Detailed Design Document for SPORTS

Authors:

Prajwal Umesha

Poorna Bengaluru Shivaji Rao

Haifeng Huang

Wan Wang

Group: 2

This page intentionally left blank.

Document Revision History

Date	Version	Description	Author
11/25/2025	0.0	Initial draft for detailed design document	Poorna, Prajwal, Haifeng Huang, Wan Wang
TBD	TBD	TBD	TBD

Contents

1 Introduction

This section provides an overview of the document, addressing its purpose, audience, and a high-level view of the system's design goals. It also outlines the project's scope and includes references to supporting materials.

1.1 Purpose

The purpose of this document is to detail the design of the SPORTS system, which aims to streamline event management for small-scale sports leagues. This design document builds on the requirements specification, offering a more technical perspective that explains **how** the system's goals will be achieved. It includes architectural diagrams, detailed descriptions of each module, and implementation strategies.

The document is organized as follows:

- **Introduction:** Overview of the purpose, system, and design objectives.
- **System Architecture:** Technical design and structure of the system.
- **Module Design:** Detailed design of individual components.
- **Non-functional Considerations:** Addressing performance, security, and scalability.
- **References:** Supporting materials and documentation.

1.2 System Overview

The SPORTS system is designed for managing small-scale sports leagues, offering features like venue scheduling, team registration, match management, and real-time data analysis. It supports league managers, team managers, venue owners, and players, ensuring efficient coordination and event execution.

The target environment includes both web-based and mobile platforms to provide accessibility for diverse users. The system integrates external APIs (e.g., weather and payment gateways) and operates in a secure, responsive, and user-friendly manner.

1.3 Design Objectives

The primary design objectives for the SPORTS system include:

- **Core Functionality:** Ensuring smooth registration, scheduling, and data management for small-scale sports leagues.
- **Performance:** Supporting up to 500 concurrent users with a response time of under 2 seconds.
- **Usability:** Providing an intuitive user interface with real-time updates for schedules and standings.
- **Security:** Ensuring data integrity and compliance with regulations like GDPR through encryption and secure authentication.
- **Scalability:** Designing a modular architecture that can adapt to larger-scale deployments in the future.

The system prioritizes the most critical user workflows, such as managing leagues and scheduling matches, while leaving advanced analytics and integrations for future iterations. This focus ensures a robust, lightweight design that fulfills the immediate needs of local sports organizations.

1.4 References

Software Requirements Specification (SRS) for SPORTS, Version 2.1 (10/26/2024)

Unified Modeling Language (UML) Specification, Version 2.5.1

IEEE Std 830-1998: Guidelines for Software Requirements Specifications

1.5 Definitions, Acronyms, and Abbreviations

- **SPORTS:** System for Production of Recreational Team Scheduling, the software being designed and developed for small-scale sports league management.
- **User:** Any individual interacting with the SPORTS system, including league managers, team managers, players, venue owners, and city officials.
- **Module:** A distinct functional unit of the system, such as user registration, schedule management, or payment processing.
- **SRS:** Software Requirements Specification.
- **UML:** Unified Modeling Language, used for system modeling.
- **API:** Application Programming Interface, enabling integration with external services such as payment gateways and weather monitoring systems.
- **GDPR:** General Data Protection Regulation, a legal framework for data protection and privacy in the European Union.
- **DB:** Database, where system data such as users, teams, and schedules are stored.
- **GUI:** Graphical User Interface, the visual interface for user interaction with the system.
- **HTTPS:** Hypertext Transfer Protocol Secure, ensuring secure communication between the system and its users.

2 Design Overview

2.1 Introduction

- **Design Approach:** The SPORTS system adopts an object-oriented design for this application, with an emphasis on modularity, separation of concerns, and reusable components. The architecture follows a client-server model where the client interacts with a web application front-end, which communicates with backend services and a relational database.
- **Proposed Architecture:**
  - The system is designed using a client-server architecture. The client will interact with the web application and interact with the server where the business logic is written to manage venue scheduling, league match organization, and conflict resolution effectively.

    - This document outlines the design approach and architecture for deploying a cloud-based web application hosted on AWS infrastructure. The application leverages a Docker container hosted on Amazon Elastic Beanstalk, with a relational database managed through Amazon RDS (Relational Database Service), specifically using Amazon Aurora for the SQL database. The overall design focuses on scalability, availability, and integration of AWS services to support modern cloud-native application deployment practices.
- **Tools Used:**
    - **draw.io:** Used to create detailed class diagrams, UML diagrams, providing a clear visual representation of the system's design.
    - **Lucid chart-** Used to create detailed sequence diagrams, providing a clear visual of the systems interactions and timelines.

This design approach ensures the SPORTS system is well-structured, efficient, and capable of meeting user requirements while maintaining long-term maintainability.

2.2 Environment Overview

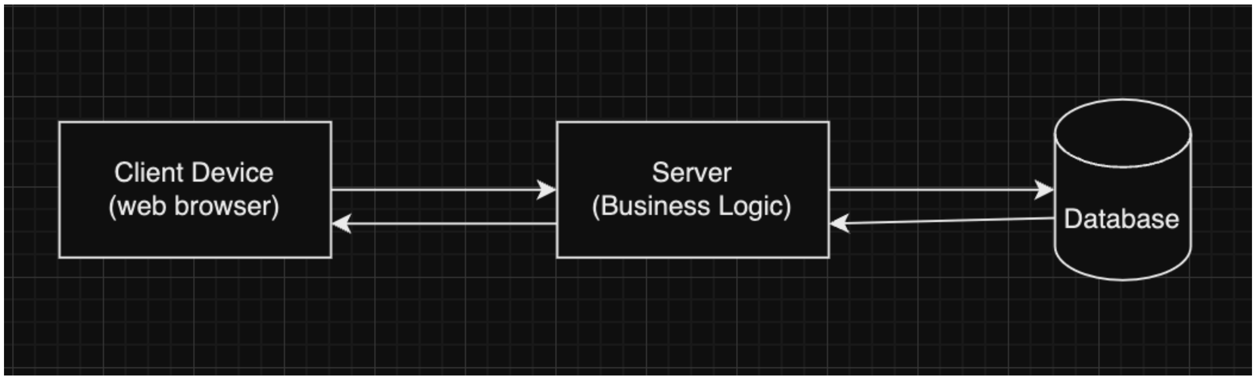
The SPORTS system is designed to function as a web-based application for efficient venue scheduling and league match organization. The environment in which the system operates includes client devices, a centralized server, and a database for managing user interactions and data.

- **System Scope:**
  The SPORTS system handles scheduling operations, conflict resolution, and schedule modification for leagues. It ensures smooth user interactions through a responsive client interface and efficient backend processing.
- **System Environment:**
    - **Client-Side:**
      End users, including league organizers and administrators, access the system via web browsers on devices connected to the internet. These devices form the interface for interacting with the system's features.
    - **Server-Side:**
      A centralized application server hosts the business logic of the system. It handles scheduling computations, conflict detection, and data storage operations. The server is scalable to meet varying user demands.
    - **Database:**
      A relational database is used to store critical data, including match schedules, venue details, user profiles, and system logs.
- **Deployment Location:**
    - The application will be deployed on Amazon Web Services (AWS) for reliability, scalability, and modern cloud-native support. Specifically:
    -
    - The application is hosted using a Docker container deployed on Amazon Elastic Beanstalk, which simplifies the deployment, scaling, and management of the web application.

    - A relational database is managed using Amazon RDS (Relational Database Service), specifically leveraging Amazon Aurora for its high performance and availability in SQL database operations.
    - AWS infrastructure ensures secure, scalable, and highly available services, supporting the SPORTS system's requirements for efficient scheduling, conflict resolution, and data handling.

This deployment strategy integrates AWS's scalable services, providing a robust environment for future enhancements and increased user demands.

- **Execution Flow:**
    - Users log in through client devices, entering their credentials on the user interface.
    - Requests are sent from the client to the server through a secure RESTful API.
    - The server processes these requests, performs computations or conflict checks, and interacts with the database for necessary updates or queries.
    - Responses, including schedule details or confirmation messages, are sent back to the client interface for display.
- **Environment Diagram:**
  The following diagrams outline the system's environment:
    - **Client-Server Interaction Diagram:** Shows the flow of requests and responses between client devices and the server.
    - **Database Connectivity Diagram:** Illustrates how the server queries and updates the relational database.



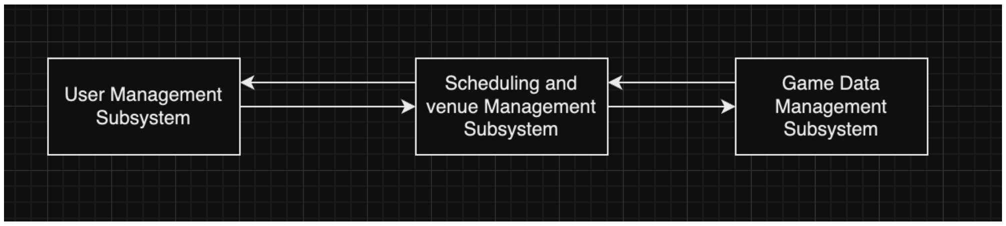
These components ensure that the SPORTS system operates efficiently and reliably in its intended environment.

2.3 System Architecture

The SPORTS system consists of three major components:

1. **User Management Subsystem**
2. **Scheduling and Venue Management Subsystem**
3. **Game Data Management Subsystem**

The interaction between these subsystems is multi-directional. The **User Management Subsystem** handles player, team, and league organizer data, which feeds into the **Scheduling and Venue Management Subsystem** for creating and managing schedules. The **Game Data Management Subsystem** receives updates from the scheduling system and tracks game results and statistics.


User Management Subsystem
Scheduling and venue Management Subsystem
Game Data Management Subsystem

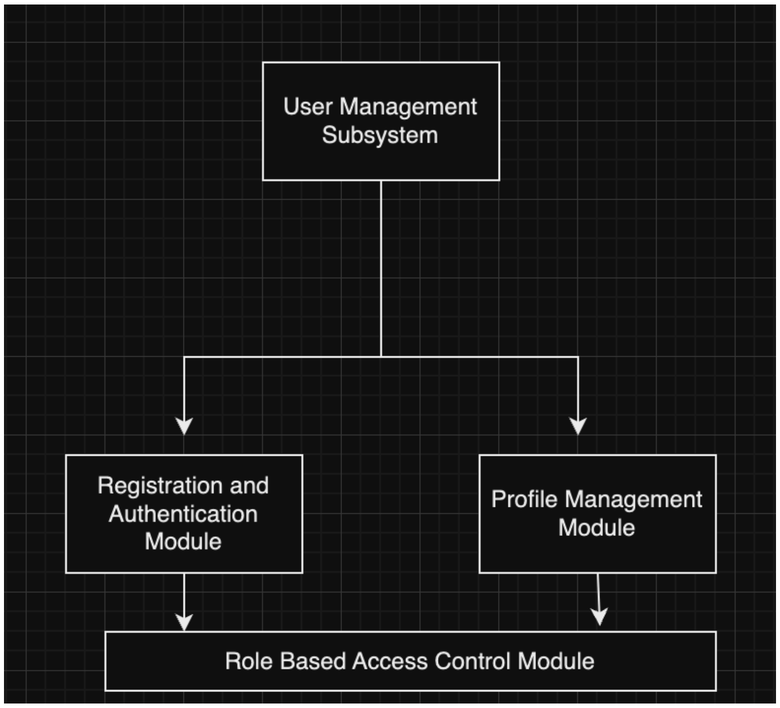
2.3.1 User Management Subsystem

The **User Management Subsystem** includes two primary components:

- **Registration and Authentication Module:** Handles user accounts, roles, and permissions for players, organizers, and referees.
- **Profile Management Module:** Maintains detailed player and team information.

Interactions:

- Provides user data to the **Scheduling and Venue Management Subsystem** for scheduling games.
- Interacts with the **Game Data Management Subsystem** to update player and team statistics.


User Management Subsystem
Registration and Authentication Module
Profile Management Module
Role Based Access Control Module

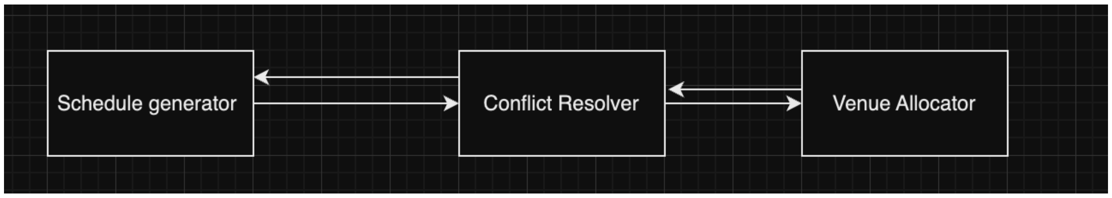
2.3.2 Scheduling and Venue Management Subsystem

This subsystem is central to the SPORTS system. It includes:

- **Schedule Generator:** Automatically generates match schedules based on league rules, team availability, and venue availability.
- **Conflict Resolver:** Detects and resolves scheduling conflicts.
- **Venue Allocator:** Assigns venues for games and updates their status.

