SLAM BLaMo Design Specification

Authors:

Oskar Alyn

Michael Nguyen

Ajay Manicka

Zane Meyer

Group: 5

Table of Contents:

Revision History:

Date	Version	Description	Author
11/17/2021	v1.0	Create Document	Oskar Alyn
11/30/2021	v2.0	Merge in other docs, and add assignment 6 requirements	Group 5
12/12/2021	v3.0	Add assignment 7 requirements	Group 5

1. Introduction

1.1. Purpose

This design document outlines the Sports League Administration Manager system's Business Layer Module, designed for the Minneapolis Parks and Recreation department. The design is intentionally high level in order to provide understanding of the system as a whole, but is intended to have enough detail for a reasonable programmer to be able to implement this system. Our design used interfaces for the BlaMo request layer because we needed to have a well-defined set of functions that the UI component had access to, and needed the ability to send requests to the classes in the rest of the BLaMo system. We also used an abstract class to represent the User because they are an effective way of identifying unifying functions across multiple classes. Since we didn't have one general user, but there were several subtypes of users that all had a set of base methods, the abstract class was perfectly suited for our purposes. We also used two main different types of data stores. One database that can persist and retrieve data in the form of objects, and two other smaller data stores that are implemented with json files.

1.2. References

SLAM Requirements Document from Assignment 3
SLAM Conceptual Model from Assignment 2

2. Glossary

This glossary contains all acronyms and terms used in this document.

SLAM = Sports League Administration Manager - A software application for the Municipal Parks and Recreation Board that allows the given stakeholders to complete their tasks.
BLaMo = Business Layer Module - Layer in software that receives and processes UI requests and then relies on the infrastructure layer.
DB = Database - A collection of data stored on a computer system that is organized and can be accessed.
DataStore - a method for storing persistent data

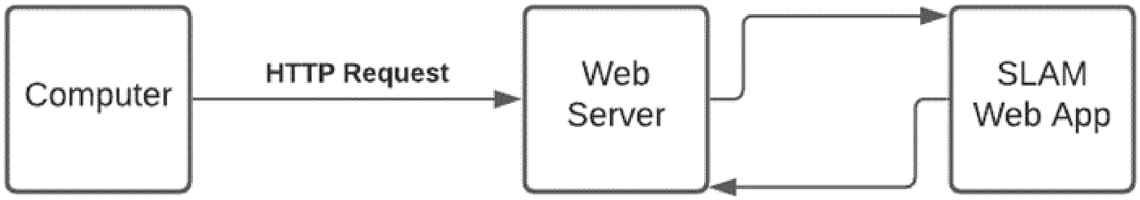
3. Design Overview

3.1. Introduction

We adopt an Object-Oriented design approach for creating the class diagram for BLaMo. This design features the addition of interfaces the UI submodule can use to interact with BLaMo modules. It also creates several Data Stores the Infrastructure layer can interact with to maintain security and other pertinent details over information entered on the SLAM system. Moreover, we adopt a layered architecture for facilitating user requests to access and change data stored on BLaMo's databases. Both designs are visualized using LucidApp.

3.2. Environment Overview

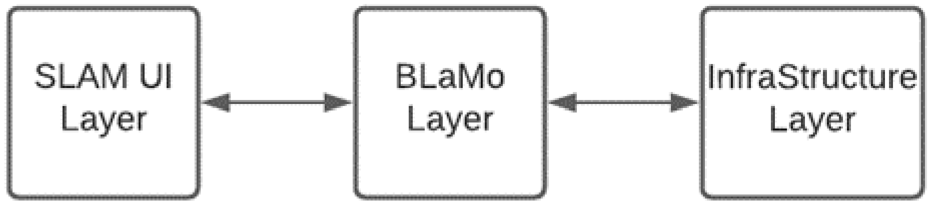
The SLAM system will be run in a stand-alone web application environment. The people who use this software are generally people involved in Leagues at a Park and Rec center and their friends and family. The system is not meant to interfere largely with a user's personal data and is therefore designed to capture essential data from the user including name and email. This leads to the very simple environmental network diagram for the SLAM system and its environment (user's computer) trying to access the software. The SLAM software is relatively straightforward and encapsulates all its databases within its infrastructure layer. Therefore, there is no external database interfacing required in this design proposal which leads to a simple environmental model.



3.3. System Architecture

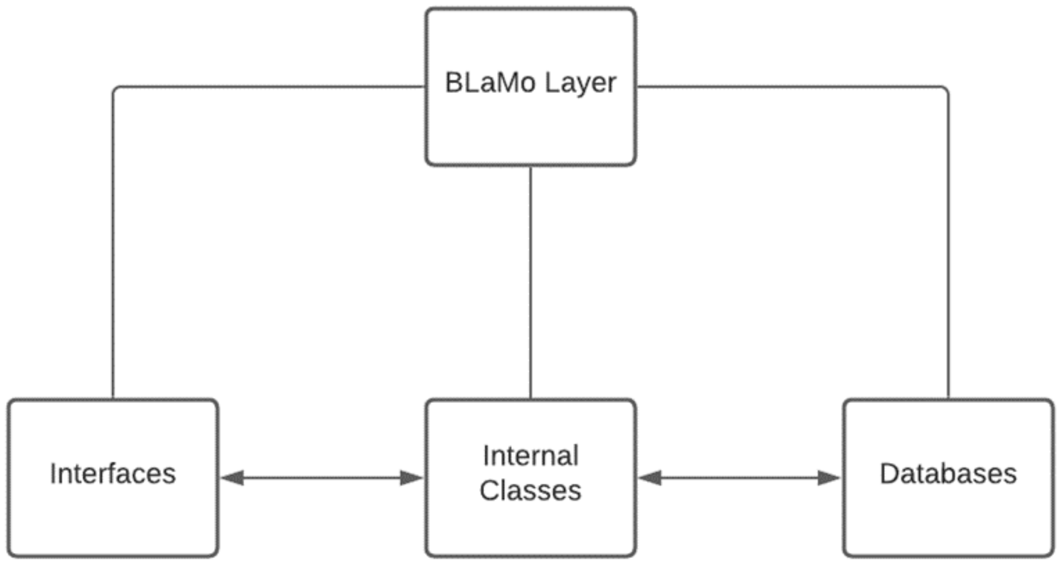
This section provides a high-level overview of the SLAM software and then describes the setup within the BLaMo

3.3.1. Top-level system structure of SLAM system



The system consists of three major layers which all interact with each other. The SLAM UI layer is the only layer the user can interact with, but that layer accesses modules in BLaMo which may affect databases which the Infrastructure Layer handles. This is a top-down architecture where the UI is the top layer and the Infrastructure layer is at the bottom with BLaMo in the middle.

3.3.2. BLaMo Sub-system



The BLaMo sub-system has three components: Interfaces, Internal Classes, and Databases. The interfaces allow the UI to use modules within the BLaMo internal classes without directly accessing those methods. Operations within these classes may result in data addition or loss which must be updated. This is done by populating databases which directly interact with the library interfaces within the Internal Classes substructure. The Internal Classes capture all the major functionality of SLAM including league, team, and user management.

3.4. Interfaces and Data Stores

3.4.1. System Interfaces

The system will be providing four high level interfaces for the UI to use to use the BLaMo submodules. They are separated by type of user to encapsulate each command more easily by the type of user who may use it. There are several methods which are repeated between different interfaces, and this is done to ensure that command can be executed by multiple different roles.

3.4.1.1. General Interface

This interface is meant for users who are not registered as a member of any events on SLAM to view public League Information. This was made to prevent attackers from the public from being able to access control of commands which can ruin the structure of the league. The interface is implemented primarily by the public and parents, who can view league information and standings, while updating their own account information.

