Football Management System Documentation

Database Schema Overview

The **Database Schema Overview** provides a detailed explanation of the database structure, including entities, attributes, relationships, and business logic. This section is essential for understanding how data is organized and managed in the Football Management System.

## Entities and Attributes

### Lookup Tables

These tables store predefined values and their translations.

|  |  |  |
| --- | --- | --- |
| **Table** | **Attributes** | **Description** |
| **UserRole** | roleId, roleName, description, isAdmin | Defines user roles (e.g., Admin, Editor). |
| **GameStatus** | statusId, statusName, isActive | Defines game statuses (e.g., Scheduled, Live). |
| **CompetitionType** | typeId, typeName, fifaRecognized | Defines competition types (e.g., League, Cup). |
| **Season** | seasonId, seasonCode, startDate, endDate | Defines football seasons. |
| **PlayerPosition** | positionId, positionName, description | Defines player positions (e.g., Goalkeeper, Forward). |
| **CoachRole** | roleId, roleName, description | Defines coaching roles (e.g., Head Coach, Assistant Coach). |
| **StaffRole** | roleId, roleName, description | Defines non-coaching staff roles (e.g., Physio, Scout). |
| **LeagueFormat** | formatId, formatName, description | Defines league formats (e.g., League, Knockout). |

### **Core Entities**

These tables represent the core entities in the system.

|  |  |  |
| --- | --- | --- |
| **Table** | **Attributes** | **Description** |
| **User** | userId, username, email, passwordHash, roleId, isActive | Stores system users with authentication details. |
| **Address** | addressId, countryId, city, state, postalCode, latitude, longitude | Stores geographical locations for stadiums and clubs. |
| **Stadium** | stadiumId, countryId, officialName, fifaId, addressId, capacity | Stores details about football stadiums. |
| **Referee** | refereeId, firstName, lastName, licenseLevel, countryId, careerStart | Stores referees' professional details. |
| **Club** | clubId, countryId, addressId, officialName, shortName, founded | Stores details about football clubs. |
| **Person** | personId, countryId, firstName, lastName, dateOfBirth, gender | Stores personal details of individuals (players, staff, etc.). |

### Football Domain Entities

These tables manage teams, competitions, and related data.

|  |  |  |
| --- | --- | --- |
| **Table** | **Attributes** | **Description** |
| **Team** | teamId, clubId, teamName, seasonId, division | Represents teams participating in competitions. |
| **Competition** | competitionId, competitionName, typeId, seasonId, formatId | Represents football competitions. |

### Player Management

These tables manage players, contracts, and player statistics.

|  |  |  |
| --- | --- | --- |
| **Table** | **Attributes** | **Description** |
| **Player** | playerId, personId, positionId, height,  weight, preferredFoot | Stores detailed information about football players. |
| **Contract** | contractId, playerId, teamId, startDate,  endDate, weeklySalary | Manages player contracts. |
| **PlayerStatistics** | statisticId, playerId, gameId, goals, assists, yellowCards | Stores game-specific statistics for players. |

### Staff Management

These tables manage non-playing staff and their assignments.

|  |  |  |
| --- | --- | --- |
| **Table** | **Attributes** | **Description** |
| **Staff** | staffId, personId, roleId, countryId,  specialization, joinedDate | Stores information about non-playing staff. |
| **StaffAssignment** | assignmentId, staffId, roleId, clubId,   teamId, startDate, endDate | Tracks staff assignments to clubs or teams. |

### Game Management

These tables manage games, goals, substitutions, bookings, and events.

|  |  |  |
| --- | --- | --- |
| **Table** | **Attributes** | **Description** |
| **Game** | gameId, homeTeamId, awayTeamId, stadiumId, gameDateTime, statusId | Stores details of individual football games. |
| **Goal** | goalId, gameId, scorerId, assistId, minute, type | Stores details of goals scored in a game. |
| **Substitution** | substitutionId, gameId, playerInId, playerOutId, minute | Records player substitutions made during a game. |
| **Booking** | bookingId, gameId, playerId, minute, cardType, reason | Records bookings (yellow/red cards) issued to players. |
| **GameEvent** | eventId, gameId, minute, eventType, description | Logs various events occurring during a game (e.g., penalties, offsides). |