Interactions:

- Receives player/team availability from the **User Management Subsystem.**
- Sends finalized schedules to the **Game Data Management Subsystem.**



2.3.3 Game Data Management Subsystem

This subsystem focuses on maintaining game-related data:

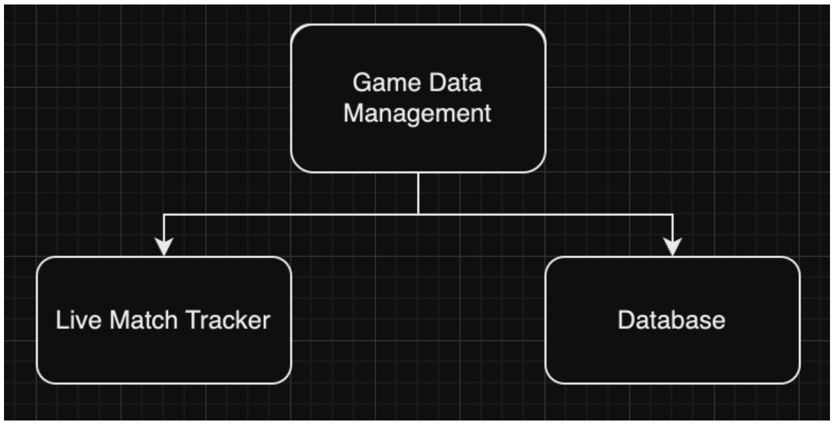
- **Live Match Tracker:** Tracks ongoing games for score updates and time management.
- **Statistics Manager:** Maintains game results, player performance metrics, and league standings.

Interactions:

- Pulls game schedules from the **Scheduling and Venue Management Subsystem.**
- Pushes updated statistics to the **User Management Subsystem** for team/player profiles.

Conventions and Notations

- Rectangles represent components/subsystems.
- Arrows show data flow between components, labeled to indicate data type.
- Double-sided arrows indicate bi-directional communication.



2.4 Constraints and Assumptions

2.4.1 Customer-Imposed Constraints

1. **Real-Time Conflict Detection:**
   Scheduling conflicts must be detected and resolved in real-time.
   Accommodation:
   - This is achieved using optimized scheduling algorithms implemented in the server-side logic.
   - The server processes scheduling requests within a 500-millisecond latency threshold to ensure responsiveness.
2. **Cross-Platform Accessibility:**
   The system must be accessible through all modern web browsers.
   Accommodation:
   - The client-side application is developed as a web-based application using HTML5, CSS3, and JavaScript for compatibility across devices.
3. **Secure User Authentication:**
   User authentication must adhere to industry standards to prevent unauthorized access.
   Accommodation:
   - The system employs OAuth 2.0 and HTTPS to secure API communication and protect sensitive user data.

2.4.2 External System Constraints

1. **AWS Cloud Services:**
   The application relies on Amazon Elastic Beanstalk for hosting and Amazon RDS for database management.
   Constraints:
   - Integration with AWS services requires adherence to AWS-specific configurations and APIs.
     Accommodation:
   - The system design follows best practices for AWS deployment, including Docker containerization and optimized resource allocation.

2. **Email Notifications:**
   The system sends notifications to users via email.
   Constraint:
   - The chosen email service (e.g., Amazon SES) supports plain-text and basic HTML messages.
     Accommodation:
   - Notification messages are designed using plain text and lightweight HTML templates to ensure compatibility and prevent delays.

2.4.3 Platform Constraints

1. **Languages and Frameworks:**
   The backend is developed in Python using Flask for API development, while the frontend uses React for user interfaces.
   Constraints:
   - The system is limited by the capabilities and libraries available in these technologies.
     Accommodation:
   - Libraries and tools such as SQLAlchemy for database integration and React Router for client-side routing ensure a robust implementation.
2. **Relational Database:**
   The system uses Amazon RDS with a relational schema.
   Constraint:
   - Relational databases require strict schema definitions, limiting flexibility for unstructured data.
     Accommodation:
   - The database schema is designed to balance flexibility and performance, with support for future migrations.

2.4.4 Design-Driven Constraints

1. **Trade-Offs in Conflict Resolution:**
   The scheduling system prioritizes accuracy over speed, which might lead to slight delays during high-load periods.
   Design Choice:
   - A heuristic-based conflict resolution algorithm was chosen over simpler approaches due to its higher accuracy in resolving complex scheduling issues.
   - While this introduces minimal processing delays, it significantly enhances reliability.
2. **Resource Allocation in AWS:**
   The system is deployed in a cloud-native architecture to ensure scalability.
   Design Choice:
   - RDS is selected for their seamless scaling capabilities, even though they incur higher initial costs compared to self-hosted solutions.

2.4.5 Assumptions

1. **Stable Internet Connection:**
   The system assumes that users will access the application with a stable internet connection for smooth interactions.
2. **User Proficiency:**
   Users (e.g., league organizers) are assumed to have basic proficiency in using web applications.
3. **AWS Availability:**
   The system assumes uninterrupted availability of AWS services for hosting and database management.

3 Interfaces and Data Stores

This section describes the interfaces into and out of the system as well as the data stores you will be including in your system.

3.1 System Interfaces

In this section, we define the various interfaces provided by the SPORTS system to users and external systems. For example, we describe the ***User Interface****, which allows players and managers to interact with the system for tasks like registration, scheduling, and viewing match results. We also detail external integrations, such as the* ***Weather API Interface****, which helps us fetch real-time weather data for venue scheduling, and the* ***Payment Gateway Interface****, which enables secure payment processing during team registration.*

Additionally, we explain how we handle data export and import, such as through the ***Database Interface****, which manages data storage and retrieval for users, teams, matches, and schedules. For GUI-based interfaces like the* ***Venue Scheduling Interface****, we describe them using clear prose to highlight how venue managers can input and modify availability directly in the system. Instead of including JavaDocs or purely technical details, we focus on high-level functionality and usability to ensure clarity for all stakeholders.*

Interface Name	**Type**	**Description**	**Inputs/Outputs**	**Associated Requirements**

User Interface	GUI	Provides users (league managers, players, etc.) with an intuitive interface for interacting with the system.	Inputs: User actions (e.g., registration, scheduling). Outputs: Feedback, schedules.	3.1, 4.1, 4.8
RESTful API	Software Interface	Enables communication between the client and server using JSON for data exchange.	Inputs: HTTP requests (e.g., GET/POST). Outputs: JSON responses.	3.3, 4.1
Payment Gateway	Software Interface	Facilitates secure transactions for user and team registration payments.	Inputs: Payment details. Outputs: Payment confirmation or errors.	4.1.4, 4.2.4
Database Interface	Software Interface	Handles data storage and retrieval for user, team, match, and schedule information.	Inputs: SQL queries. Outputs: Data records.	3.3, 4.6

Weather API	Software Interface	Integrates with external APIs to fetch real-time weather updates for venue scheduling.	Inputs: Venue details. Outputs: Weather conditions and alerts.	4.5.3
City Regulations System	Software Interface	Ensures compliance by accessing and verifying local regulatory requirements.	Inputs: Venue and match details. Outputs: Compliance status.	4.5.1, 4.5.2
Venue Scheduling Interface	GUI	Allows venue managers to input and modify venue schedules.	Inputs: Venue and schedule details. Outputs: Updated schedules.	4.3, 4.7
Notification System	Communicatio n	Sends email updates for changes in schedules, payments, and other key events.	Inputs: Notification details. Outputs: Email notifications.	4.7.3, 4.5

3.1.1 User Interface

This interface provides an interactive graphical environment for users, including league managers, team managers, and players, to access the SPORTS system. Users can perform tasks such as registering teams, scheduling matches, and viewing statistics. The interface is designed for both desktop and mobile platforms, ensuring accessibility. Data is transmitted to the backend using RESTful APIs for further processing.

3.1.2 RESTful API

This interface facilitates communication between the client and server. It uses HTTP methods such as GET, POST, PUT, and DELETE to enable data exchange in JSON format. This API supports functionalities like user authentication, venue scheduling, match registration, and payment processing. It ensures secure data transfer using HTTPS protocols and follows RESTful design principles to maintain compatibility with various client applications.

3.1.3 Payment Gateway Interface

This interface enables users to complete secure financial transactions during the registration process. It supports multiple payment methods, such as credit cards, debit cards, and online banking. A third-party payment processor is used to handle these transactions, and sensitive payment details are encrypted during transmission. Confirmation messages are provided for successful payments, while errors are promptly reported with suggestions for resolution.

3.1.4 Weather API Interface

This interface connects to external weather services, such as OpenWeather, to retrieve real-time weather data. The system uses this information to monitor venue conditions and ensure event safety. Data fields include temperature, precipitation, and severe weather alerts. The API queries are triggered during venue scheduling processes and provide updated weather information as needed.

3.1.5 Notification System Interface

This interface ensures that users are informed about key updates, such as schedule changes, payment confirmations, or weather alerts. Notifications are sent via email or displayed in-app. The system formats messages based on the event type and delivers them securely through SMTP or equivalent protocols.

3.1.6 Database Interface

This interface enables the system to store and retrieve data from the backend database, such as user information, team registrations, match schedules, and payment records. It uses SQL queries to interact with a relational database (e.g., MySQL or PostgreSQL). The interface supports CRUD operations (Create, Read, Update, Delete) and ensures data integrity through transaction management and error handling mechanisms.

3.1.7 City Regulations System Interface

This interface connects to external city regulation platforms to ensure compliance with local rules for event scheduling and venue use. It verifies permits, venue restrictions, and other regulatory requirements during the match and venue scheduling process. Data exchanged includes venue IDs, event types, and regulation status.

3.1.8 Venue Scheduling Interface

This interface allows venue managers to input, update, and manage venue availability. It provides a calendar view for easy scheduling and prevents conflicts by checking overlapping time slots. The interface accepts inputs like venue names, dates, and available time slots and updates the schedule database accordingly.

3.1.9 Search Interface

This interface provides a search functionality for users to find teams, leagues, and venues based on specific keywords. It supports full-text search and partial keyword matches, offering real-time suggestions as the user types. Results are displayed in a structured format, including relevant details like team rosters, schedules, or venue addresses.

3.1.10 Player Schedule Interface

This interface enables players to view their personal schedules for registered leagues and matches. It fetches data from the schedule database and displays it in a calendar-like format. The system ensures real-time updates for any changes in match schedules and notifies users of new updates via the notification system.

3.2 Data Stores

This section outlines the data stores in the SPORTS system, including the User Data Store for credentials and roles, the Team Data Store and Match Data Store for managing team details, match results, and statistics. Specialized stores, such as the Venue Data Store for availability, the Weather Data Store for real-time updates, and the Payment Data Store for transactions, ensure smooth operations. The Regulation Compliance Store tracks adherence to city regulations, supporting efficient event management and scheduling.

Data Store Name	Description	Schema Details	Associated Requirements
User Data Store	Stores user information, including login credentials and roles.	Fields: userID, name, email, password, role	3.3, 4.1
Team Data Store	Stores team-related information, including members and affiliations.	Fields: teamID, teamName, sportsType, members	4.2
Match Data Store	Stores match details, results, and statistics.	Fields: matchID, teams, date, location, score, statistics	4.6
Venue Data Store	Maintains venue information and availability.	Fields: venueID, name, location, capacity, facilities, availability	4.3
Schedule Data Store	Handles scheduling data for teams, venues, and matches.	Fields: scheduleID, matchID, venueID, timeSlot	4.7
Payment Data Store	Tracks payment transactions for users and teams.	Fields: paymentID, userID, amount, status	4.1.4, 4.2.4
Weather Data Store	Temporarily stores real-time weather information fetched via API.	Fields: venueID, temperature, humidity, windSpeed, precipitation, alerts	4.5
Regulation Compliance Store	Tracks venue compliance with city regulations.	Fields: venueID, regulationStatus, warnings, timestamps	4.5

3.2.1 User Data Store

This data store securely maintains user-related information, including login credentials, names, email addresses, and roles. It supports authentication and role-based access control for league managers, team managers, and players. The data is structured with fields like userID, name, email, password, and role, ensuring secure access and functionality for authorized users.

3.2.2 Team Data Store

This data store organizes information about teams, including their names, sports type, and associated members. It helps manage team registrations and affiliations within the system. The schema includes fields like teamID, teamName, sportsType, and members, supporting dynamic updates as teams evolve throughout the season.

3.2.3 Match Data Store

This data store captures match-specific details, including participating teams, schedules, locations, and results. It plays a critical role in league management by storing and retrieving match outcomes and statistics. Schema fields include matchID, teams, date, location, score, and statistics, facilitating efficient reporting and analysis.