3.4.1.2. Staff Interface

This interface allows staff members to manipulate records, pay officials, manage leagues, and create matches directly from the UI. They will log in through the UI layer and will use this interface to perform their desired actions within the SLAM system.

3.4.1.3. Coach Interface

This interface helps coaches more closely manage their teams without the overhead of navigating through different pages meant for different roles. They can *directly manage their team, manage players, manage league payment, and even schedule matches directly through the interface.*

3.4.1.4. Official Interface

The Official Interface is used by Officials to sign up to officiate a match and submit scores after they have concluded.

3.4.2. Data Store Interfaces

This interface is used to get and store persistent data. All pre-existing non-static data used from here forward can be assumed to come from this interface unless stated otherwise.

3.5. Constraints and Assumptions

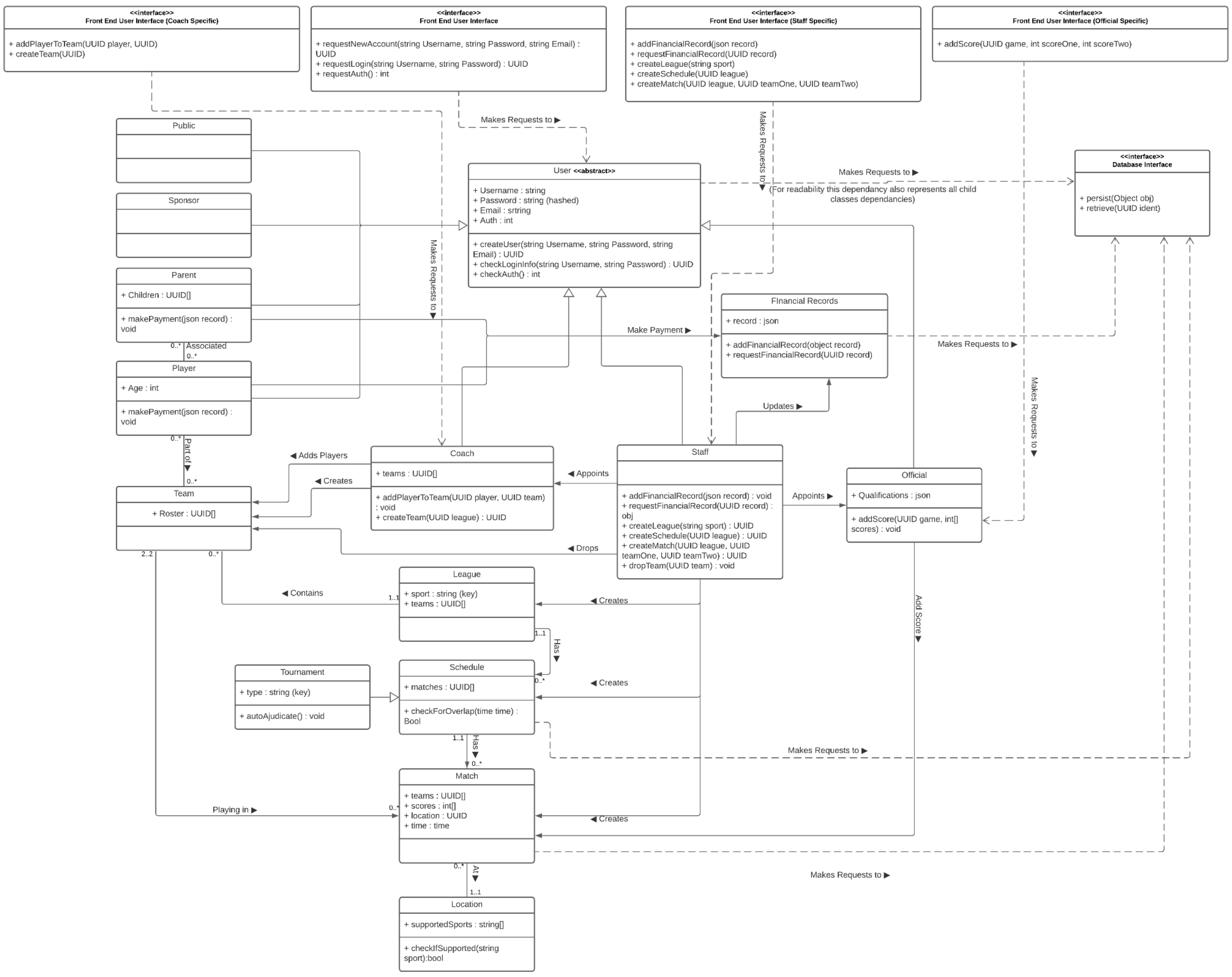
The following assumptions have been made:

- The data provided from the database will have been checked for compliance with any constraints. That is, the reporting system does not have to perform any data analysis.
- The UML provided in previous assignments was correct in every way except the things commented on by TAs. If anything about those diagrams was off, those errors would likely be reflected in this document.
- Views will be generated elsewhere. The BLaMo module only needs to ensure the DB has an accurate image of the system, a view into this system can be generated by some other subsystem.
- User security and access level can be verified outside of the system in the UI layer.

The only major constraint on the system was the structure of the DB interface, with a more sophisticated DB interface it would likely be possible to reduce the system complexity massively, but as a result of it's limited methods the classes need to store more data about their own relationships in order to guarantee access.

4. Structural Design

4.1. Class Diagram



<<interface>> Front End User Interface (Coach Specific)
+ addPlayerToTeam(UUID player, UUID)
+ createTeam(UUID)
<<interface>> Front End User Interface
+ requestNewAccount(string Username, string Password, string Email) : UUID
+ requestLogin(string Username, string Password) : UUID
+ requestAuth() : int
<<interface>> Front End User Interface (Staff Specific)
+ addFinancialRecord(json record)
+ requestFinancialRecord(UUID record)
+ createLeague(string sport)
+ createSchedule(UUID league)
+ createMatch(UUID league, UUID teamOne, UUID teamTwo)
<<interface>> Front End User Interface (Official Specific)
+ addScore(UUID game, int scoreOne, int scoreTwo)
Makes Requests to ▶
Makes Requests to ▼
Makes Requests to ▶
(For readability this dependancy also represents all child classes dependancies)
<<interface>> Database Interface
+ persist(Object obj)
+ retrieve(UUID ident)
Public
Sponsor
Parent
+ Children : UUID[]
+ makePayment(json record) : void
User <<abstract>>
+ Username : string
+ Password : string (hashed)
+ Email : srtring
+ Auth : int
+ createUser(string Username, string Password, string Email) : UUID
+ checkLoginInfo(string Username, string Password) : UUID
+ checkAuth() : int
Makes Requests to ▼
Financial Records
+ record : json
+ addFinancialRecord(object record)
+ requestFinancialRecord(UUID record)
Make Payment ▶
Makes Requests to ▶
0..* Associated 0..*
Player
+ Age : int
+ makePayment(json record) : void
Updates ▶
Makes Requests to ▼
0..* Part of ▼ 0..*
◀ Adds Players
◀ Creates
Coach
+ teams : UUID[]
+ addPlayerToTeam(UUID player, UUID team) : void
+ createTeam(UUID league) : UUID
◀ Appoints
Staff
+ addFinancialRecord(json record) : void
+ requestFinancialRecord(UUID record) : obj
+ createLeague(string sport) : UUID
+ createSchedule(UUID league) : UUID
+ createMatch(UUID league, UUID teamOne, UUID teamTwo) : UUID
+ dropTeam(UUID team) : void
Appoints ▶
Official
+ Qualifications : json
+ addScore(UUID game, int[] scores) : void
Team
+ Roster : UUID[]
◀ Drops
2..2
0..*
◀ Contains
1..1
League
+ sport : string (key)
+ teams : UUID[]
◀ Creates
1..1 Has ▼ 0..*
Add Score ▼
Tournament
+ type : string (key)
+ autoAjudicate() : void
Schedule
+ matches : UUID[]
+ checkForOverlap(time time) : Bool
◀ Creates
1..1 Has ▼ 0..*
Makes Requests to ▶
Match
+ teams : UUID[]
+ scores : int[]
+ location : UUID
+ time : time
Playing in ▶
0..*
◀ Creates
0..* At ▼ 1..1
Makes Requests to ▶
Location
+ supportedSports : string[]
+ checkIfSupported(string sport):bool