## Relationships

### One-to-Many Relationships

|  |  |  |
| --- | --- | --- |
| **Parent Table** | **Child Table** | **Description** |
| **UserRole** | **User** | One role can be assigned to many users. |
| **Country** | **Address** | One country can have many addresses. |
| **Club** | **Team** | One club can have many teams. |
| **Season** | **Team** | One season can have many teams. |
| **Player** | **Contract** | One player can have many contracts. |
| **Staff** | **StaffAssignment** | One staff member can have many assignments. |
| **Game** | **Goal** | One game can have many goals. |
| **Game** | **Substitution** | One game can have many substitutions. |
| **Game** | **Booking** | One game can have many bookings. |
| **Game** | **GameEvent** | One game can have many events. |

### Many-to-Many Relationships

|  |  |  |
| --- | --- | --- |
| **Table 1** | **Table 2** | **Description** |
| **Player** | **Team** | A player can belong to multiple teams over time (via contracts). |
| **Staff** | **Club/Team** | A staff member can be assigned to multiple clubs or teams over time. |

### One-to-One Relationships

|  |  |  |
| --- | --- | --- |
| **Table 1** | **Table 2** | **Description** |
| **Person** | **Player** | One person can be one player. |
| **Person** | **Staff** | One person can be one staff member. |

## Business Logic Enforcement

### Triggers

|  |  |
| --- | --- |
| **Trigger** | **Description** |
| trg\_referee\_careerStart\_validation | Ensures the referee's career start date is valid. |
| trg\_club\_founded\_validation | Ensures the club's founded year is valid. |
| trg\_competition\_date\_validation | Ensures the competition's end date is after the start date. |
| trg\_player\_dob\_validation | Ensures the player's date of birth is valid. |
| trg\_contract\_overlap | Prevents overlapping contracts for the same player and team. |
| trg\_staff\_dob\_validation | Ensures the staff member's date of birth is valid. |
| trg\_staff\_assignment\_overlap | Prevents overlapping assignments for the same staff member and role. |
| trg\_game\_attendance\_check | Ensures game attendance does not exceed stadium capacity. |

## Performance Optimization

### Indexes

|  |  |
| --- | --- |
| **Index** | **Description** |
| idx\_user\_email | Optimizes email lookups in the User table. |
| idx\_stadium\_fifaId | Optimizes FIFA stadium code lookups. |
| idx\_club\_fifaId | Optimizes FIFA club code lookups. |
| idx\_person\_names | Optimizes searches by last name and first name in the Person table. |
| idx\_team\_season | Optimizes season-based team queries. |
| idx\_competition\_season | Optimizes season-based competition queries. |
| idx\_player\_fifaId | Optimizes FIFA player ID lookups. |
| idx\_player\_positionId | Optimizes position-based player queries. |
| idx\_contract\_player\_team\_dates | Optimizes contract queries by player, team, and date ranges. |
| idx\_country\_name | Optimizes country name lookups. |
| idx\_staff\_assignment\_roleId | Optimizes role-based staff assignment queries. |
| idx\_game\_statusId | Optimizes status-based game queries. |
| idx\_competition\_typeId | Optimizes type-based competition queries. |

# General Folder Structure of the System

The **General Folder Structure** provides a detailed explanation of the system's organization, including backend, frontend, and supporting files. This section is essential for understanding the project's architecture, the roles of each folder and file, and how they work together to create a cohesive application.

## Backend

The backend is the server-side component of the application, built using **Node.js** and **Express.js**, with **Sequelize** as the ORM for database management. It handles data storage, business logic, and API communication.

