scaling. Since the service is stateless, multiple instances of the application can run simultaneously without synchronization issues.

6. Caching

Caching mechanisms are implemented to improve efficiency and reduce unnecessary computation and database load within the Dockerized Sports Rust Service.

6.1 Docker Build Caching

The project uses multi-stage Dockerfiles to optimize build performance and reduce image size. During the build process, dependencies defined in Cargo.toml and Cargo.lock are copied and compiled first. Docker caches these layers so that they are not rebuilt unless dependency files change. This significantly speeds up rebuild times during development. The final runtime image excludes build tools and compilers, resulting in a smaller and more production image.

6.2 Application-Level Caching

At the application level, frequently accessed sports data can be temporarily stored in memory to reduce repeated database queries. This improves response time and reduces load on the PostgreSQL container. Although optional, integrating an external caching solution in the

future could further enhance performance for read-heavy operations. These caching strategies contribute to faster execution and better resource utilization.

7. Non-Functional Requirements

Non-functional requirements define the quality standards that the Dockerized Sports Rust Service must meet, including security, performance, scalability, and reliability.

7.1 Security Aspects

Security is implemented at multiple levels within the system. Multi-stage Docker builds reduce the attack surface by excluding unnecessary build tools from the runtime image. Minimal base images are selected to decrease potential vulnerabilities. Containers are configured to run as non-root users to prevent privilege escalation. Sensitive information such as database credentials is managed through environment variables rather than hardcoded values. Network isolation ensures that the PostgreSQL database is not publicly accessible. Additionally, input validation and proper error handling are implemented in Rust to prevent injection attacks and information leakage. These measures collectively strengthen the security posture of the application.

7.2 Performance Aspects

Performance optimization is achieved through efficient coding practices and container best practices. Rust provides high performance due to its memory safety and compiled nature. The application is built in release mode to ensure optimized binary output. Multi-stage Docker builds reduce image size, leading to faster startup times. Docker Compose allows defining CPU and memory limits to maintain stable performance under load. The stateless design supports horizontal scaling, enabling multiple container instances to handle increased traffic. Database indexing and efficient queries further enhance response times. Together, these strategies ensure consistent and reliable performance.