3.2.4 Venue Data Store

This data store manages information about venues, such as their names, locations, capacity, and facilities. It also tracks venue availability for scheduling purposes. The schema includes fields like venueID, name, location, capacity, and availability, ensuring conflicts are avoided during match planning.

3.2.5 Schedule Data Store

This data store handles the scheduling data for matches, teams, and venues. It ensures time slots are appropriately allocated and conflicts are avoided. The schema includes fields like scheduleID, matchID, venueID, and timeSlot, enabling efficient time management for all league events.

3.2.6 Payment Data Store

This data store tracks payment transactions for user and team registrations. It securely records details such as paymentID, userID, amount, and status, ensuring transparency and reliability for financial operations.

3.2.7 Weather Data Store

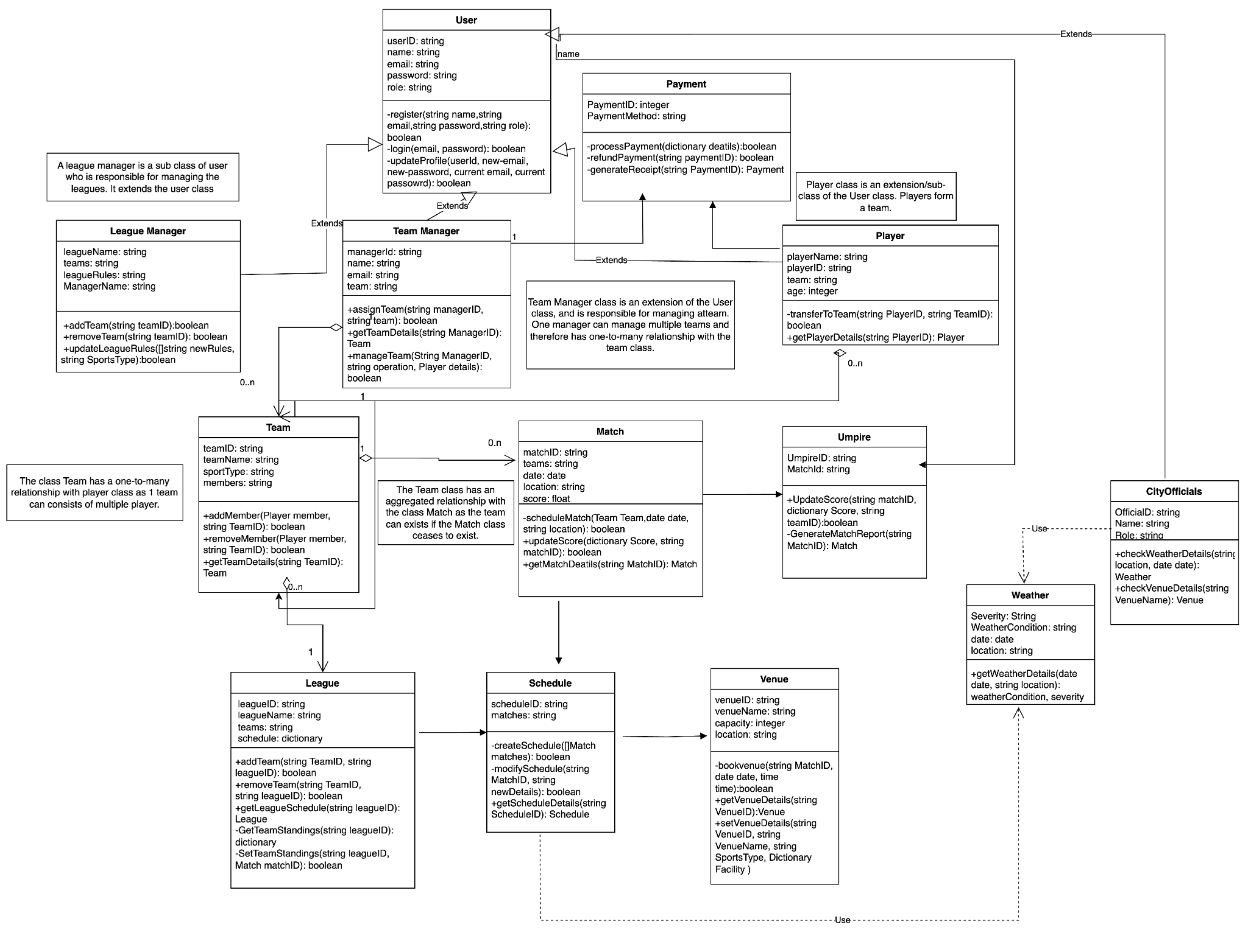
This temporary data store integrates with external APIs to store real-time weather updates for venues. It includes fields like venueID, temperature, humidity, windSpeed, precipitation, and alerts, supporting weather-based decisions for scheduling.

3.2.8 Regulation Compliance Store

This data store monitors venue compliance with city regulations. It includes fields such as venueID, regulationStatus, warnings, and timestamps, ensuring events adhere to local policies and safety standards.

4 Structural Design

4.1 Class Diagram


User
userID: string
name: string
email: string
password: string
role: string
-register(string name,string email,string password,string role): boolean
-login(email, password): boolean
-updateProfile(userId, new-email, new-password, current email, current passowrd): boolean
Extends
name
Payment
PaymentID: integer
PaymentMethod: string
-processPayment(dictionary deatils):boolean
-refundPayment(string paymentID): boolean
-generateReceipt(string PaymentID): Payment
A league manager is a sub class of user who is responsible for managing the leagues. It extends the user class
Player class is an extension/sub-class of the User class. Players form a team.
Extends
Extends
League Manager
leagueName: string
teams: string
leagueRules: string
ManagerName: string
+addTeam(string teamID):boolean
+removeTeam(string teamID): boolean
+updateLeagueRules([]string newRules, string SportsType):boolean
Team Manager
managerId: string
name: string
email: string
team: string
+assignTeam(string managerID, string team): boolean
+getTeamDetails(string ManagerID): Team
+manageTeam(String ManagerID, string operation, Player details): boolean
1
Extends
Player
playerName: string
playerID: string
team: string
age: integer
-transferToTeam(string PlayerID, string TeamID): boolean
+getPlayerDetails(string PlayerID): Player
Team Manager class is an extension of the User class, and is responsible for managing atteam. One manager can manage multiple teams and therefore has one-to-many relationship with the team class.
0..n
0..n
1
Team
teamID: string
teamName: string
sportType: string
members: string
+addMember(Player member, string TeamID): boolean
+removeMember(Player member, string TeamID): boolean
+getTeamDetails(string TeamID): Team
1
0.n
0..n
The class Team has a one-to-many relationship with player class as 1 team can consists of multiple player.
The Team class has an aggregated relationship with the class Match as the team can exists if the Match class ceases to exist.
Match
matchID: string
teams: string
date: date
location: string
score: float
-scheduleMatch(Team Team,date date, string location): boolean
+updateScore(dictionary Score, string matchID): boolean
+getMatchDeatils(string MatchID): Match
Umpire
UmpireID: string
MatchId: string
+UpdateScore(string matchID, dictionary Score, string teamID):boolean
-GenerateMatchReport(string MatchID): Match
CityOfficials
OfficialD: string
Name: string
Role: string
+checkWeatherDetails(string location, date date): Weather
+checkVenueDetails(string VenueName): Venue
Use
Weather
Severity: String
WeatherCondition: string
date: date
location: string
+getWeatherDetails(date date, string location): weatherCondition, severity
1
League
leagueID: string
leagueName: string
teams: string
schedule: dictionary
+addTeam(string TeamID, string leagueID): boolean
+removeTeam(string TeamID, string leagueID): boolean
+getLeagueSchedule(string leagueID): League
-GetTeamStandings(string leagueID): dictionary
-SetTeamStandings(string leagueID, Match matchID): boolean
Schedule
scheduleID: string
matches: string
-createSchedule([]Match matches): boolean
-modifySchedule(string MatchID, string newDetails): boolean
+getScheduleDetails(string ScheduleID): Schedule
Venue
venueID: string
venueName: string
capacity: integer
location: string
-bookvenue(string MatchID, date date, time time):boolean
+getVenueDetails(string VenueID):Venue
+setVenueDetails(string VenueID, string VenueName, string SportsType, Dictionary Facility)
Use

4.2 Class Description

4.3.1 Class: User

Purpose:
To represent the User entity of the system, storing relevant information such as user identification, authentication credentials, and role within the system. This class handles user registration, login, and profile updates.
Constraints:
- userID must be unique for each user.
- password must be stored securely (e.g., encrypted).
- email must follow a valid email format.
- role should be one of the predefined roles (e.g., Admin, Regular User).

Persistent:
This class would be persistent, as user information needs to be stored in a database to maintain user state across sessions.

4.3.1.1 Attribute Descriptions
1. Attribute: userID
   - Type: String (or Integer, depending on the implementation)
   - Description: A unique identifier for each user in the system.
   - Constraints: Must be unique across all users.
2. Attribute: name
   - Type: String
   - Description: The full name of the user.
   - Constraints: Non-empty.
3. Attribute: email
   - Type: String
   - Description: The email address associated with the user.
   - Constraints: Must be a valid email format (e.g., "user@example.com"). Should be unique across all users.
4. Attribute: password
   - Type: String (encrypted)
   - Description: The password associated with the user's account, stored securely.
   - Constraints: Should follow security guidelines (e.g., minimum length, mixed characters).
5. Attribute: role
   - Type: String
   - Description: The role or permission level of the user (e.g., "Admin", "Regular User").
   - Constraints: Must be one of the predefined roles.

4.3.1.2 Method Descriptions

1. Method: register()
    - Return Type: boolean
    - Parameters:
        - name: String (User's name)
        - email: String (User's email)
        - password: String (User's password)
        - role: String (User's role)
    - Return Value: Success or failure (true or false)
    - Pre-condition: Email is valid, password is secure, role is predefined.
    - Post-condition: User is registered in the system, with data stored in the database.
    - Attributes read/used: name, email, password, role
    - Methods called: Validate email, Validate password, Save user data to database.
    - Processing Logic:
        - Validate the user's email and password.
        - Encrypt the password before storing it in the database.
        - Add the user to the system with the appropriate role.
2. Test case: Register a user with valid email, name, and password. The expected result is: User is successfully registered and stored in the database.

2. Method: login()
    - Return Type: boolean
    - Parameters:
        - email: String (User's email)
        - password: String (User's password)
    - Return Value: Success or failure (true or false)
    - Pre-condition: User's email and password exist in the database.
    - Post-condition: User is authenticated and logged in to the system.
    - Attributes read/used: email, password
    - Methods called: Validate user credentials, check stored password hash.
    - Processing Logic:
        - Retrieve the user data based on the provided email.
        - Validate the password by comparing the entered password with the stored password hash.
        - If credentials are correct, the user is logged in.

Test case: Login with valid credentials (email and password). Expected output: User successfully logs in.

3. Method: updateProfile()

    - Return Type: boolean
    - Parameters:
        - userID: String (User's unique identifier)
        - newName: String (New name for the user)
        - newEmail: String (New email for the user)
        - newPassword: String (New password for the user)
    - Return Value: Success or failure (true or false)
    - Pre-condition: The user is logged in and has the necessary permissions.
    - Post-condition: The user’s profile is updated in the system.
    - Attributes read/used: userID, newName, newEmail, newPassword
    - Methods called: Validate new email, Encrypt new password, Update user data in the database.
    - Processing Logic:
        - Validate the new email and password.
        - If valid, update the user's profile data in the database.

4. Test case: Update the profile with valid details. Expected output: User profile is successfully updated.

4.3.2 Class: TeamManager

Purpose:

To represent the Team Manager entity in the system. This class extends the User class and inherits its attributes and methods. In addition, it provides functionality for assigning teams, retrieving team details, and managing team information.

Constraints:

- managerId must be unique for each team manager.
- email must follow a valid email format and be unique.
- team must reference a valid team entity in the system.

Persistent:

This class would be persistent, as the team manager’s data, including their assigned team and personal details, needs to be stored in the database for future reference.

4.3.2.1 Attribute Descriptions

1. Attribute: managerId
    - Type: String (or Integer, depending on the implementation)
    - Description: A unique identifier for each team manager.

    - Constraints: Must be unique across all team managers.

2. Attribute: email

- Type: String
- Description: The email address associated with the team manager.
- Constraints: Must be a valid email format (e.g., "manager@example.com") and unique.

3. Attribute: team

- Type: String (or Object, depending on the implementation)
- Description: The team assigned to the manager.
- Constraints: Must reference an existing and valid team entity.

