UML DIAGRAMS

1. **Class Diagrams**
   * + **Data Entity:** Player

This class diagram provides a visual representation of the ‘Player’ class and its attributes, including their data types. It doesn’t include methods or relationships with other classes, as it primarily focuses on the attributes of the ‘Player’ entity.

* + - **Repository:** PlayerRepository

The primary purpose of the ‘PlayerRepository’ interface is to provide basic data access operations for the ‘Player’ entity. It doesn’t show additional custom methods.

* + - **Controller:** PlayerController

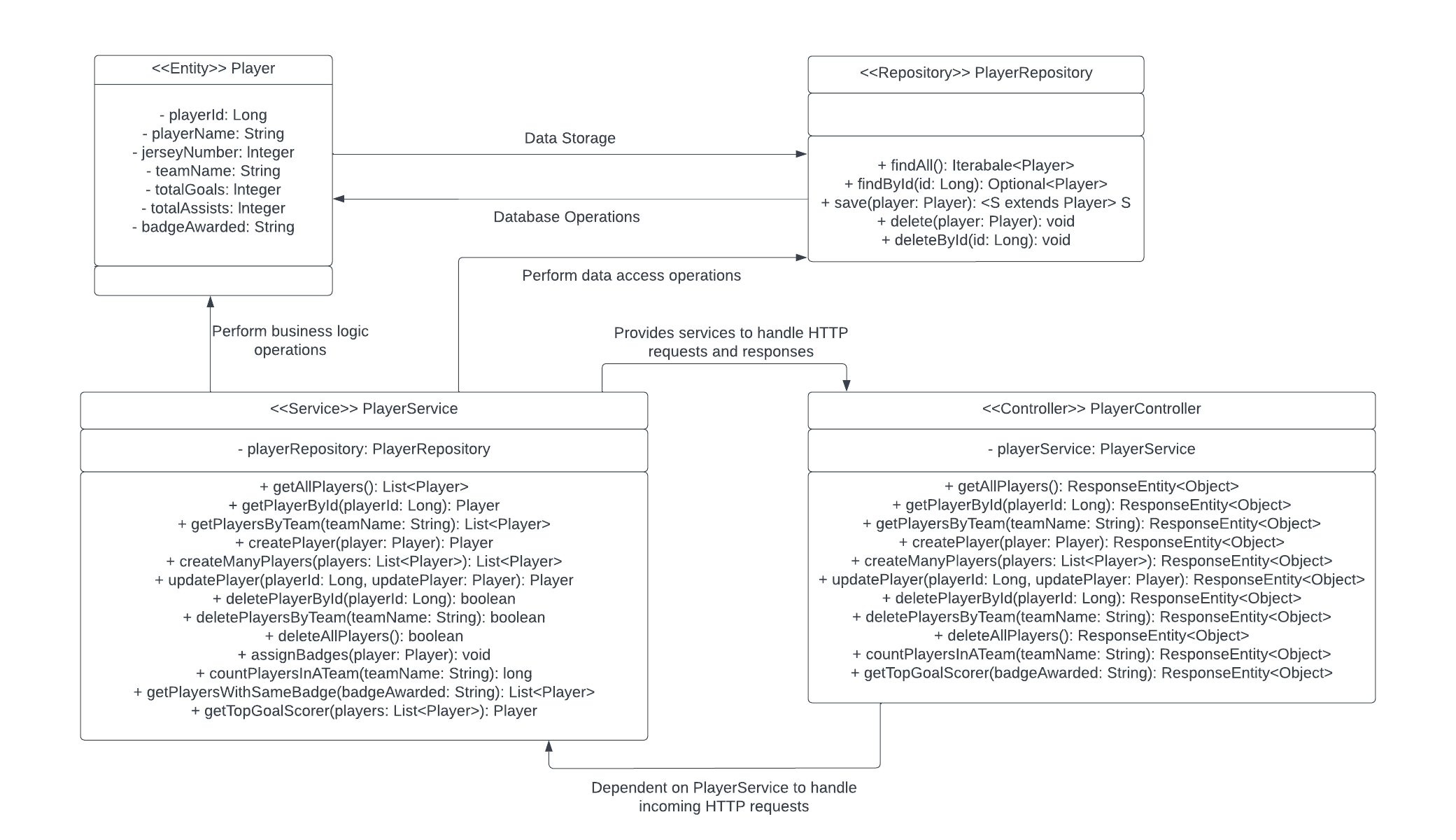
This diagram indicates a dependency on the ‘PlayerService’ class. It includes various methods that map to HTTP requests to handle player-related operations.