4.2. User Classes' Descriptions

4.2.1. User

Purpose: To model a person in the system primarily for login and authentication, and serve as a base to extend into specific user types.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.2.1.1. Attribute Descriptions

4.2.1.1.1. Attribute: Username

Type: string
Description: It is the username for login and display.
Constraints: must be unique

4.2.1.1.2. Attribute: Password

Type: string
Description: It is the password for login.
Constraints: It must be stored hashed and salted to avoid security concerns.

4.2.1.1.3. Attribute: Email

Type: string
Description: It is the email for login and notifications, staff may use to contact if needed.
Constraints: It must be a valid email.

4.2.1.1.4. Attribute: Auth

Type: int
Description: The authentication level for the user, needed frequently by frontend to know what kind of site to display for a given user.
Constraints: It must be a valid auth level.

4.2.1.2. Method Descriptions

4.2.1.2.1. Method: createUser

Return Type: UUID
Parameters: username, password, and email for a given user.
Return value: success or failure
Pre-condition: It has received the request from the BLaMo user request interface.
Post-condition: UUID given and new user persisted in DB
Methods called: DBInterface.persist().

Processing logic:

Check validity of username, password and email, then check uniqueness of username in DB, if all are valid create and persist new user in DB.

4.2.1.2.2. Method: checkLoginInfo
Return Type: UUID
Parameters: username, and password for a given user.
Return value: success or failure
Pre-condition: It has received the request from the BLaMo user request interface.
Post-condition: The UUID is now passed back as auth for session.

Processing logic:
Check validity of username, and password in DB, if all are a valid user return the UUID.

4.2.1.2.3. Method: checkAuth
Return Type: int
Parameters: void
Return value: NA
Pre-condition: It has received the request from the BLaMo user request interface.
Post-condition: Auth is given to the requester.

Processing logic:
Return Auth attribute

4.2.2. Public

Purpose: To model a member of the public in the system. Primarily in place for tracking and extensibility in the future.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.2.3. Sponsor

Purpose: To model a sponsor in the system. Primarily in place for tracking and extensibility in the future.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.2.4. Parent

Purpose: To model a parent in the system, store relations to child players, and give authentication for parent activities.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.2.4.1. Attribute Descriptions

4.2.4.1.1. Attribute: Children

Type: UUID[]
Description: is an array of UUIDs representing children of the parent
Constraints: none

4.2.4.2. Method Descriptions

4.2.4.2.1. Method: makePayment

Return Type: void
Parameters: payment
Return value: success or failure
Pre-condition: has received the request from the BLaMo user request interface
Post-condition: UUID given and new user persisted in DB
Methods called: FinancialRecord.addFinancialRecord()

Processing logic:
Submit the required payment and persist the record.

4.2.5. Player

Purpose: To model a player in the system.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.2.5.1. Attribute Descriptions

4.2.5.1.1. Attribute: Age

Type: int
Description: age of the player, relevant for certain leagues and sports.
Constraints: none

4.2.5.2. Method Descriptions

4.2.5.2.1. Method: makePayment

Return Type: void
Parameters: payment
Return value: success or failure
Pre-condition: It has received the request from the BLaMo user request interface.
Post-condition: The UUID given and new user persisted in DB.
Methods called: FinancialRecord.addFinancialRecord()

Processing logic:
Submit the required payment and persist the record.

4.2.6. Coach

Purpose: To model a coach in the system, and give authentication for coach activities.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.2.6.1. Attribute Descriptions

4.2.6.1.1. Attribute: teams

Type: UUID[]
Description: It is an array of UUIDs representing teams of the coach.
Constraints: none

4.2.6.2. Method Descriptions

4.2.6.2.1. Method: addPlayerToTeam

Return Type: void
Parameters: player to add to team, and team to add player to
Return value: success or failure
Pre-condition: It has received the request from the BLaMo user request interface.
Post-condition: The UUID given and new roster persisted in DB.
Methods called: DBInterface.persist()

Processing logic:
Check if a player can be validly added to a team based on sport or league restriction, then update DB with a

new team roster for team UUID.

4.2.6.2.2. Method: createTeam

Return Type: void
Parameters: league to add team to
Return value: success or failure
Pre-condition: has received the request from the BLaMo user request interface
Post-condition: UUID given and new team persisted in DB
Methods called: DBInterface.persist()

Processing logic:

Check if the team can be validly added to the league based on sport or league restriction, then update DB with new team, and return new team UUID and store as part of coach teams.

4.2.7. Staff

Purpose: To model a staff member in the system, and give authentication for staff activities
Constraints: None
Persistent: Yes (Stored through DB interface)

4.2.7.1. Method Descriptions

4.2.7.1.1. Method: addFinancialRecord

Return Type: void
Parameters: record to add
Return value: success or failure
Pre-condition: It has received the request from the BLaMo user request interface.
Post-condition: new record persisted in DB
Methods called: FinancialRecord.addFinancialRecord()

Processing logic:

Verify user authentication, then call FinancialRecord.addFinancialRecord() with the record.

4.2.7.1.2. Method: requestFinancialRecord

Return Type: json
Parameters: UUID for the record being requested
Return value: success or failure

Pre-condition: It has received the request from the BLaMo user request interface.
Post-condition: json record returned for use
Methods called: DBInterface.persist()

Processing logic:
Verify user authentication, then callFinancialRecord.requestFinancialRecord() with UUID.

4.2.7.1.3. Method: createLeague

Return Type: UUID
Parameters: sport that the league will support
Return value: success or failure
Pre-condition: It has received the request from the BLaMo user request interface.
Post-condition: The UUID given and new league persisted in DB.
Methods called: DBInterface.persist()

Processing logic:
Verify user authentication, then Create a new league, and persist it in the DB.

4.2.7.1.4. Method: createSchedule

Return Type: UUID
Parameters: league to create schedule for
Return value: success or failure
Pre-condition: It has received the request from the BLaMo user request interface.
Post-condition: The UUID given and new schedule persisted in DB.
Methods called: DBInterface.persist()

Processing logic:
Verify user authentication, then Create a new schedule, and persist it in the DB.

4.2.7.1.5. Method: createMatch

Return Type: UUID
Parameters: schedule to add match to
Return value: success or failure
Pre-condition: It has received the request from the BLaMo

user request interface.
Post-condition: UUID given and new team persisted in DB
Methods called: DBInterface.persist(), Schedule.checkForOverlap()

Processing logic:
Verify user authentication, then use overlap check to check for validity, then schedule if it is valid, persist the changes, and return the UUID of the new game.