4.3.2.2 Method Descriptions

1. Method: assignTeam()
    - Return Type: boolean
    - Parameters:
        - managerId: String (Unique identifier of the manager)
        - team: String (Identifier or name of the team to be assigned)
    - Return Value: Success or failure (true or false).
    - Pre-condition:
        - The manager exists in the system.
        - The team exists and is valid.
    - Post-condition:
        - The team is assigned to the manager.
        - Data is updated in the database.
    - Attributes read/used: managerId, team
    - Methods called: Validate team, Update team assignment in the database.
    - Processing Logic:
        - Verify that the team exists.
        - Assign the team to the manager.
        - Update the manager's record in the database.
    - Test Case: Assign a valid team to a manager. Expected result: Team is successfully assigned, and changes are reflected in the database.
2. Method: getTeamDetails()
    - Return Type: Team Object
    - Parameters:
        - managerId: String (Unique identifier of the manager)
    - Return Value: The details of the team managed by the manager.
    - Pre-condition: The manager exists in the system and is assigned a team.
    - Post-condition: The team details are retrieved and returned.
    - Attributes read/used: managerId, team

- Methods called: Fetch team details from the database.
- Processing Logic:
    - Retrieve the manager's assigned team based on managerId.
    - Fetch team details from the database.
    - Return the details.
- Test Case: Retrieve team details for a valid manager. Expected result: Accurate team details are returned.

3. Method: manageTeam()
    - Return Type: boolean
    - Parameters:
        - managerId: String (Unique identifier of the manager)
        - operation: String (The action to perform, e.g., "addPlayer", "removePlayer").
        - details: Player Object (Information required for the operation).
    - Return Value: Success or failure (true or false).
    - Pre-condition:
        - The manager is logged in and authorized to manage the team.
        - The operation is valid and supported.
    - Post-condition:
        - The team data is updated based on the operation.
    - Attributes used: managerId, team
    - Methods called: Validate operation, Update team data in the database.
    - Processing Logic:
        - Verify the manager's permissions to manage the team.
        - Validate the requested operation.
        - Perform the operation and update the team data in the database.
    - Test Case: Perform an "addPlayer" operation for a valid team. Expected result: Player is added successfully, and the database reflects the updated team.

4.3.3 Class: Team

Purpose:
To represent the Team entity in the system, storing relevant information about teams, their members, and associated sports. This class manages team-related operations such as adding or removing members and retrieving team details.

Constraints:

- teamID must be unique for each team.
- teamName must be non-empty.

- sportsType must be a valid sport recognized by the system.
- Each team must have at least one member after creation.

Persistent:
This class would be persistent, as team-related data needs to be stored in the database for managing leagues, members, and schedules effectively.

4.3.3.1 Attribute Descriptions

1. Attribute: teamID
   Type: String (or Integer)
   Description: A unique identifier for each team.
   Constraints: Must be unique across all teams.
2. Attribute: teamName
   Type: String
   Description: The name of the team.
   Constraints: Must be non-empty.
3. Attribute: sportsType
   Type: String
   Description: The type of sport associated with the team (e.g., football, basketball).
   Constraints: Must be a valid sport recognized by the system.
4. Attribute: members
   Type: List of User objects
   Description: A list of users who are members of the team.
   Constraints: Must contain at least one user after creation.

4.3.3.2 Method Descriptions

1. Method: addMember()
   - Return Type: boolean
     Parameters:
     i. member: Player object (The member to be added)
   - Team ID: String team Id to which player needs to be added
   - Return Value: Success or failure (true or false)
   - Pre-condition: Members should not already be in the team.
   - Post-condition: The new member is added to the team.
   - Attributes read/used: members
   - Processing Logic:
     i. Check if the member is already in the team.
     ii. If not, add the member to the team.

- Test case: Add a valid new member to a team. Expected output: Member is successfully added.

2. Method: removeMember()
    - Return Type: boolean
      Parameters:
        i. memberID: String (Unique identifier of the member.
        ii. Team ID: team Id to which player needs to be removed from.
    - Return Value: Success or failure (true or false)
    - Pre-condition: Members should already be part of the team.
    - Post-condition: The member is removed from the team.
    - Attributes read/used: members
      Processing Logic:
        i. Check if the member exists in the team.
        ii. If found, remove the member.
    - Test case: Remove an existing member. Expected output: Member is successfully removed.
3. Method: getTeamDetails()
    - Return Type: Team Object
    - Parameters: Team ID: team Id to which we want details.
    - Return Value: A formatted string containing team details.
    - Pre-condition: The team must exist in the system.
    - Post-condition: Team details are retrieved.
    - Attributes read/used: teamID, teamName, sportsType, members
    - Processing Logic:
        i. Retrieve and format team details (ID, name, sport type, and members).
    - Test case: Fetch details of an existing team. Expected output: A string with all relevant team details.

4.3.4 Class: League

Purpose:
To represent the League entity, which organizes multiple teams and schedules matches. This class manages league-related operations like adding and removing teams and retrieving schedules.

Constraints:

- leagueID must be unique for each league.
- leagueName must be non-empty.
- teams should contain at least two teams.
- schedule must be valid and conflict-free.

Persistent:
Yes, this class would be persistent as league information, including teams and schedules, needs to be maintained across sessions.

4.3.4.1 Attribute Descriptions

1. Attribute: leagueID
   Type: String (or Integer)
   Description: A unique identifier for the league.
   Constraints: Must be unique across all leagues.
2. Attribute: leagueName
   Type: String
   Description: The name of the league.
   Constraints: Must be non-empty.
3. Attribute: teams
   Type: List of Team objects
   Description: A list of teams participating in the league.
   Constraints: Must contain at least two teams.
4. Attribute: schedule
   Type: Dictionary
   Description: The schedule of matches for the league.
   Constraints: Must be valid and conflict-free.

4.3.4.2 Method Descriptions

1. Method: addTeam()
   - Return Type: boolean
   - Parameters:
   - team: String Team ID (The team to be added)
   - League ID: League to which the team needs to be added.
   - Return Value: Success or failure (true or false)
   - Pre-condition: The team is not already part of the league.
   - Post-condition: The team is added to the league.
   - Attributes read/used: teams
   - Processing Logic:
     i. Check if the team already exists in the league.
     ii. If not, add the team to the league.
   - Test case: Add a new team to the league. Expected output: Team is successfully added.
2. Method: removeTeam()
   - Return Type: boolean
   - Parameters:

    - teamID: String (Unique identifier of the team)
    - League ID: League to which the team needs to be removed from.
    - Return Value: Success or failure (true or false)
    - Pre-condition: The team must exist in the league.
    - Post-condition: The team is removed from the league.
    - Attributes read/used: teams
    - Processing Logic:
        i. Check if the team exists in the league.
        ii. If found, remove the team.
    - Test case: Remove an existing team from the league. Expected output: Team is successfully removed.
3. Method: getSchedule()
    - Return Type: League Object
    - Parameters: League ID - schedule for the league ID specified.
    - Return Value: A formatted string containing the league schedule.
    - Pre-condition: The league must have a valid schedule.
    - Post-condition: The schedule is retrieved.
    - Attributes read/used: schedule
    - Processing Logic:
        i. Fetch and format the league schedule.
    - Test case: Fetch the schedule of an existing league. Expected output: A string with all scheduled matches.

4.3.5 Class: Match

Purpose:
To represent an individual match within a league, including details about participating teams, match location, and score updates.

Constraints:

- matchID must be unique for each match.
- teams must contain exactly two teams.
- date must be in a valid format and not conflict with existing schedules.
- location must be a valid venue recognized by the system.

Persistent:
Yes, match details should be stored for historical records, scheduling, and reporting purposes.

4.3.5.1 Attribute Descriptions

1. Attribute: matchID
   Type: String (or Integer)
   Description: A unique identifier for each match.
   Constraints: Must be unique across all matches.
2. Attribute: teams
   Type: List of two Team objects
   Description: The two teams participating in the match.
   Constraints: Exactly two teams must be specified.
3. Attribute: date
   Type: Date/Time
   Description: The date and time of the match.
   Constraints: Must not conflict with existing matches in the schedule.
4. Attribute: location
   Type: Venue object
   Description: The venue where the match will take place.
   Constraints: Must be a valid venue in the system.
5. Attribute: score
   Type: JSON or Map
   Description: Records the scores of the two teams.
   Constraints: Can only be updated after the match starts.

4.3.5.2 Method Descriptions

1. Method: scheduleMatch()
   - Return Type: boolean
   - Parameters:
     i. teams: List of Team objects
     ii. date: Date/Time
     iii. location: Venue object
   - Return Value: Success or failure (true or false)Pre-condition: Teams and location must be available on the specified date.
   - Post-condition: The match is scheduled in the system.
   - Attributes read/used: teams, date, location
   - Processing Logic:
     i. Validate the availability of the teams and venue.
     ii. Schedule the match if constraints are satisfied.
2. Method: updateScore()
   - Return Type: boolean
   - Parameters:
     i. score: Dictionary
   - Return Value: Success or failure (true or false)

- Pre-condition: The match must have started.
    - Post-condition: The score is updated in the system.
    - Attributes read/used: score
    - Processing Logic:
        i. Validate the match has started.
        ii. Update the score with new values.
3. Method: getMatchDetails()
    - Return Type: Match Object
    - Parameters: Match ID String.
    - Return Value: A formatted string with match details.
    - Attributes read/used: matchID, teams, date, location, score
    - Processing Logic:
    - Retrieve and format match details.

4.3.6 Class: Schedule

Purpose:
To represent the schedule for a league, organizing matches by date, time, and venue.

Constraints:

- scheduleID must be unique.
- matches must not contain overlapping times or conflicting venues.

Persistent:
Yes, the schedule is critical for match organization and should be stored persistently.

4.3.6.1 Attribute Descriptions

1. Attribute: scheduleID
   Type: String (or Integer)
   Description: A unique identifier for the schedule.
   Constraints: Must be unique.
2. Attribute: matches
   Type: List of Match objects
   Description: The matches included in the schedule.
   Constraints: Must not have time or venue conflicts.

4.3.6.2 Method Descriptions

1. Method: createSchedule()
    - Return Type: boolean
    - Parameters:

        i. matches: List of Match objects
    - Return Value: Success or failure (true or false)
    - Processing Logic:
        i. Validate match times and venues.
        ii. Add matches to the schedule if constraints are met.
2. Method: modifySchedule()
    - Return Type: boolean
    - Parameters:
        i. matchID: String
        ii. newDetails: Dictionary (Updated match details)
    - Return Value: Success or failure (true or false)
    - Processing Logic:
        i. Locate the match in the schedule.
        ii. Update its details if valid.
3. Method: getScheduleDetails()
    - Return Type: Schedule Object
    - Parameters: Schedule ID

4.3.7 Class: Venue

Purpose:
To represent a venue where matches can be scheduled, including its name, location, and capacity.

Constraints:

- venueID must be unique.
- capacity must be greater than 0.

Persistent:
Yes, venue information must be stored for booking and management purposes.

4.3.7.1 Attribute Descriptions

1. Attribute: venueID
   Type: String
   Description: A unique identifier for the venue.
2. Attribute: venueName
   Type: String
   Description: The name of the venue.

3. Attribute: capacity
   Type: Integer
   Description: The seating capacity of the venue.
4. Attribute: location
   Type: String
   Description: The address or location of the venue.

4.3.7.2 Method Descriptions

1. Method: bookVenue()
   Return Type: boolean
   Parameters:
    - matchID: String Match ID for which the venue is booked
    - date: Date/Time
      Return Value: Success or failure (true or false)
2. Method: getVenueDetails()
   Return Type: Venue Object
   Parameters: Venue ID

4.3.8 Class: Umpire

Purpose:
To represent the umpire officiating a match, including reporting match results and updating scores.

4.3.8.1 Attribute Descriptions

1. Attribute: umpireID
   Type: String
   Description: Unique identifier for the umpire.
2. Attribute: matchID
   Type: String
   Description: Identifier for the match being officiated.

4.3.8.2 Method Descriptions

1. Method: updateScore()
    - Return Type: boolean
    - Parameters: Match ID, Team ID, Score
2. Method: getMatchReport()
    - Return Type: Match Object
    - Parameters: Match ID

4.3.9 Class: Player

Purpose:
To represent the player within the system, managing their team affiliation, personal details, and interactions with the league.

Constraints:

- A player must be assigned to at least one team.
- Players can only belong to one team at any given time.

Persistent:
Yes, the player's information must be stored in a database for tracking their participation, team affiliation, and other relevant data.

4.3.9.1 Attribute Descriptions

1. Attribute: playerName
   Type: String
   Description: The name of the player.
   Constraints: Non-empty string.
2. Attribute: playerID
   Type: String
   Description: A unique identifier for the player.
   Constraints: Must be unique for each player.
3. Attribute: teams
   Type: List of Team objects
   Description: The teams the player is associated with.
   Constraints: A player can only belong to one team at a time.
4. Attribute: age
   Type: Integer
   Description: The age of the player.
   Constraints: Must be a positive integer.

4.3.9.2 Method Descriptions

1. Method: transferToTeam()
   - Return Type: boolean
   - Parameters:
     i. team: Team object
   - Return Value: Success or failure (true or false)
   - Pre-condition: The player must not already be in the team.

- Post-condition: The player is transferred to the new team.
- Attributes used: playerID, team ID
- Process Logic:
    i. The method checks if the player is not already a member of the given. team and then assigns the player to the new team.
- Test case: Transfer a player to a new team, verify the team membership.