* + - **Service:** PlayerService

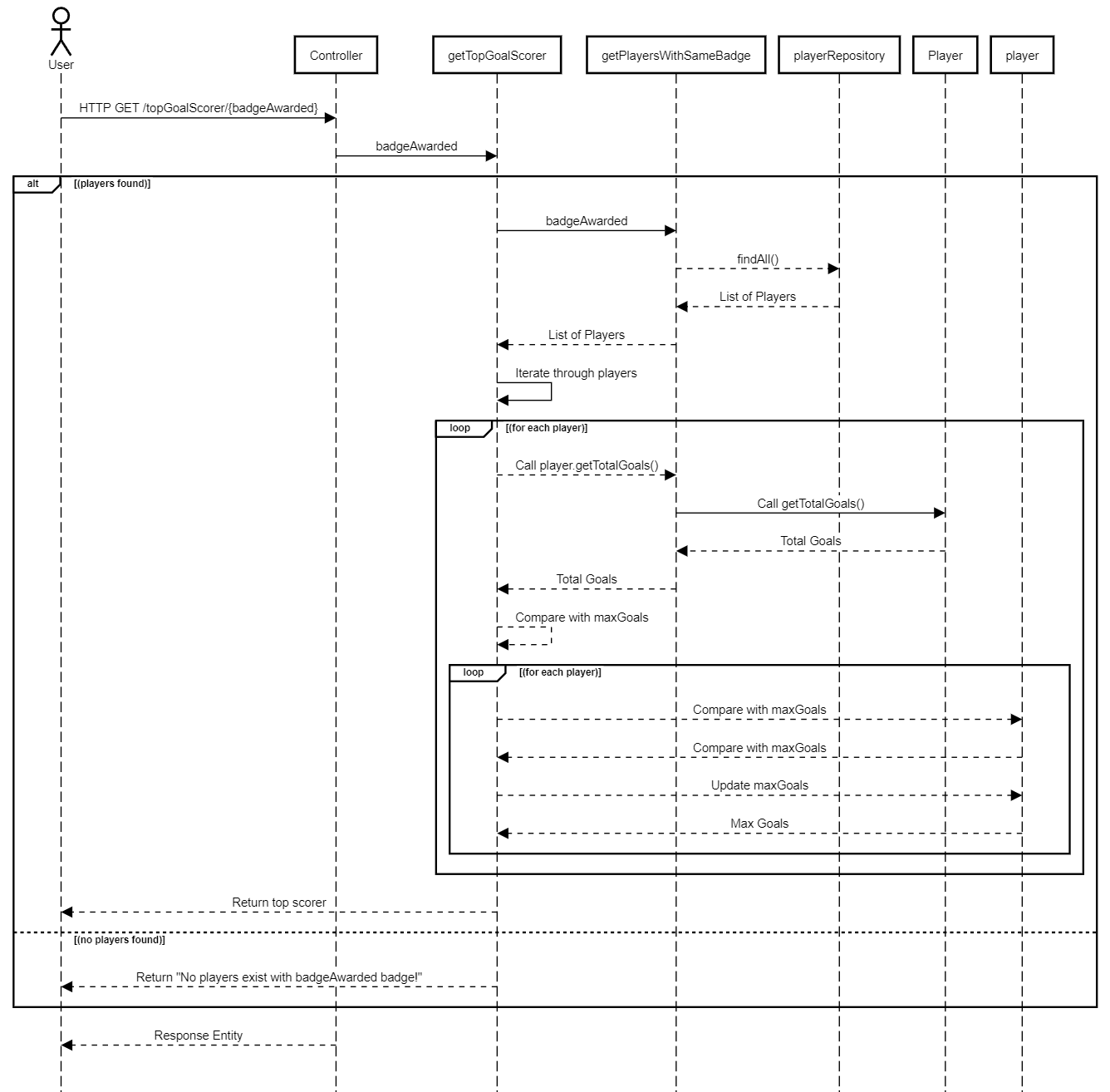
This diagram contains a reference to the ‘PlayerRepository’ for performing data access operations. It includes methods for various functionalities such as retrieving players, and additional methods like assignBadges to perform specific business logic operations.

Here are all the relationships between the components of the application:

1. **Player to PlayerRepository:** This is an association where the Player interacts with PlayerRepository for data storage.
2. **PlayerRepository to Player:** This is an association where the PlayerRepository interacts with the Player to perform database operations. PlayerRepository is responsible for managing the persistence of Player objects.
3. **PlayerController to PlayerService:** PlayerController has a dependency on PlayerService, which means it uses the services provided by the PlayerService to handle incoming HTTP requests and perform player-related operations.
4. **PlayerService to PlayerRepository:** PlayerService interacts with PlayerRepository to perform data access operations. It uses the repository to retrieve, save, update, and delete Player entities from the database.
5. **PlayerService to Player:** PlayerService performs various business logic operations on Player objects, such as assigning badges, counting players, and calculating the top goal scorer.
6. **PlayerService to PlayerController:** PlayerService provides services to PlayerController to handle HTTP requests and responses. PlayerController relies on PlayerService to perform player-related operations.