### Folder Structure

|  |  |
| --- | --- |
| **Folder** | **Description** |
| **config/** | Contains configuration files for the database, authentication, and environment variables. |
| **models/** | Defines database models using Sequelize ORM. Each model corresponds to a database table. |
| **repositories/** | Implements the data access layer. Contains files for querying the database. |
| **services/** | Implements business logic. Contains files for processing data and handling application logic. |
| **controllers/** | Handles API requests. Contains files for managing HTTP requests and responses. |
| **routes/** | Defines API routes. Each file corresponds to a specific entity (e.g., user, club). |
| **migrations/** | Contains database migration files for creating and modifying database tables. |
| **seeders/** | Contains database seeder files for populating the database with initial data. |
| **validations/** | Contains input validation logic for API requests. |
| **middleware/** | Contains custom middleware for authentication, error handling, and more. |
| **server.js** | The main entry point for the backend application. |
| **.env** | Stores environment variables (e.g., database credentials, API keys). |
| **.env.example** | A template for the .env file. |
| **package.json** | Lists backend dependencies and scripts. |
| **README.md** | Provides documentation for the backend. |
| **.gitignore** | Specifies files and folders to ignore in version control. |

### Key Files

|  |  |
| --- | --- |
| **File** | **Description** |
| **db.config.js** | Configures the database connection. |
| **auth.config.js** | Configures authentication settings (e.g., JWT secret). |
| **env.config.js** | Loads and validates environment variables. |
| **User.js** | Defines the User model for the User table. |
| **userRepository.js** | Implements database queries for the User table. |
| **userService.js** | Implements business logic for user-related operations. |
| **userController.js** | Handles HTTP requests for user-related endpoints. |
| **userRoutes.js** | Defines API routes for user-related endpoints. |
| **authMiddleware.js** | Implements authentication middleware for securing routes. |
| **errorHandler.js** | Implements error handling middleware. |

### How It Works Together

1. **Request Flow**:
   * A client (frontend) sends an HTTP request to the backend.
   * The request is routed to the appropriate controller via the routes/ folder.
   * The controller calls the corresponding service in the services/ folder.
   * The service interacts with the database through the repository in the repositories/ folder.
   * The repository uses the model in the models/ folder to query the database.
   * The response is sent back to the client.
2. **Middleware**:
   * Middleware in the middleware/ folder handles tasks like authentication and error handling.
   * For example, authMiddleware.js ensures that only authenticated users can access certain routes.
3. **Database Management**:
   * The migrations/ folder contains scripts for creating and modifying database tables.
   * The seeders/ folder contains scripts for populating the database with initial data.

## Frontend

The frontend is the client-side component of the application, built using **React.js**. It provides the user interface and interacts with the backend via API calls.

### Folder Structure

|  |  |
| --- | --- |
| **Folder** | **Description** |
| **admin/** | Contains the admin dashboard application. |
| **user/** | Contains the user-facing application. |
| **components/** | Contains reusable UI components (e.g., buttons, forms). |
| **pages/** | Contains page components (e.g., Home, Dashboard). |
| **services/** | Contains files for making API calls to the backend. |
| **hooks/** | Contains custom React hooks for reusable logic. |
| **context/** | Contains React context for global state management. |
| **styles/** | Contains custom styles (e.g., CSS, SCSS). |
| **App.js** | The main component for the application. |
| **index.js** | The entry point for the application. |
| **.env** | Stores environment variables (e.g., API base URL). |
| **package.json** | Lists frontend dependencies and scripts. |
| **README.md** | Provides documentation for the frontend. |
| **.gitignore** | Specifies files and folders to ignore in version control. |

### Key Files

|  |  |
| --- | --- |
| **File** | **Description** |
| **App.js** | The root component for the application. |
| **index.js** | Renders the root component into the DOM. |
| **userService.js** | Implements API calls for user-related operations. |
| **authContext.js** | Manages authentication state using React context. |
| **Home.js** | A page component for the home page. |
| **Button.js** | A reusable UI component for buttons. |