4.2.7.1.6. Method: dropTeam
Return Type: void
Parameters: team to delete
Return value: success or failure
Pre-condition: has received the request from the BLaMo user request interface.
Post-condition: team no longer exists in the DB
Methods called: DBInterface.persist()

Processing logic:
Verify user authentication, then remove team from DB

4.2.8. Official

Purpose: To model a official in the system, and give sufficient authentication to add scores.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.2.8.1. Attribute Descriptions

4.2.8.1.1. Attribute: qualifications
Type: json
Description: is a qualification set

4.2.8.2. Method Descriptions

4.2.8.2.1. Method: addScore
Return Type: void
Parameters: game to add score to, scores of teams involved in order
Return value: success or failure
Pre-condition: game exists and coach is assigned to it
Post-condition: scores are mapped in game object, and updated object is persisted in DB.

Methods called: DBInterface.persist()

Processing logic:

Verify authentication, then copy scores from arguments into game's scores, then persist changes.

4.3. League and Match Classes' Descriptions

4.3.1. League

Purpose: To model a league in the system. Used as top-level data structure for most of the non-user operations.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.3.1.1. Attribute Descriptions

4.3.1.1.1. Attribute: sport

Type: string
Description: sport the league supports
Constraints: none

4.3.1.1.2. Attribute: teams

Type: UUID[]
Description: It is an array of UUIDs representing teams of the league.
Constraints: none

4.3.2. Team

Purpose: To model a team in the system, primarily a data structure.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.3.2.1. Attribute Descriptions

4.3.2.1.1. Attribute: Roster

Type: UUID[]
Description: It is an array of UUIDs representing players of the tem
Constraints: none

4.3.3. Schedule

Purpose: To model a schedule in the system, and allow for checking of overlap.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.3.3.1. Attribute Descriptions

4.3.3.1.1. Attribute: matches

Type: UUID[]
Description: It is an array of UUIDs representing matches of the schedule.
Constraints: none

4.3.3.2. Method Descriptions

4.3.3.2.1. Method: checkForOverlap

Return Type: boolean
Parameters: time to check
Return value: success or failure
Pre-condition: Has received the request from another class, likely staff.
Post-condition: Bool returned
Methods called: None

Processing logic:
Use the sport of the league the schedule is associated with to get match time, and see if the proposed time overlaps any of the other games associated.

4.3.4. Match

Purpose: The purpose is to model a match in the system, primarily a data structure.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.3.4.1. Attribute Descriptions

4.3.4.1.1. Attribute: teams

Type: UUID[]
Description: It is an array of UUIDs representing teams in the match.
Constraints: none

4.3.4.1.2. Attribute: scores

Type: int[]
Description: It is an array of ints representing scores of the teams at corresponding index.
Constraints: none

4.3.4.1.3. Attribute: location

Type: UUID
Description: corresponds to location
Constraints: none

4.3.4.1.4. Attribute: time

Type: time
Description: time of the match
Constraints: none

4.3.5. Tournament

Purpose: To model a tournament in the system, and automatically run tournament brackets.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.3.5.1. Attribute Descriptions

4.3.5.1.1. Attribute: type

Type: string
Description: It is a key to a set of adjudication rules in class method.
Constraints: none

4.3.5.2. Method Descriptions

4.3.5.2.1. Method: autoAdjudicate

Return Type: void
Parameters: none
Return value: success or failure
Pre-condition: It has received the request from the BLaMo user request interface.
Post-condition: Matches updated
Methods called: DBInterface.persist()

Processing logic:

When called, check over all scores in the tournament and check tournament type, then create new matches with new teams according to type and persist changes.

4.3.6. Location

Purpose: To model a location in the system, store supported sports for that given location.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.3.6.1. Attribute Descriptions

4.3.6.1.1. Attribute: supportedSports
Type: string[]
Description: array of supported sports
Constraints: none

4.3.6.2. Method Descriptions

4.3.6.2.1. Method: checkIfSupported
Return Type: bool
Parameters: sport to check against
Return value: success or failure
Pre-condition: It has received the request from somewhere.
Post-condition: Value returned
Methods called: none

Processing logic:
Check if the sport given is in the array or if it is compatible with one of the sports in the array.

4.4. Finance Classes' Descriptions

4.4.1. Financial Records

Purpose: To be a generalized record that can be extended as system needs become more specific.
Constraints: None
Persistent: Yes (Stored through DB interface)

4.4.1.1. Attribute Descriptions

4.4.1.1.1. Attribute: record
Type: json
Description: is a record containing financial data
Constraints: none

4.4.1.2. Method Descriptions

4.4.1.2.1. Method: addFinancialRecord

Return Type: void
Parameters: player to add to team, and team to add player to
Return value: success or failure
Pre-condition: has received the request
Post-condition: UUID given and new roster persisted in DB
Methods called: DBInterface.persist()

Processing logic:
```
    Persist changes
```

4.4.1.2.2. Method: requestFinancialRecord

Return Type: void
Parameters: player to add to team, and team to add player to
Return value: success or failure
Pre-condition: has received the request
Post-condition: UUID given and new roster persisted in DB
Methods called: DBInterface.persist()

Processing logic:
```
    Return requested record
```

4.5. BLaMo Request Interface Classes' Descriptions

Although this is one class, it is so large that it is easier to read and understand if we consider it as 4 separate components:

4.5.1. User Interface

Purpose: User related requests shared across all user types.
Constraints: None
Persistent: No (No attributes to persist)

4.5.1.1. Method Descriptions

4.5.1.1.1. Method: requestNewAccount

Return Type: UUID
Parameters: username, password, and email for a given user.
Return value: success or failure

Pre-condition: has received the request from the BLaMo user request interface
Post-condition: UUID given and new user persisted in DB
Methods called: User.requestNewAccount()

Processing logic:
Check validity of username, password and email, then check uniqueness of username in DB, if all are valid create and persist new user in DB.

4.5.1.1.2. Method: requestLogin

Return Type: UUID
Parameters: username, and password for a given user.
Return value: success or failure
Pre-condition: has received the request from the BLaMo user request interface
Post-condition: UUID is now passed back as auth for session
Methods called: User.login()

Processing logic:
Check validity of username, and password in DB, if all are a valid user return the UUID.

4.5.1.1.3. Method: requestCheckAuth

Return Type: int
Parameters: void
Return value: NA
Pre-condition: has received the request from the BLaMo user request interface
Post-condition: auth is given to requester
Methods called: User.checkAuth()

Processing logic:
Return Auth attribute

4.5.2. User Interface (Coach)

Purpose: Coach specific user interface requests
Constraints: None
Persistent: No (No attributes to persist)

4.5.2.1. Method Descriptions

4.5.2.1.1. Method: addPlayerToTeam

Return Type: void
Parameters: player to add to team, and team to add player to
Return value: success or failure
Pre-condition: has received the request to the BLaMo user request interface
Post-condition: UUID given and new roster persisted in DB
Methods called: Coach.addPlayerToTeam()

Processing logic:
Check if player can be validly added to team based on sport or league restriction, then update DB with new team roster for team UUID

4.5.2.1.2. Method: createTeam

Return Type: void
Parameters: league to add team to
Return value: success or failure
Pre-condition: has received the request to the BLaMo user request interface
Post-condition: UUID given and new team persisted in DB
Methods called: Coach.createTeam()

Processing logic:
Check if team can be validly added to league based on sport or league restriction, then update DB with new team, and return new team UUID and store as part of coach.teams

4.5.3. User Interface (Staff)

Purpose: Staff specific user interface requests
Constraints: None
Persistent: No (No attributes to persist)

4.5.3.1. Method Descriptions

4.5.3.1.1. Method: addFinancialRecord

Return Type: void
Parameters: record to add
Return value: success or failure

Pre-condition: has received the request to the BLaMo user request interface
Post-condition: new record persisted in DB
Methods called: Staff.addFinancialRecord()

Processing logic:
Verify user authentication, then call FinancialRecord.addFinancialRecord() with the record.