1. **Sequence Diagrams**
2. **getTopGoalScorer()**

****

**Actors:** User – initiates the HTTP GET request to find the top goal scorer with a specific badge.

**Participants:**

* + - * PlayerController – Represents the controller class that handles the incoming HTTP request.
      * PlayerService –

getTopGoalScorer – Represents the function responsible for finding the top goal scorer.

getPlayersWithSameBadge – Represents the function responsible for retrieving players with the same badge.

* + - * PlayerRepository – Respresents the repository or data access component responsible for database operations.
      * Player – Represents the Player class.

**Sequence of Events:**

* + - * The User actor sends an HTTP GET request to the Controller with the URl **/topGoalScorer/{badgeAwarded}**
      * The Controller forwards the **badgeAwarded** parameter to the **getTopGoalScorer** function.
      * If players with the specified badge are found, the following sequence of events occurs:

**getTopGoalScorer** calls **getPlayersWithSameBadge** with the **badgeAwarded.**

**getPlayersWithSameBadge** interacts with **playerRepository** to retrieve a list of players.

The list of players is then passed back to **getTopGoalScorer.**

**getTopGoalScorer** iterates through the list of players, comparing their total goals to find the top scorer.

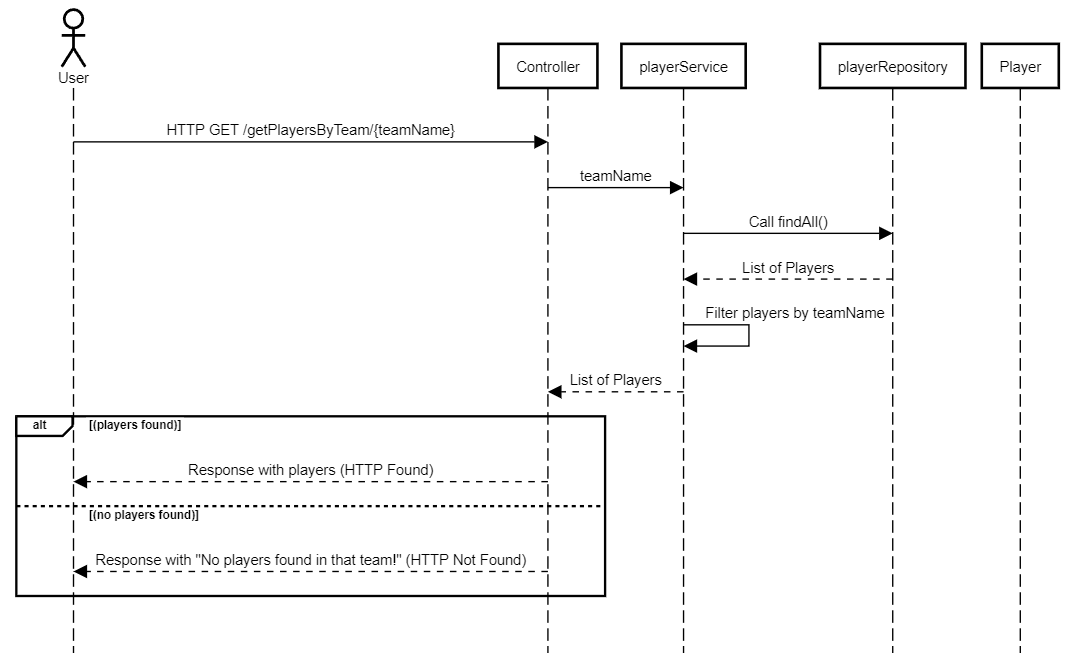
The top scorer is returned to the Controller, and the result is sent back to the User actor.

* + - * If no players with the specified badge are found, a “No players exist with **{badgeAwarded}** badge!” message is returned to the User actor.

**Loops and Conditions:**

* + - * The diagram uses the **alt** and **else** syntax to represent conditional branching based on whether players with the specified badge are found.
      * Withing the loops, the diagram shows the iterative process of comparing player goals and updating the top scorer as the loop progresses.

1. **getPlayersByTeam()**

****

**Actors:** User – initiates the HTTP GET request to retrieve players by team name.

**Participants:**

* + - * PlayerController – Represents the controller class that handles the incoming HTTP request.
      * PlayerService – Represents the service layer responsible for business logic.
      * PlayerRepository – Respresents the repository or data access component responsible for database operations.
      * Player – Represents the Player class.

**Sequence of Events:**

* + - * The User actor sends an HTTP GET request to the Controller with the URl **/getPlayersByTeam/{teamName}**
      * The Controller forwards the **teamName** parameter to the **playerService.**
      * The **playerService** calls **playerRepository** to retrieve a list of all players from the database.
      * The retrieved list of players is filtered by **playerService**  to include only players whose team name matches the provided **teamName.**
      * The filtered list of players is returned to the Controller.
      * The Controller checks if players are found and sends an appropriate response to the User actor:

If players are found, if responds with the list of players (HTTP Found)**.**

If no players are found, it responds with a “No players found in that team!” message (HTTP Not Found).

* + - * If no players with the specified badge are found, a “No players exist with **{badgeAwarded}** badge!” message is returned to the User actor.