2. Method: getPlayerDetails()
    - Return Type: Player object
    - Parameters: Player ID - string.
    - Return Value: Returns the player’s details - Player Object.
    - Pre-condition: The player must exist in the system.
    - Post-condition: Returns the details of the player.
    - Attributes read/used: playerName, playerID, teams, age
    - Methods called: Retrieve player details from the database.
    - Processing Logic: Retrieve and return all the relevant player details from the system based on the player's unique identifier.
    - Test case: Retrieve details of a player by their ID.

4.3.10 Class: League Manager

Purpose:
Represents the person responsible for managing a league, including the addition and removal of teams and updating league rules.

Constraints:

- A League Manager can manage multiple teams but each league manager is linked to a single league.

Persistent:
Yes, league manager information must be persistently stored for league governance.

4.3.10.1 Attribute Descriptions

1. Attribute: leagueName
   Type: String
   Description: Name of the league managed by the manager.
2. Attribute: teams
   Type: List of Team objects
   Description: Teams participating in the league.

3. Attribute: leagueRules
   Type: JSON or String
   Description: Rules for the league's functioning.
4. Attribute: managerName
   Type: String
   Description: Name of the manager responsible for the league.

4.3.10.2 Method Descriptions

1. Method: addTeam()
   - Return Type: boolean
   - Parameters:
     i. team: Team ID String, League ID String
   - Return Value: Success or failure (true or false)
   - Pre-condition: The team must not already be added to the league.
   - Post-condition: The team is added to the league.
   - Attributes read/used: teams
   - Methods called: Validate team, add team to league.
   - Processing Logic:
     i. The method checks if the team already exists in the league and adds it if it doesn't.
   - Test case: Add a new team to the league.
2. Method: removeTeam()
   - Return Type: boolean
   - Parameters:
   - team: Team ID String, League ID String
   - Return Value: Success or failure (true or false)
   - Pre-condition: The team must exist in the league.
   - Post-condition: The team is removed from the league.
   - Attributes read/used: teams
   - Methods called: Check if the team exists, remove team from league.
   - Processing Logic:
     i. The method checks if the team exists in the league before removing it.
   - Test case: Remove a team from the league.
3. Method: updateLeagueRules()
   - Return Type: boolean
   - Parameters:
     i. newRules: String Sports type, List of string - rules
   - Return Value: Success or failure (true or false)
   - Pre-condition: The new rules should be validated before updating.

- Post-condition: The league rules are updated.
    - Attributes read/used: leagueRules
    - Methods called: Validate and update league rules.
    - Processing Logic:
        i. The method updates the rules of the league with the new values.
    - Test case: Update the league rules with valid inputs.

4.3.11 Class: Payment

Purpose:
To manage financial transactions for players, including payments for team registration and refunds.

Constraints:

- Payment method must be valid.
- Each player/team must handle payments individually.

Persistent:
Yes, payment records need to be stored for tracking and auditing purposes.

4.3.11.1 Attribute Descriptions

1. Attribute: paymentID
   Type: String
   Description: Unique identifier for each payment.
2. Attribute: paymentMethod
   Type: String
   Description: Payment method used (e.g., Credit Card, PayPal).

4.3.11.2 Method Descriptions

1. Method: processPayment()
    - Return Type: boolean
    - Parameters:
        i. paymentDetails: dictionary object containing payment information.
    - Return Value: Success or failure (true or false)
    - Pre-condition: Payment details must be valid.
    - Post-condition: Payment is successfully processed.
    - Attributes read/used: paymentMethod
    - Methods called: Validate payment details, process payment.
    - Processing Logic:

        i. Process the payment using the specified method, returning success or failure based on the outcome.
    - Test case: Process a payment with valid details.
2. Method: refundPayment()
    - Return Type: boolean
    - Parameters:
        i. paymentID: String
    - Return Value: Success or failure (true or false)
    - Pre-condition: The payment ID must exist in the system.
    - Post-condition: Refund is processed.
    - Attributes read/used: paymentID
    - Methods called: Validate payment ID, process refund.
    - Processing Logic:
        i. Process a refund for the given payment ID.
    - Test case: Refund a payment with a valid payment ID.
3. Method: generateReceipt()
    - Return Type: Payment object
    - Parameters:
        i. paymentID: String
    - Return Value: A receipt for the transaction.
    - Pre-condition: The payment ID must be valid.
    - Post-condition: A receipt is generated.
    - Attributes read/used: paymentID
    - Methods called: Generate receipt for the payment.
    - Processing Logic:
        i. Generate and return a receipt for the specified payment.
    - Test case: Generate a receipt for a successful payment.

5 Dynamic Model

5.1 Scenarios

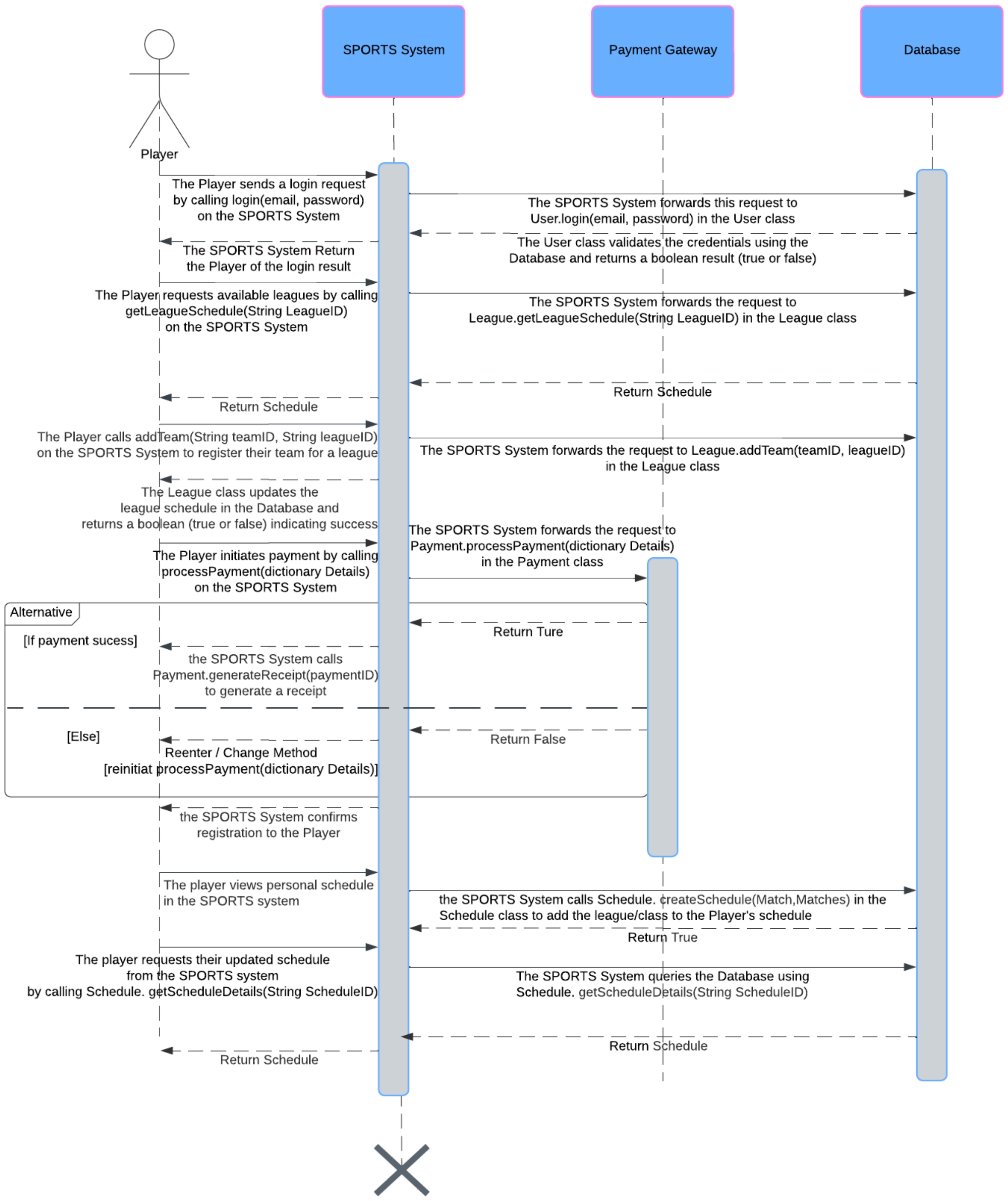
5.1.1 **Scenario Name: Player Registers for a League or Classs**

- Scenario Description:

  The player begins by logging into the SPORTS System using their credentials (email and password). The system validates the credentials by querying the database and provides feedback on whether the login was successful. Upon successful login, the player can view a list of available leagues by selecting the desired category or sport. The SPORTS System retrieves this information from the database and displays it to the player.

After selecting a specific league, the player submits their registration details, which the system processes to confirm eligibility. The player then proceeds to make the payment for registration by initiating the transaction. The SPORTS System forwards the payment request to the Payment Gateway, which validates and processes the payment. If the payment is successful, the registration is confirmed, and the player is added to the league. If the payment fails, the system notifies the player and allows them to retry. Finally, the player can access their updated personal schedule, which reflects their league registration.

- Sequence Diagram:

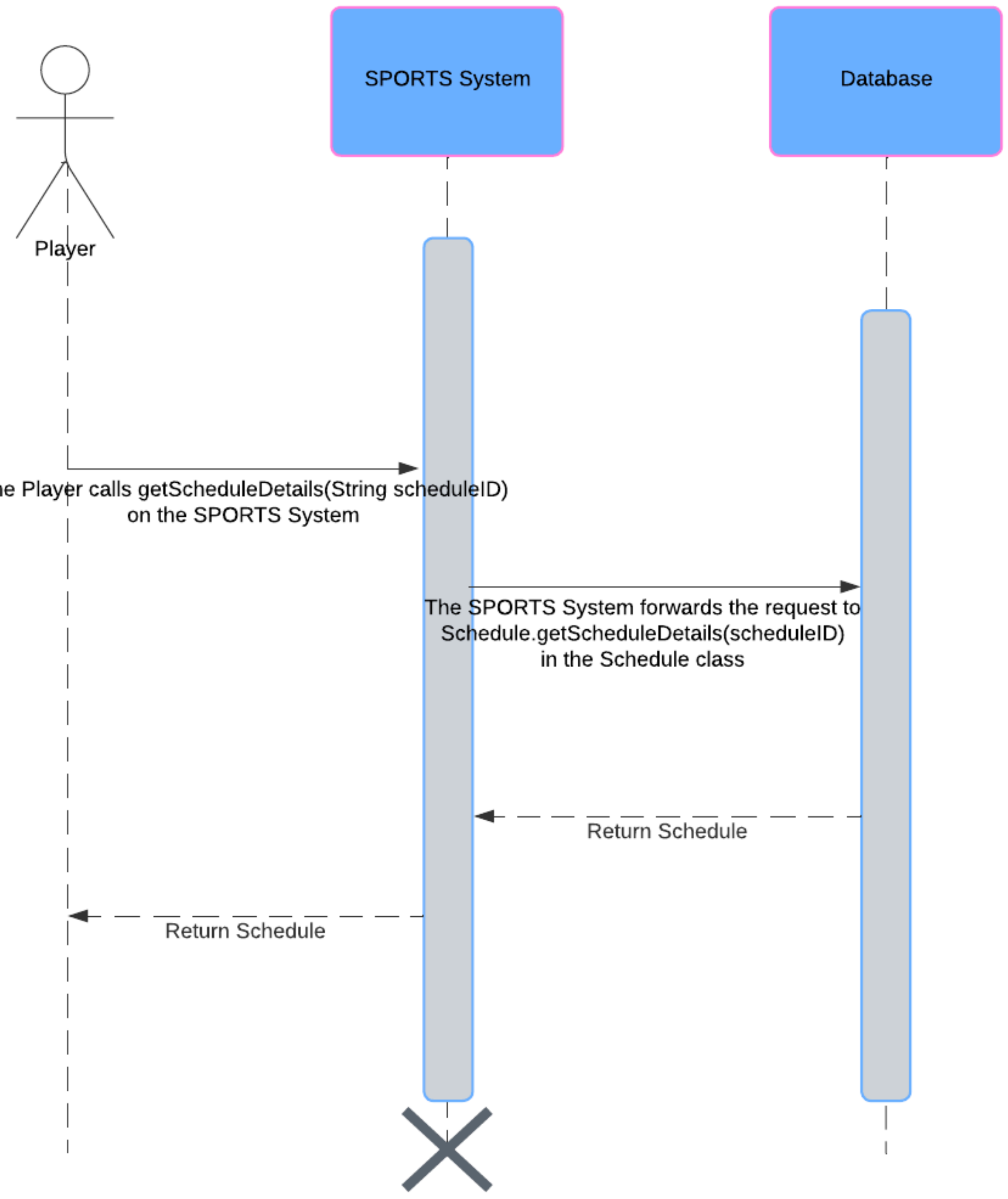

SPORTS System
Payment Gateway
Database
Player
The Player sends a login request by calling login(email, password) on the SPORTS System
The SPORTS System forwards this request to User.login(email, password) in the User class
The User class validates the credentials using the Database and returns a boolean result (true or false)
The SPORTS System Return the Player of the login result
The Player requests available leagues by calling getLeagueSchedule(String LeagueID) on the SPORTS System
The SPORTS System forwards the request to League.getLeagueSchedule(String LeagueID) in the League class
Return Schedule
Return Schedule
The Player calls addTeam(String teamID, String leagueID) on the SPORTS System to register their team for a league
The SPORTS System forwards the request to League.addTeam(teamID, leagueID) in the League class
The League class updates the league schedule in the Database and returns a boolean (true or false) indicating success
The SPORTS System forwards the request to Payment.processPayment(dictionary Details) in the Payment class
The Player initiates payment by calling processPayment(dictionary Details) on the SPORTS System
Alternative
[If payment sucess]
Return Ture
the SPORTS System calls Payment.generateReceipt(paymentID) to generate a receipt
[Else]
Return False
Reenter / Change Method [reinitiat processPayment(dictionary Details)]
the SPORTS System confirms registration to the Player
The player views personal schedule in the SPORTS system
the SPORTS System calls Schedule. createSchedule(Match,Matches) in the Schedule class to add the league/class to the Player's schedule
Return True
The player requests their updated schedule from the SPORTS system by calling Schedule. getScheduleDetails(String ScheduleID)
The SPORTS System queries the Database using Schedule. getScheduleDetails(String ScheduleID)
Return Schedule
Return Schedule

