UI for GoTeam Application

* **Address:** <http://localhost:8089/html/index.html>

This is the entry point for GoTeam application when running locally. The application is hosted on port 8089, and the user starts with the ‘index.html’ page to interact with various features.

* **Framework Choice:**
* **Frontend:** I opted for a straightforward and effective combination of **HTML** for the markup and **JavaScript** for the dynamic aspects and logic. This choice allows for direct interaction with RESTful endpoints without the need for an additional library or framework.
* **Backend:** I have chosen **Java** as the backend technology, using **Spring Boot** framework, which excels at creating web services and interacting with databases.
* **Features and Functionalities:**

1. **Index Page (index.html, index.js):**

The main page has a dropdown menu listing all the available actions the user can perform.

1. **Create Players Page (CreatePlayers.html, CreatePlayers.js):**
2. **Create a Player:** Users can create an individual player with attributes like player name, jersey number, team name, total goals, total assists through an input form. On form submission, a **POST** request ***‘/players/createPlayer’*** is made to an endpoint to save this player in the database.
3. **Create Multiple Players:** There is a bulk entry option, where users can input or upload data for multiple players at once using JavaScript format. On form submission, a **POST** request ***‘/players/createManyPlayer’*** is made to an endpoint to save these players in the database.
4. **Get Players Page (GetPlayers.html, GetPlayers.js):**
5. **Get All Players:** A **GET** request ***‘/players/getAllPlayers’*** to an endpoint fetches a list of all players in the system. The returned data is then displayed in a tabular format.
6. **Get Player by ID:** Users can retrieve data for a specific player by inputting the player’s ID. A **GET** request ***‘/players/getPlayerById/${playerId}’*** is made fetching and displaying this player’s data.
7. **Get Players by Team:** By specifying a team name, users can retrieve a list of all players from that team. A **GET** request ***‘/players/getPlayersByTeam/${teamName}’*** is made to serve this purpose.
8. **Update Players Page (UpdatePlayers.html, UpdatePlayers.js):** Users can modify the details of a specific player by inputting their ID and the updated data. A **PUT** request ***‘/players/updatePlayer/${playerId}’*** handles the update process.
9. **Delete Players Page (DeletePlayers.html, DeletePlayers.js):**
10. **Delete All Players:** A feature that allows users to delete all players with a single action. This triggers a **DELETE** request to the ***‘/players/deleteAllPlayers’*** endpoint.
11. **Delete Player By ID:** Users can remove a specific player by specifying their ID, which sends a **DELETE** request to ***‘/players/deletePlayerById/${playerId}’.***
12. **Delete Players by Team:** All players from a particular team can be removed by specifying the team’s name. This sends a **DELETE** request ***‘/players/deletePlayersByTeam/${teamName}’*** endpoint.
13. **Count Players Page (CountPlayers.html, CountPlayers.js):**
14. **Count All Players:** Users can get a total count of all players in the system. A **GET** request ***‘/players/countAllPlayers’*** returns this count.
15. **Count Players in A Team:** By specifying a team name, users can get a count of all players from that team. A **GET** request to ***‘/players/countPlayersInATeam/${teamName}’*** serves this purpose.
16. **Top Goal Scorer Page (TopGoalScorer.html, TopGoalScorer.js):** Users can find out who the top goal scorer is in a particular category i.e., for a certain badge. A **GET** request to ***‘/players/topGoalScorer/${badgeAwarded}’*** returns the top goal scorer for the given badge.

* **Object-Oriented Design Principles:**

1. **Encapsulation:**
2. **Frontend:** The separation of HTML, CSS, and JS files ensures that the structure, presentation, and behaviour are encapsulated in their respective layers. This modular approach ensures that a change in one layer doesn't impact the others directly.
3. **Backend:** Classes and methods encapsulate specific functionalities. For instance, each REST controller method handles a distinct HTTP request. The logic associated with data retrieval or modification is kept inside these methods, preventing external entities from directly accessing the internal workings.
4. **Polymorphism:** Through method overriding in the backend or dynamically rendering components in the frontend based on user input, the application can decide at runtime how to respond, embodying polymorphism.
5. **Abstraction:**
6. **Frontend:** Different pages for different actions mean that the user doesn't need to understand the intricacies of how each action is performed. They interact with an abstracted interface (buttons, dropdowns), and the underlying code handles the complexities.
7. **Backend:** The usage of interfaces or abstract classes in the service layer ensures that the concrete implementation details are hidden, and only essential functionalities are exposed.
8. **DRY:** Reusable CSS classes mean that styles aren't repeatedly defined. JS functions that are used across different actions can be defined once and called wherever needed.
9. **Meaningful Naming and Documentation:** Using meaningful naming conventions helps developers understand the purpose and usage of classes, methods, variables, and other entities. Moreover, well-commented code aids in clarity, ensuring that the application is maintainable and scalable in the long run.
10. **Feedback to Users:** Using messages from the ResponseEntity in the backend to inform users about the result of their actions is another layer of abstraction. Instead of revealing intricate details about database operations or possible errors, the application gives clean, understandable feedback.