4.5.3.1.2. Method: requestFinancialRecord
Return Type: json
Parameters: UUID for the record being requested
Return value: success or failure
Pre-condition: has received the request to the BLaMo user request interface
Post-condition: json record returned for use
Methods called: Staff.requestFinancialRecord()

Processing logic:
Verify user authentication, then callFinancialRecord.requestFinancialRecord() with UUID.

4.5.3.1.3. Method: createLeague
Return Type: UUID
Parameters: sport that the league will support
Return value: success or failure
Pre-condition: has received the request to the BLaMo user request interface
Post-condition: UUID given and new league persisted in DB
Methods called: Staff.createLeague()

Processing logic:
Verify user authentication, then Create a new league, and persist it in the DB.

4.5.3.1.4. Method: createSchedule
Return Type: UUID
Parameters: league to create schedule for
Return value: success or failure
Pre-condition: has received the request to the BLaMo user request interface
Post-condition: UUID given and new schedule persisted in

DB
Methods called: Staff.createSchedule()

Processing logic:
Verify user authentication, then Create a new schedule, and persist it in the DB

4.5.3.1.5. Method: createMatch

Return Type: UUID
Parameters: schedule to add match to
Return value: success or failure
Pre-condition: has received the request to the BLaMo user request interface
Post-condition: UUID given and new team persisted in DB
Methods called: Staff.createMatch()

Processing logic:
Verify user authentication, then use overlap check to check for validity, then schedule if it is valid, persist the changes, and return the UUID of the new game.

4.5.3.1.6. Method: dropTeam

Return Type: void
Parameters: team to delete
Return value: success or failure
Pre-condition: has received the request to the BLaMo user request interface.
Post-condition: team no longer exists in the DB
Methods called: Staff.dropTeam()

Processing logic:
Verify user authentication, then remove team from DB

4.5.4. User Interface (Official)

Purpose: Official specific user interface requests
Constraints: None
Persistent: No (No attributes to persist)

4.5.4.1. Method Descriptions

4.5.4.1.1. Method: addScore

Return Type: void

Parameters: game to add score to, scores of teams involved in order

Return value: success or failure

Pre-condition: game exists and coach is assigned to it

Post-condition: scores are mapped in game object, and updated object is persisted in DB.

Methods called: Official.addScore()

Processing logic:

Verify authentication, then copy scores from arguments into game's scores, then persist changes.

5. Non-functional Requirements

5.1. On Use of Non-Functional Requirements

In our previous designs of the SLAM BLaMo system, non-functional requirements weren't included/considered before. This was mostly a result of attempting to design an intuitive system without having any constraints or system properties to slow down development. Now that we are finalizing our design for the system, we must create non-functional requirements in order for our functional requirements to work properly and to verify the correctness of the system. For the system to work as intended, it must meet the following non functional requirements.

5.2. Performance Requirements

5.2.1. Any page on the system will take no more than 10 seconds to load.

5.2.1.1. When using the system, we want the user to be able to use the software as they desire . In order to do this, the system needs to operate smoothly and timely to allow the user to access the proper pages. In our design, we made sure that only the necessary information would be displayed on a single page at a time.

5.2.2. System must support many leagues (e.g. 15 leagues) with as many as 9 teams included in each league.

5.2.2.1. For the SLAM BLaMo system to be successful, it needs to be able to manage and support many leagues and teams at the same time. Otherwise, the system cannot be used by many people who intend on playing in a league or team. Also, for this to be supported, we made sure that more leagues and teams can be added by the correct type of user.

5.2.3. System must support many concurrent users up to as many as 500 during peak traffic hours.

5.2.3.1. If there are many players, managers/coaches, and staff participating in any league, we must be able to support having many users online at the same time. Otherwise, multiple users will not be able to perform their desired tasks on the system.

5.2.4. Any update to the website (e.g. match scheduling, team adding a player) shall take no more than 5 seconds to become visible across all views.

5.2.4.1. When a match is scheduled or a team adds a player, we want the change to be almost instant. This is to allow the user to do any other potential interactions with the system that would be based on the update.

5.3. Security Requirements

5.3.1. The system shall require the user to log into the system using both their username and password.

5.3.1.1. This is standard protocol for security to make sure any account is being accessed by the right person. Also by having a username and password to identify an account, we can distinguish different user types in the system to allow certain users to have proper access to different functionality in the system.

5.3.2. A user password used for login must have greater than 8 characters but less than 100 characters.

5.3.2.1. To lower the chances of having an account compromised, we set a minimum of 9 characters for the password. However, to prevent any type of data overflow or to save storage, the character limit was set 100.

5.3.3. Any user with an incomplete registration shall be required to update any necessary information.

5.3.3.1. A user can successfully register and make an account without having to include all the details. However, they would have very limited access to the system. For example, we need a player's date of birth to determine their eligibility to play in a league. If this information isn't in the system, they are unable to be added into a team or sign up as a free agent.

5.4. Software Quality Attributes

5.4.1. All necessary communication between the system and user shall be expressed in the standard ASCII character set.

5.4.1.1. This was implemented for the sake of usability. By having a standard character set used in the system, users can intuitively interact with the system without much conflict (e.g. setting a match time using dates).

5.4.2. The system shall be accessible from any type of browser.

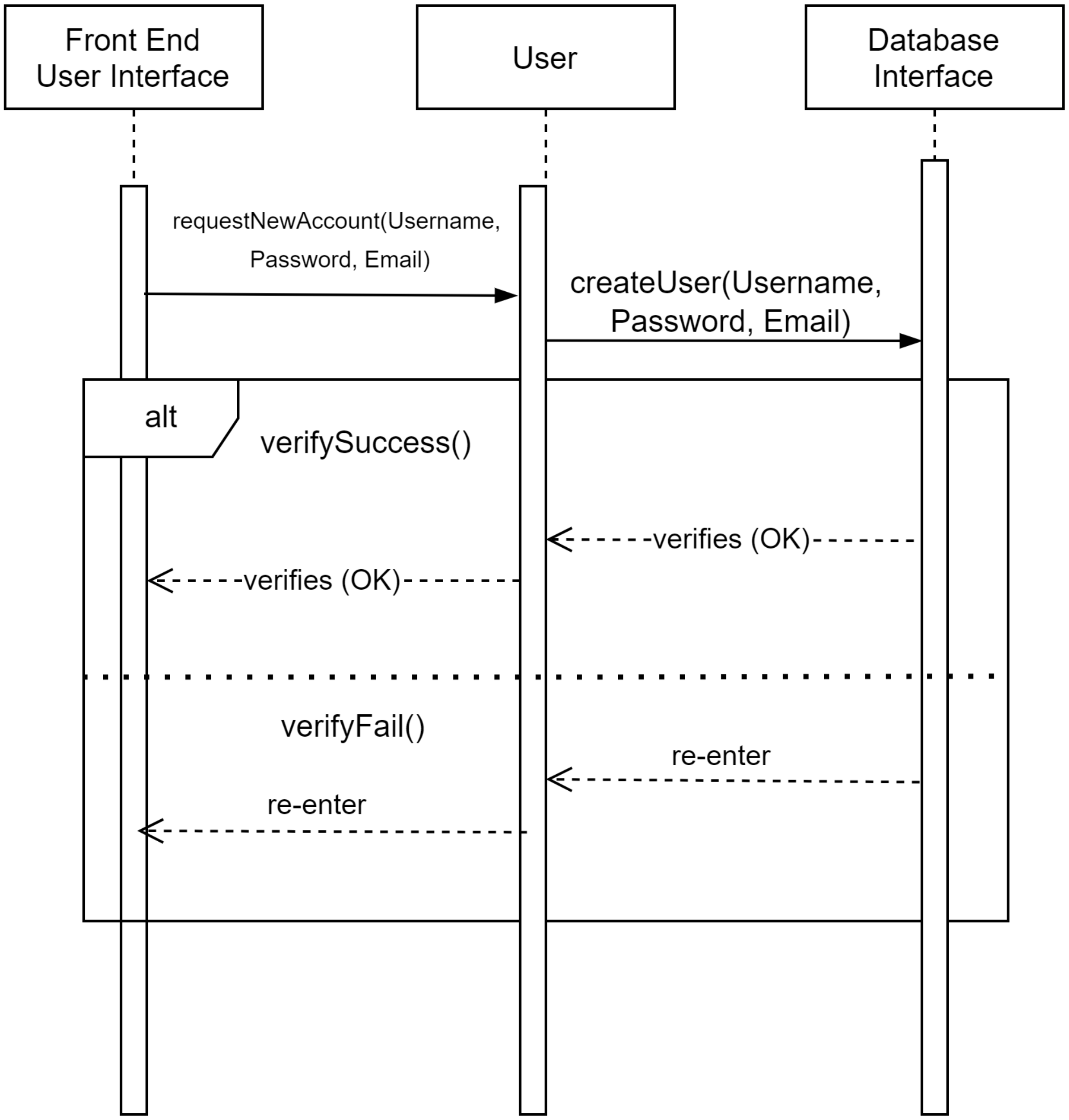
5.4.2.1. Allowing the system to be accessed from any browser lets more users use the system without having to go through any extra steps of installing more software. This also helps with portability from allowing the system to be run from one browser to another.