5.1.2 **Scenario Name: Player Views Personal Schedule**

- Scenario Description:

The player logs into the SPORTS System by entering their credentials. After the system verifies their login information, the player navigates to their personal schedule section. They submit a request to view their schedule, which includes upcoming matches, training sessions, and events. The SPORTS System retrieves this information from the database by querying the relevant schedule details and returns the data. The player can view their full schedule with details such as match times, opponents, and venues. This feature helps players stay organized and up-to-date with their commitments in the league.

- Sequence Diagram:


SPORTS System
Database
Player
The Player calls getScheduleDetails(String scheduleID) on the SPORTS System
The SPORTS System forwards the request to Schedule.getScheduleDetails(scheduleID) in the Schedule class
Return Schedule
Return Schedule

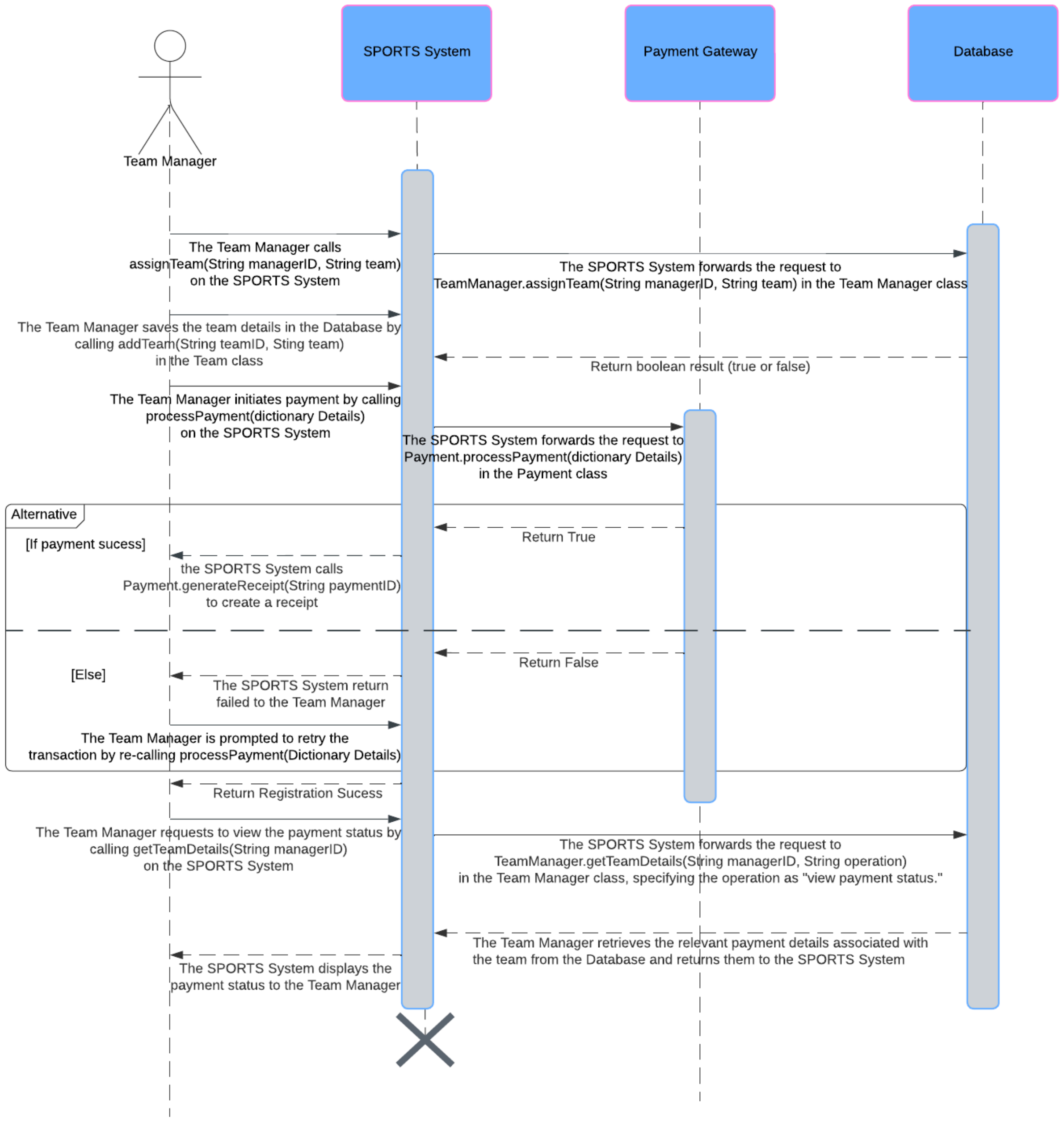
5.1.3 **Scenario Name: Team Manager Registers Team and Manages Payments**

- Scenario Description:

The Team Manager starts by logging into the SPORTS System to manage their team registrations. After successfully logging in, the manager assigns a team to their account by specifying the team name and associated league. The SPORTS System updates the team details in the database by invoking the relevant class methods.

After the manager initiates the payment process to complete the registration. The SPORTS System forwards the payment details to the Payment Gateway for processing. If the payment is successful, a receipt is generated, and the registration is confirmed. If the payment fails, the system notifies the manager and allows them to retry the payment. Additionally, the Team Manager can view the status of their payment by submitting a request. The SPORTS System retrieves the payment status from the database and displays it to the manager, providing transparency in the registration process.

- Sequence Diagram:


SPORTS System
Payment Gateway
Database
Team Manager
The Team Manager calls assignTeam(String managerID, String team) on the SPORTS System
The SPORTS System forwards the request to TeamManager.assignTeam(String managerID, String team) in the Team Manager class
The Team Manager saves the team details in the Database by calling addTeam(String teamID, Sting team) in the Team class
Return boolean result (true or false)
The Team Manager initiates payment by calling processPayment(dictionary Details) on the SPORTS System
The SPORTS System forwards the request to Payment.processPayment(dictionary Details) in the Payment class
Alternative
[If payment sucess]
Return True
the SPORTS System calls Payment.generateReceipt(String paymentID) to create a receipt
Return False
[Else]
The SPORTS System return failed to the Team Manager
The Team Manager is prompted to retry the transaction by re-calling processPayment(Dictionary Details)
Return Registration Sucess
The Team Manager requests to view the payment status by calling getTeamDetails(String managerID) on the SPORTS System
The SPORTS System forwards the request to TeamManager.getTeamDetails(String managerID, String operation) in the Team Manager class, specifying the operation as "view payment status."
The Team Manager retrieves the relevant payment details associated with the team from the Database and returns them to the SPORTS System
The SPORTS System displays the payment status to the Team Manager

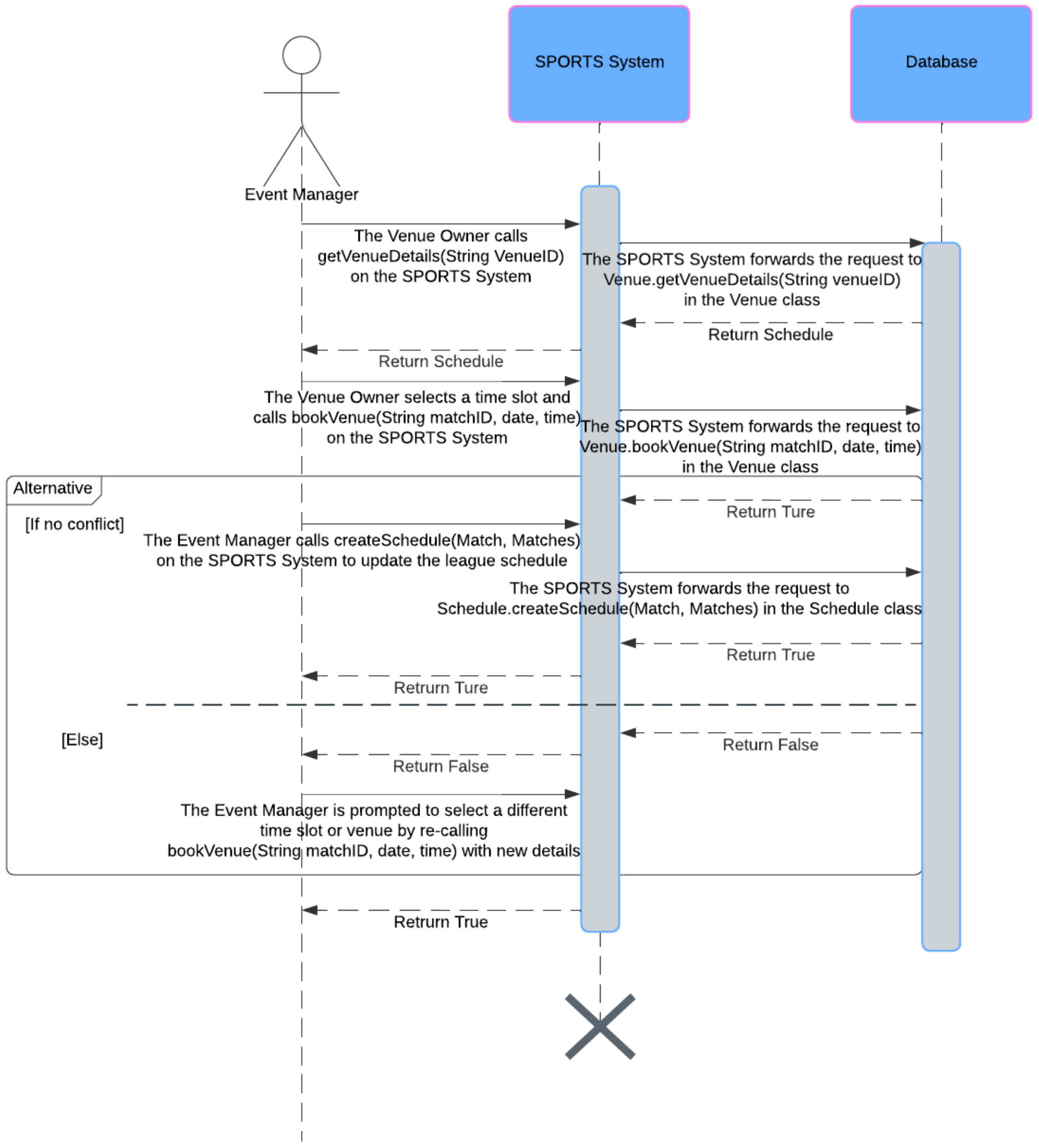
5.1.4 **Scenario Name: View and Manage Venue Schedule**

- Scenario Description:

  The Venue Owner or League Official logs into the SPORTS System to manage venue schedules. After authentication, they can view the current schedule of a selected venue. By calling the relevant methods, the system retrieves venue details, including available time slots, capacity, and existing bookings, from the database.

After the user selects a time slot to schedule a new match or event. The system verifies the request by checking for conflicts in the database. If no conflict is detected, the schedule is updated with the new booking, and the user is notified of the successful operation. If a conflict is detected, the system notifies the user, and they are prompted to select a new time slot. Once the venue schedule is updated successfully, the overall league schedule is also synchronized with the new booking.