### How It Works Together

1. **Component Hierarchy**:
   * The App.js file is the root component that renders other components.
   * The pages/ folder contains page components (e.g., Home.js, Dashboard.js).
   * The components/ folder contains reusable UI components (e.g., Button.js, Form.js).
2. **State Management**:
   * The context/ folder contains React context for managing global state (e.g., user authentication).
   * For example, authContext.js manages the user's authentication state.
3. **API Communication**:
   * The services/ folder contains files for making API calls to the backend.
   * For example, userService.js handles API calls for user-related operations.
4. **Styling**:
   * The styles/ folder contains custom styles (e.g., CSS, SCSS) for the application.

## Supporting Files

The supporting files include documentation, scripts, tests, assets, and licenses. These files are essential for deployment, testing, and maintaining the project.

### Folder Structure

|  |  |
| --- | --- |
| **Folder** | **Description** |
| **docs/** | Contains project documentation (e.g., API specs, architecture diagrams). |
| **scripts/** | Contains deployment and automation scripts (e.g., deploy.sh, backup-db.sh). |
| **tests/** | Contains unit, integration, and end-to-end tests. |
| **assets/** | Contains static assets (e.g., images, fonts, icons). |
| **build/** | Contains build outputs for deployment. |
| **licenses/** | Contains license files (e.g., LICENSE, NOTICE). |

### Key Files

|  |  |
| --- | --- |
| **File** | **Description** |
| **api-spec.md** | Provides API documentation using Swagger/OpenAPI. |
| **database-schema.png** | A diagram of the database schema. |
| **deploy.sh** | Automates the deployment process. |
| **backup-db.sh** | Automates database backups. |
| **setup-env.sh** | Sets up the development environment. |
| **user.test.js** | Contains unit tests for user-related functionality. |
| **logo.png** | The project logo. |
| **LICENSE** | The project license (e.g., Apache License 2.0). |

## Tech Stack

The Football Management System uses the following technologies:

**Backend**

* **Node.js**: A JavaScript runtime for building scalable server-side applications.
* **Express.js**: A web framework for building RESTful APIs.
* **Sequelize**: An ORM for managing database interactions.
* **MySQL**: The relational database for storing application data.
* **JWT**: JSON Web Tokens for authentication and authorization.

**Frontend**

* **React.js**: A JavaScript library for building user interfaces.
* **React Router**: For handling client-side routing.
* **Axios**: For making HTTP requests to the backend.
* **Context API**: For global state management.
* **CSS/SCSS**: For styling the application.

**DevOps**

* **Docker**: For containerizing the application.
* **Docker Compose**: For managing multi-container Docker applications.
* **Git**: For version control.
* **CI/CD**: For automated testing and deployment.

## Why This Structure?

1. **Modularity**: The structure separates concerns into distinct folders (e.g., models, services, controllers), making the codebase easier to maintain and scale.
2. **Reusability**: Reusable components (e.g., UI components, hooks) reduce code duplication and improve consistency.
3. **Separation of Concerns**: Backend and frontend are separated, allowing developers to work on each part independently.
4. **Scalability**: The structure supports adding new features without disrupting existing functionality.
5. **Documentation**: The docs folder ensures that the project is well-documented for future developers.

## How Everything Connects

1. **Frontend-Backend Communication**:
   * The frontend makes API calls to the backend using Axios.
   * The backend processes these requests, interacts with the database, and sends back responses.
2. **Database Interaction**:
   * The backend uses Sequelize models to interact with the MySQL database.
   * Migrations and seeders ensure the database is properly set up and populated.
3. **State Management**:
   * The frontend uses React context to manage global state (e.g., user authentication).
   * The backend uses JWT for authentication and authorization.
4. **Deployment**:
   * Docker and Docker Compose are used to containerize the application for deployment.
   * CI/CD pipelines automate testing and deployment processes.

This **General Folder Structure** section provides a clear and organized overview of the system's architecture. It is designed to be easy to understand and can be directly imported into a Word document. Let me know if you need further assistance!