5.4.3. Staff must be able to add new leagues, tournaments, games and financial records.

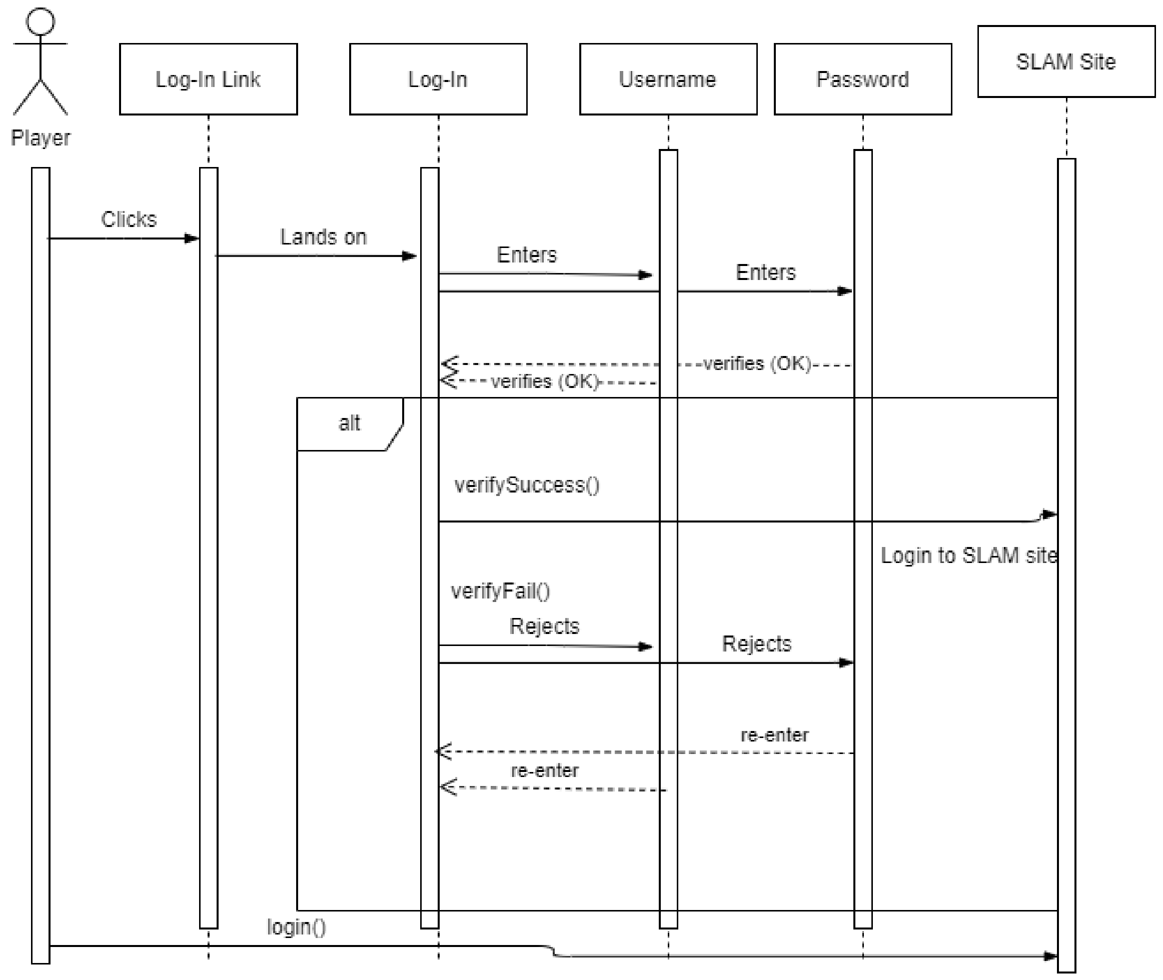
5.4.3.1. This allows the system to be extensible and lets any staff user update or add information into the system without having the system needing to manually change anything on its own. Also, having this feature is critical to the success of the SLAM BLaMo system by having more leagues in the system for players to participate in.

6. Dynamic Models

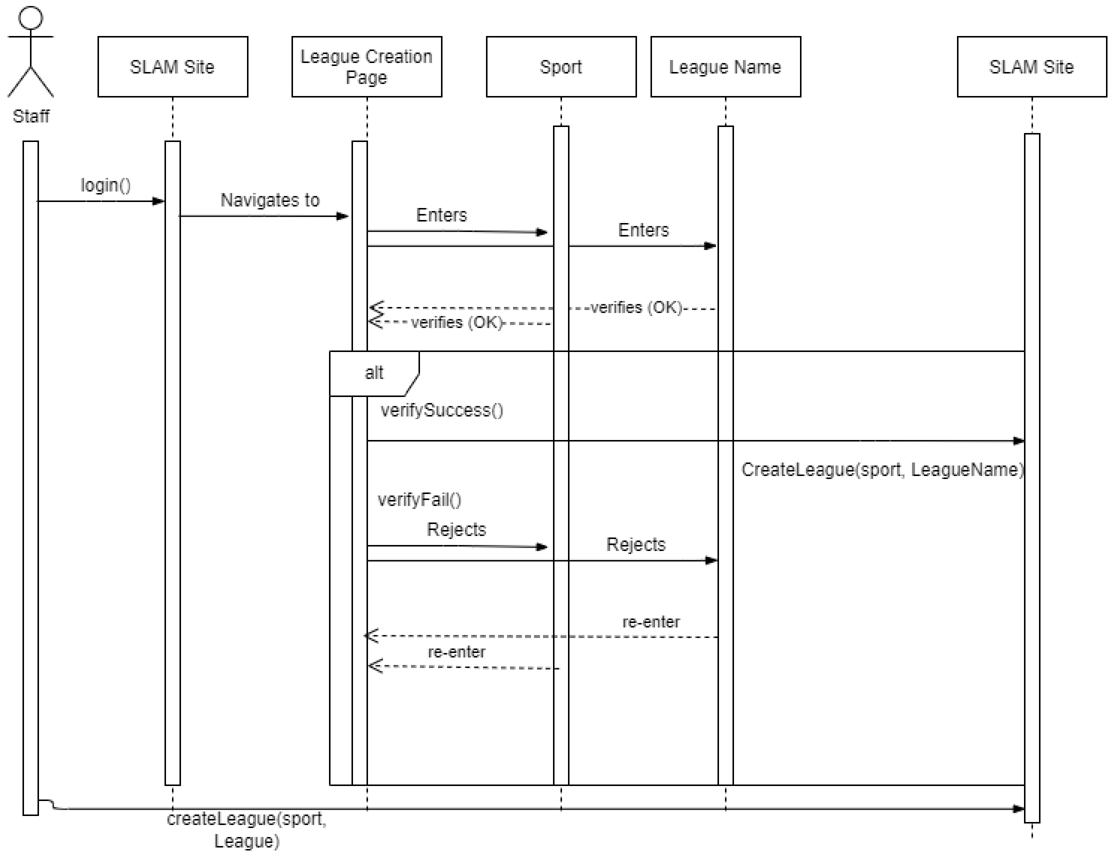
6.1. Creating an Account (Use Case 1)