- Sequence Diagram:


SPORTS System
Database
Event Manager
The Venue Owner calls getVenueDetails(String VenueID) on the SPORTS System
The SPORTS System forwards the request to Venue.getVenueDetails(String venueID) in the Venue class
Return Schedule
Return Schedule
The Venue Owner selects a time slot and calls bookVenue(String matchID, date, time) on the SPORTS System
The SPORTS System forwards the request to Venue.bookVenue(String matchID, date, time) in the Venue class
Alternative
[If no conflict]
Return Ture
The Event Manager calls createSchedule(Match, Matches) on the SPORTS System to update the league schedule
The SPORTS System forwards the request to Schedule.createSchedule(Match, Matches) in the Schedule class
Return True
Retrurn Ture
[Else]
Return False
Return False
The Event Manager is prompted to select a different time slot or venue by re-calling bookVenue(String matchID, date, time) with new details
Retrurn True

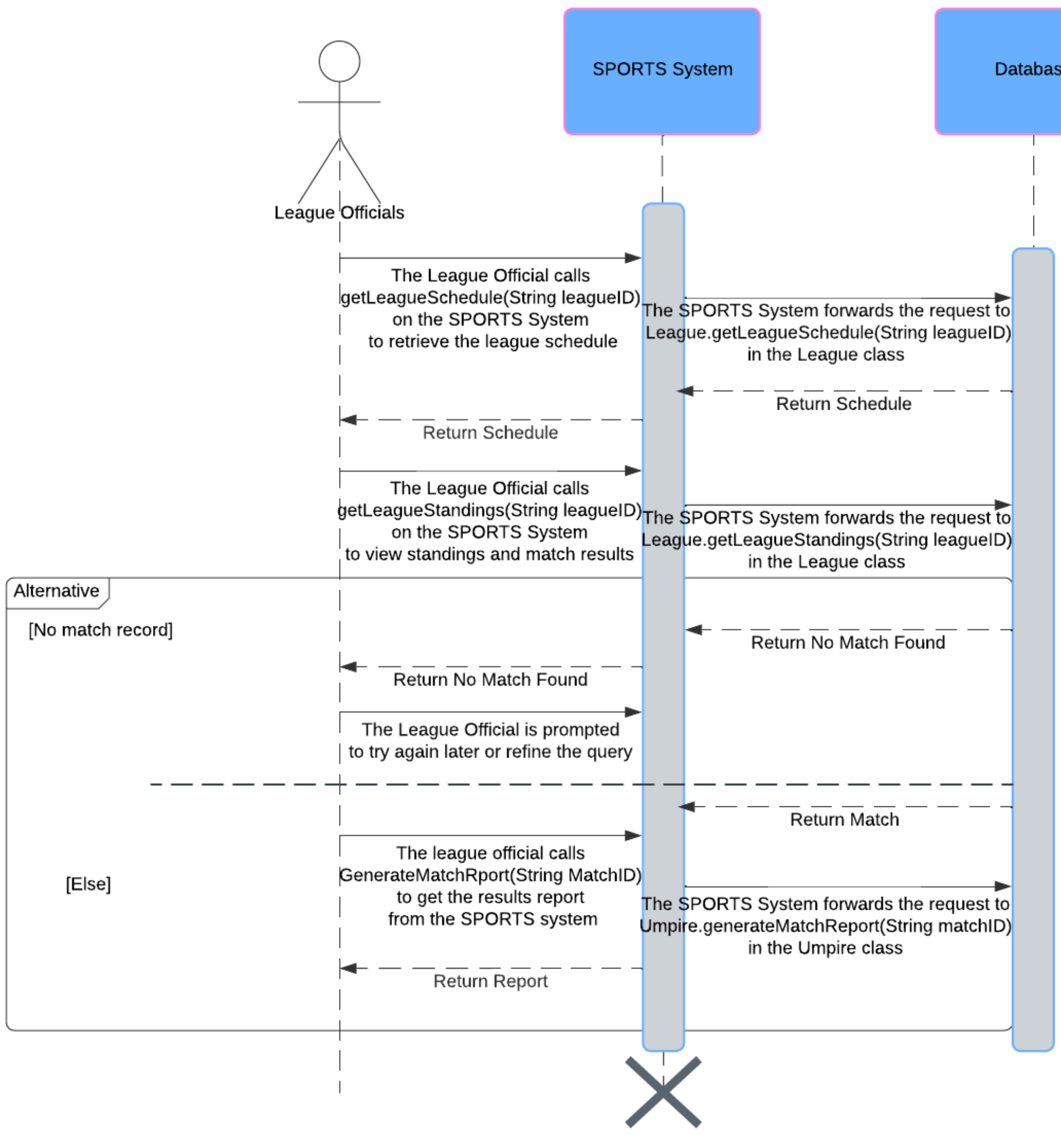
5.1.5 **Scenario Name: View League Schedule and Standings**

- Scenario Description:

  The League Official logs into the SPORTS System to monitor league schedules and standings. After authentication, they submit a request to view the league schedule, which includes match dates, venues, and teams. The SPORTS System queries the database and retrieves the schedule details, displaying them to the official.

Also, the League Official can view the current league standings by submitting a request. The system fetches the updated standings from the database, which reflect recent match results and team performances. If no match records or standings are found, the system notifies the official and prompts them to refine their query or try again later. Furthermore, the official can generate detailed match reports for specific matches, which include comprehensive statistics and summaries, aiding in decision-making and analysis.

- Sequence Diagram:


SPORTS System
Database
League Officials
The League Official calls getLeagueSchedule(String leagueID) on the SPORTS System to retrieve the league schedule
The SPORTS System forwards the request to League.getLeagueSchedule(String leagueID) in the League class
Return Schedule
Return Schedule
The League Official calls getLeagueStandings(String leagueID) on the SPORTS System to view standings and match results
The SPORTS System forwards the request to League.getLeagueStandings(String leagueID) in the League class
Alternative
[No match record]
Return No Match Found
Return No Match Found
The League Official is prompted to try again later or refine the query
Return Match
[Else]
The league official calls GenerateMatchRport(String MatchID) to get the results report from the SPORTS system
The SPORTS System forwards the request to Umpire.generateMatchReport(String matchID) in the Umpire class
Return Report

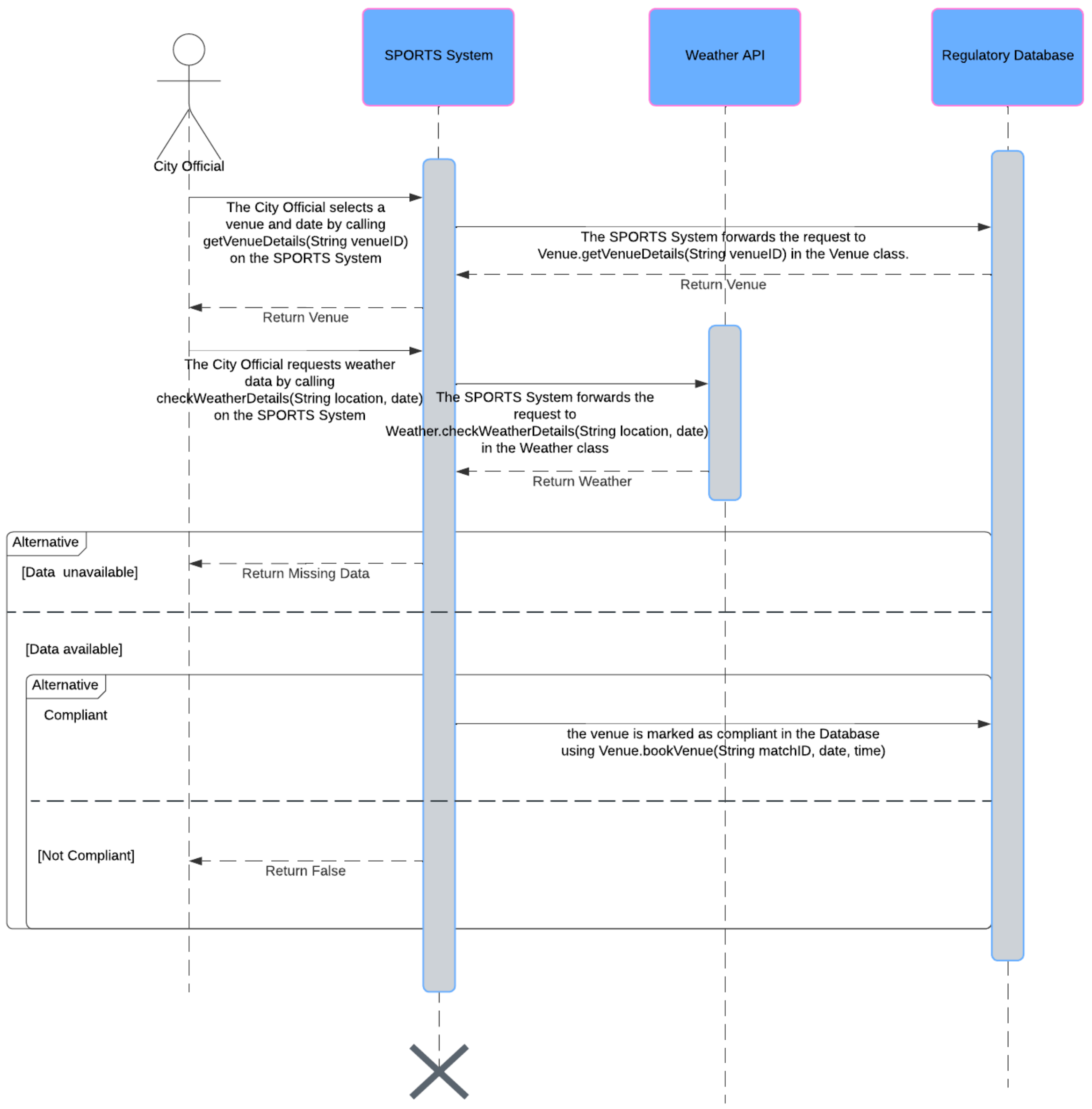
5.1.6 **Scenario Name: Monitor Venue Regulations and Weather**

- Scenario Description:

  The City Official logs into the SPORTS System to monitor compliance with venue regulations and weather conditions. They select a venue and date to begin the process. The system retrieves venue details, including availability and existing bookings, from the database. The City Official then requests weather information for the selected venue and date.

After the SPORTS System queries the database for weather data through the Weather class. If the weather data is unavailable, the system notifies the official and suggests retrying later. If weather data is retrieved, the system checks whether the venue complies with the regulatory requirements based on the weather conditions. If the venue is compliant, the system updates its status in the database and notifies the official. If not, the system provides detailed feedback to the official for further action.

- Sequence Diagram:


SPORTS System
Weather API
Regulatory Database
City Official
The City Official selects a venue and date by calling getVenueDetails(String venueID) on the SPORTS System
The SPORTS System forwards the request to Venue.getVenueDetails(String venueID) in the Venue class.
Return Venue
Return Venue
The City Official requests weather data by calling checkWeatherDetails(String location, date) on the SPORTS System
The SPORTS System forwards the request to Weather.checkWeatherDetails(String location, date) in the Weather class
Return Weather
Alternative
[Data unavailable]
Return Missing Data
[Data available]
Alternative
Compliant
the venue is marked as compliant in the Database using Venue.bookVenue(String matchID, date, time)
[Not Compliant]
Return False

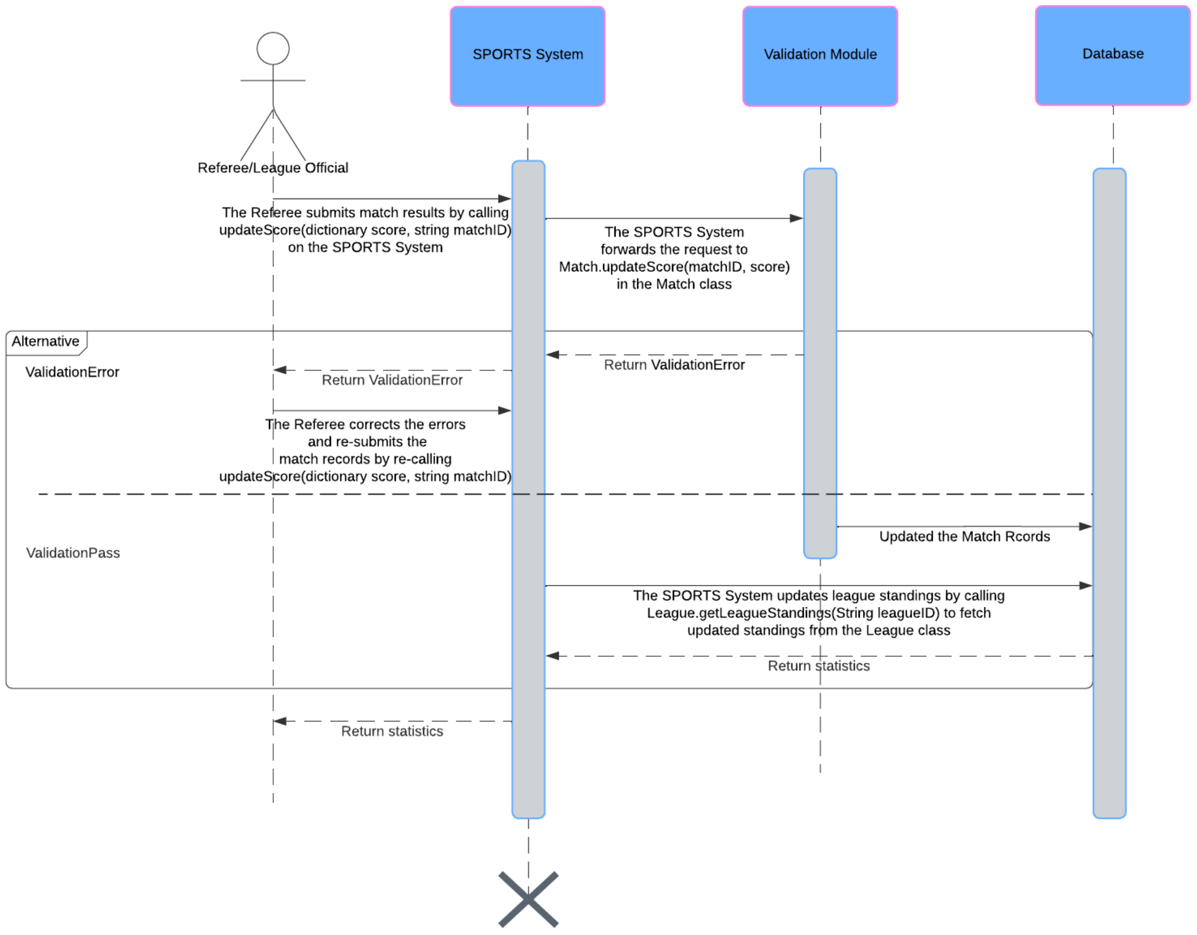
5.1.7 **Scenario Name: Maintain Match Records and Analyze Standings**