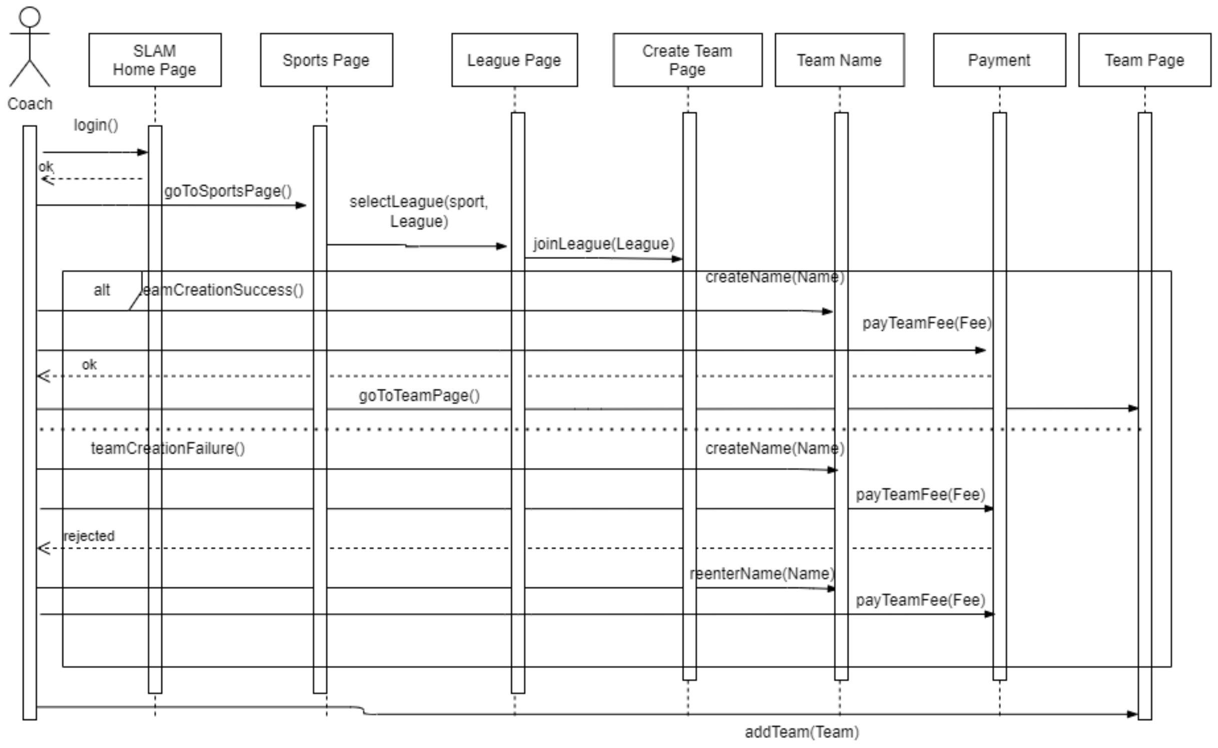
6.2. Sign into the System (Use Case 2)



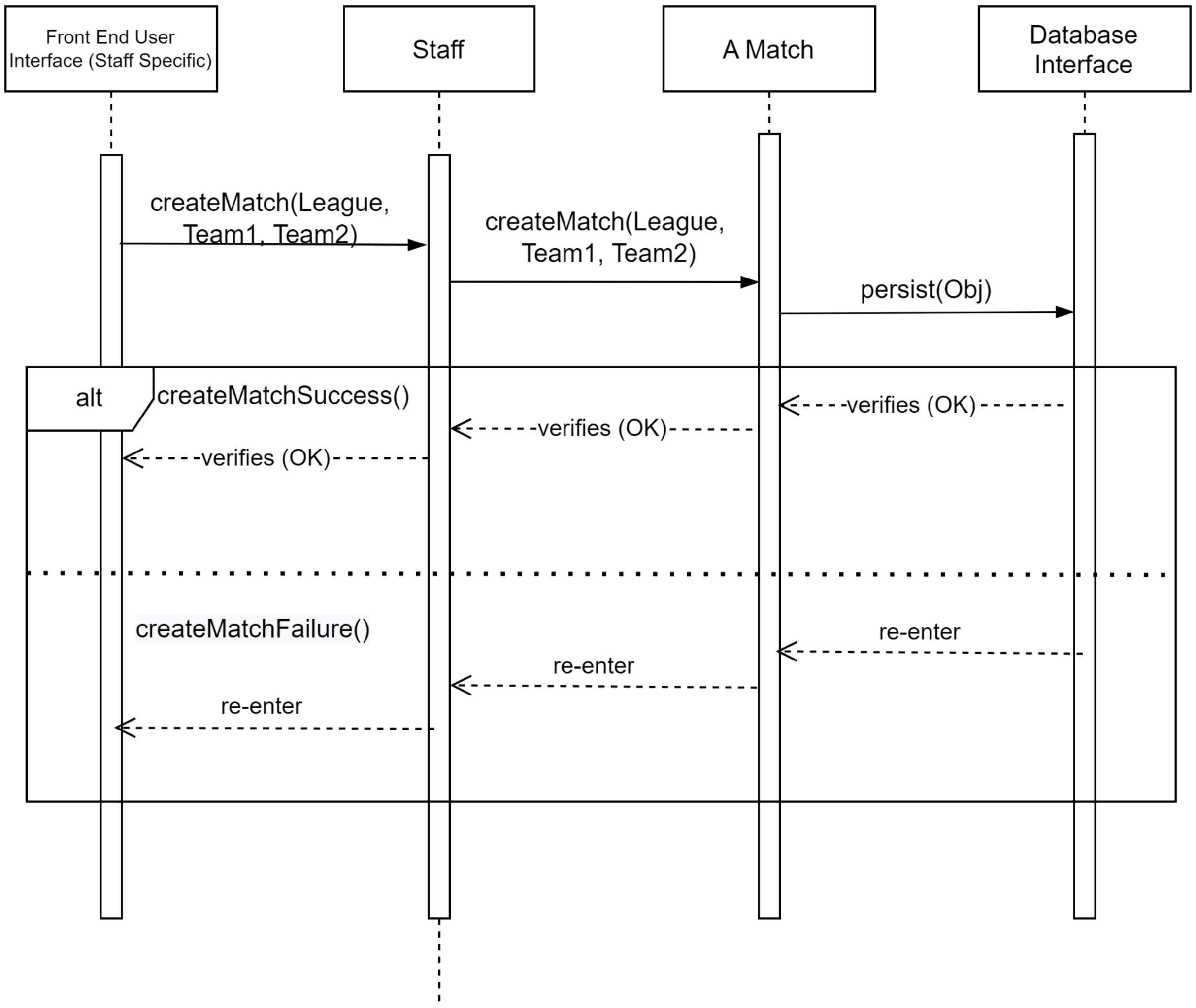
6.3. Create a League (Use Case 3)



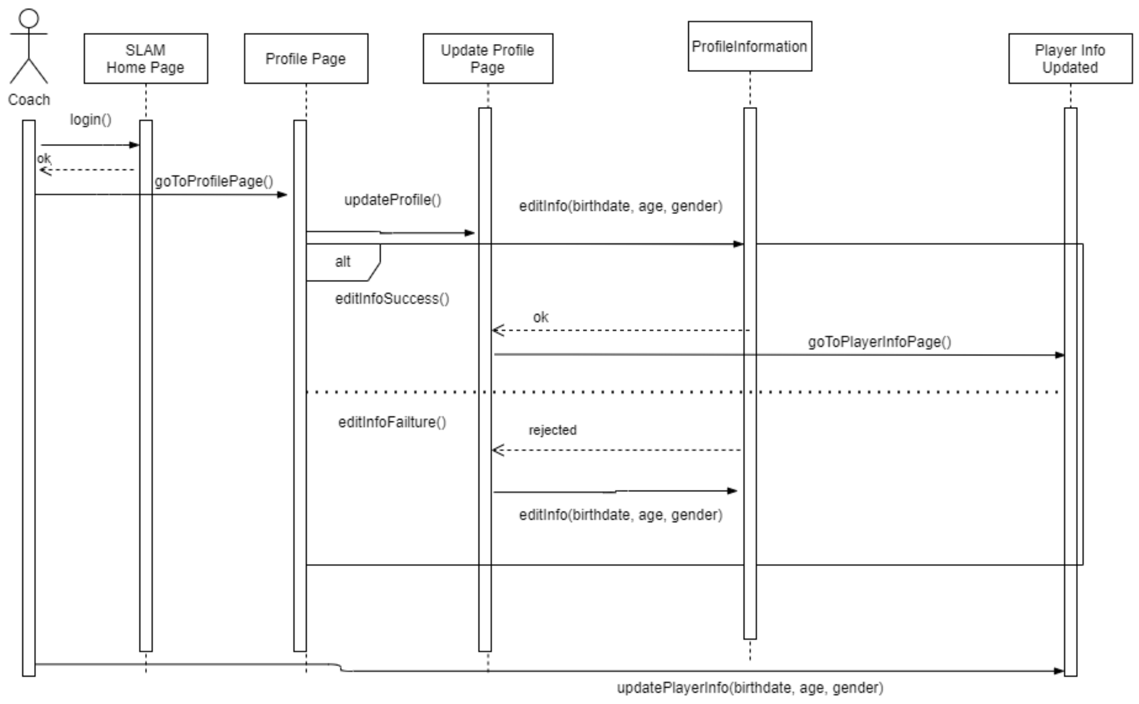
6.4. Create a Team (Use Case 4)



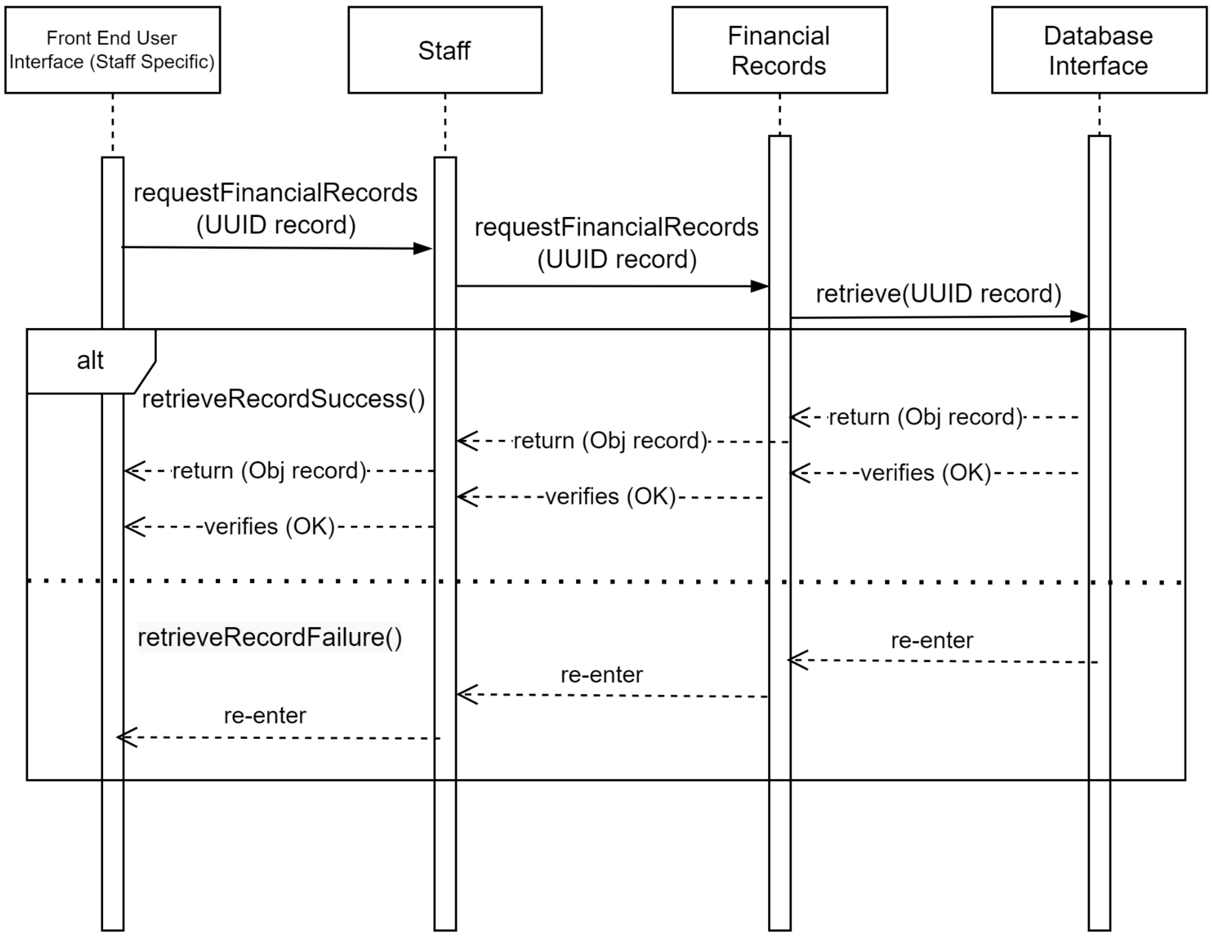
6.5. Schedule a Match (Use Case 7)



6.6. Updating Player Information (Use Case 12)



6.7. Accessing Financial Records (Use Case 13)


Front End User Interface (Staff Specific)
Staff
Financial Records
Database Interface
requestFinancialRecords (UUID record)
requestFinancialRecords (UUID record)
retrieve(UUID record)
alt
retrieveRecordSuccess()
return (Obj record)
return (Obj record)
return (Obj record)
verifies (OK)
verifies (OK)
verifies (OK)
retrieveRecordFailure()
re-enter
re-enter
re-enter