- Scenario Description:

  The Referee or League Official logs into the SPORTS System to manage match records. After successful authentication, they submit match results by entering scores and match IDs. The system forwards this data to the Match class for validation. If validation fails, the system notifies the Referee of errors and prompts them to correct and resubmit the match records. Once validation passes, the match records are stored in the database, and league standings are recalculated.

After the system then fetches the updated standings and displays them to the League Official. This process ensures the accuracy of match data and keeps the standings updated in real time. The League Official can also request detailed reports of specific matches for further analysis, enabling informed decision-making.

- Sequence Diagram:


SPORTS System
Validation Module
Database
Referee/League Official
The Referee submits match results by calling updateScore(dictionary score, string matchID) on the SPORTS System
The SPORTS System forwards the request to Match.updateScore(matchID, score) in the Match class
Alternative
ValidationError
Return ValidationError
Return ValidationError
The Referee corrects the errors and re-submits the match records by re-calling updateScore(dictionary score, string matchID)
ValidationPass
Updated the Match Rcords
The SPORTS System updates league standings by calling League.getLeagueStandings(String leagueID) to fetch updated standings from the League class
Return statistics
Return statistics

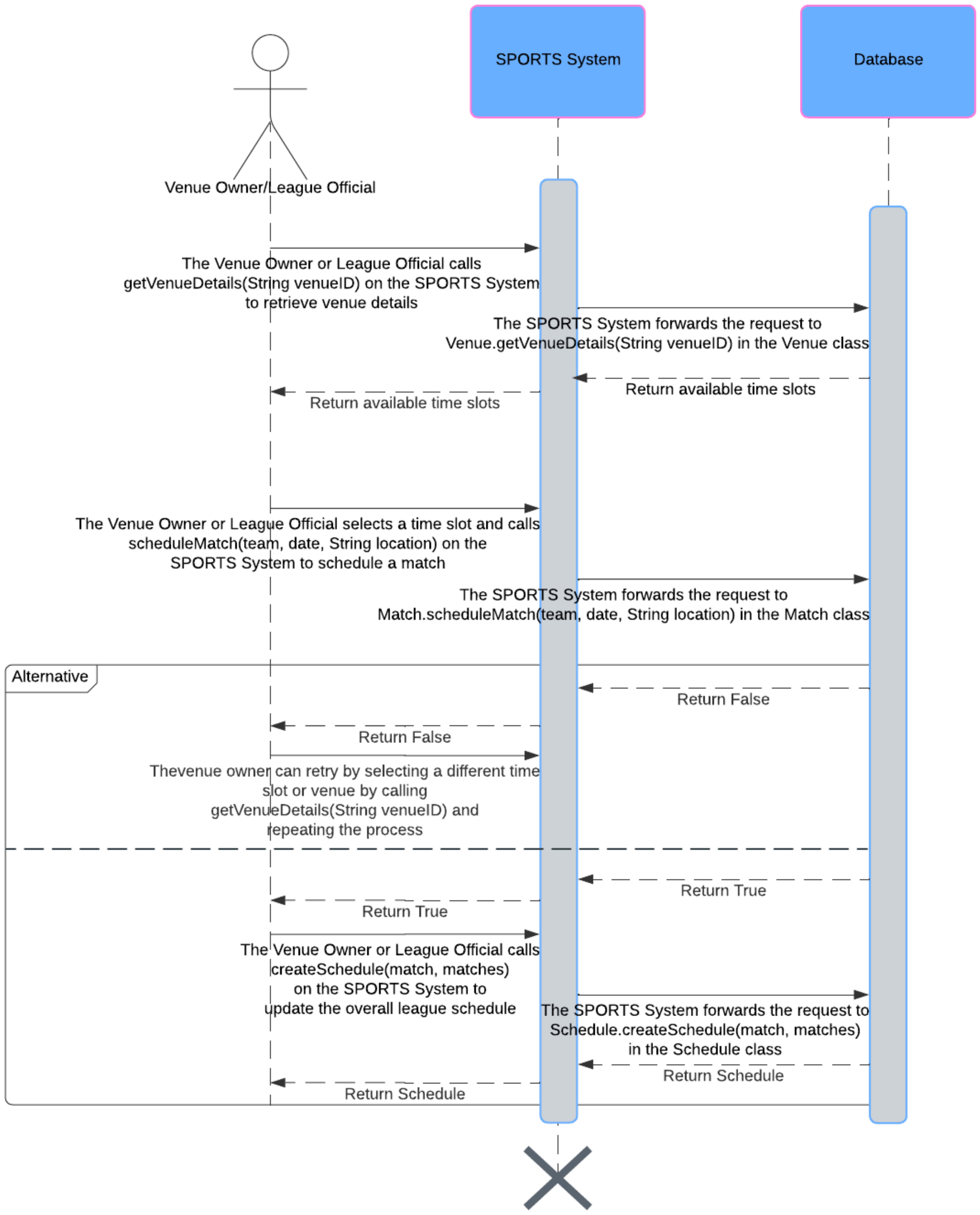
5.1.8 **Scenario Name: Scheduling League Matches at Venue**

- Scenario Description:

The Venue Owner or League Official logs into the SPORTS System to schedule league matches. They begin by selecting a venue and retrieving its details, including available time slots and existing bookings. Once the venue details are displayed, the user selects a time slot and schedules a match. The SPORTS System verifies the booking request by checking for conflicts in the database.

If a conflict is detected, the system notifies the user and prompts them to select a different time slot or venue. The process is repeated until a successful booking is made. After the match is scheduled, the system updates the overall league schedule to reflect the new match. The updated schedule is displayed to the user, ensuring they have a clear view of all scheduled activities.

- Sequence Diagram:


SPORTS System
Database
Venue Owner/League Official
The Venue Owner or League Official calls getVenueDetails(String venueID) on the SPORTS System to retrieve venue details
The SPORTS System forwards the request to Venue.getVenueDetails(String venueID) in the Venue class
Return available time slots
Return available time slots
The Venue Owner or League Official selects a time slot and calls scheduleMatch(team, date, String location) on the SPORTS System to schedule a match
The SPORTS System forwards the request to Match.scheduleMatch(team, date, String location) in the Match class
Alternative
Return False
Return False
Thevenue owner can retry by selecting a different time slot or venue by calling getVenueDetails(String venueID) and repeating the process
Return True
Return True
The Venue Owner or League Official calls createSchedule(match, matches) on the SPORTS System to update the overall league schedule
The SPORTS System forwards the request to Schedule.createSchedule(match, matches) in the Schedule class
Return Schedule
Return Schedule

6 Non-functional requirements

6.1 Performance Requirements

- **Requirement:** The system must handle up to 500 concurrent users with a response time under 2 seconds for key operations such as registration and schedule viewing.
- **Design Implementation:**
  - **Caching:** A Redis-based caching layer is implemented to reduce the load on the database for frequently accessed data, such as league schedules.
  - **Load Balancing:** NGINX is used as a reverse proxy to distribute incoming requests evenly across multiple backend servers.
  - **Efficient Queries:** Optimized SQL queries and indexed database fields minimize retrieval times.
- **Future Accommodation:** As the user base grows, the architecture supports horizontal scaling by adding more servers to the load balancer.

6.2 Security Requirements

- **Requirement:** Ensure data privacy and security for user information and payment details, complying with regulations like GDPR.
- **Design Implementation:**
  - **Encryption:** All sensitive data (e.g., passwords, payment information) is encrypted using AES for storage and transmitted using HTTPS.
  - **Authentication:** Role-based access control (RBAC) ensures users access only their authorized modules. Multi-factor authentication (MFA) is integrated for high-privilege users.
  - **Compliance Monitoring:** Regular audits and logging mechanisms ensure compliance with GDPR and other data protection standards.
- **Future Accommodation:** The design includes modular authentication components, allowing the integration of future security protocols like biometric authentication.

6.3 Usability Requirements

- **Requirement:** The system must provide an intuitive user interface that is responsive and accessible.
- **Design Implementation:**
  - **Responsive Design:** The frontend is built using React with a mobile-first approach, ensuring compatibility with a variety of devices.
  - **Accessibility Standards:** The interface follows WCAG 2.1 guidelines, supporting screen readers and high-contrast modes for better accessibility.
  - **User Feedback:** A feedback module allows users to report usability issues, enabling iterative improvements.

- **Future Accommodation:** The modular frontend design allows updates to the UI/UX without disrupting backend functionalities.

6.4 Scalability Requirements

- **Requirement:** The system must support scaling to accommodate additional users, leagues, and integrations.
- **Design Implementation:**
    - **Microservices Architecture:** Components like user management, scheduling, and payments are designed as independent services to ensure scalability.
    - **Database Partitioning:** Sharding techniques are employed in the database to distribute the data load efficiently as it grows.
- **Future Accommodation:** The system's API-driven architecture enables seamless integration of new modules or external systems, such as analytics tools or additional payment gateways.

6.5 Maintainability Requirements

- **Requirement:** The system must be easy to maintain and extend.
- **Design Implementation:**
    - **Code Modularity:** Following the SOLID principles, the codebase is divided into modules that are loosely coupled and highly cohesive.
    - **Documentation:** Comprehensive inline documentation and detailed API specifications make it easier for future developers to work on the system.
    - **Automated Testing:** Unit tests, integration tests, and end-to-end tests ensure robustness during updates.
- **Future Accommodation:** CI/CD pipelines are implemented to enable quick deployment of updates with minimal disruption.

6.6 Availability Requirements

- **Requirement:** The system must have an uptime of 99.9%, minimizing downtime for users.
- **Design Implementation:**
    - **Redundancy:** Critical components like databases and servers have redundant backups.
    - **Monitoring:** Tools like Prometheus and Grafana monitor system health in real time, allowing proactive issue resolution.
- **Future Accommodation:** Disaster recovery plans and auto-scaling features are incorporated to ensure high availability during peak loads or unexpected failures.

6.7 Anticipated System Evolution

- **Potential Changes:**
  - Increase in the number of users, leagues, or venues managed by the system.
  - Integration with advanced analytics platforms for performance insights.
  - Adapting to evolving security regulations and protocols.
- **Design Adaptation:** The system's modular architecture and microservices approach enable the seamless integration of new features and scalability to meet future demands. Frequent refactoring cycles ensure the system remains adaptable and efficient.

7 Requirements Traceability Matrix

Req No.	User	Player	Team Manager	League Manager	Payment	Umpire	Match	Team	League	Venue	City Officials	Weather
4.1.1	X	X										
4.1.2		X							X			
4.1.3		X							X			
4.1.4		X			X				X			
4.1.5		X			X				X			
4.2.1	X		X									
4.2.2			X						X			
4.2.3		X	X					X				
4.2.4			X		X							
4.2.5			X				X	X	X			
4.2.6		X	X					X				
4.3.1	X			X						X		
4.3.2	X			X						X		
4.3.3								X	X	X		
4.4.1							X	X	X			
4.5.1	X										X	
4.5.2									X	X	X	X

4.5.3										X	X	
4.6.1						X						
4.6.2	X							X	X			
4.6.3	X											
4.7.1				X				X	X			
4.7.2								X	X	X		
4.8.1		X	X				X	X				