



POLITECNICO DI MILANO

Software Engineering 2

**CodeKataBattle - Improve your
programming skills online**

Design

Document

Authors:

Nicolò Giallongo - 10764261

Giovanni Orciuolo - 10994077

Giuseppe Vitello - 10766482

Table of Contents

1 - Introduction.....	3
1.1 - Purpose.....	3
1.2 - Scope.....	3
1.3 - Definitions, Acronyms, Abbreviations.....	3
1.4 - Revision History.....	5
1.5 - Reference Documents.....	5
1.6 - Document Structure.....	5
2 - Architectural Design.....	6
2.1 - Overview.....	6
2.2 - Component View.....	6
2.2.1 - Component Diagram.....	7
2.2.2 - Components Description.....	8
2.3 - Deployment View.....	9
2.3.1 - Deployment View Description.....	9
2.4 - Component Interfaces.....	10
2.5 - Runtime View.....	13
2.6 - Selected Architectural Styles and Patterns.....	41
2.7 - Other Design Decisions.....	42
2.7.1 - API-first Development.....	42
2.7.2 - Entity-Relationship Diagram.....	42
2.7.3 - Separate databases: Main DBMS and Notification DBMS.....	43
3 - User Interface Design.....	44
3.1 - Overview.....	44
3.2 - Mockups.....	46
4 - Requirements Traceability.....	50
5 - Implementation, Integration and Test Plan.....	53
5.1 - Implementation and Integration.....	53
5.2 - Test Plan.....	54
5.2.1 - Functional Testing (Unit + Integration).....	54
5.2.2 - End to End Testing (E2E).....	55
5.2.3 - MVP testing.....	55
6 - Effort Spent.....	55
7 - References.....	56

1 - Introduction

1.1 - Purpose

The purpose of the CKB application is to provide students and educators with a platform to improve their programming skills by taking part in friendly competitions (called “Tournaments”) where they can resolve various programming problems (called “Katas”) in their preferred language of choice (e.g. Java or Python).

The platform relies heavily upon GitHub in order to store solutions and to run tests and other activities on each push. As such, the users are required to use their GitHub account to access the application (or register one if they don’t have it).

1.2 - Scope

In this section of the document, we will discuss the main design choices that we have taken in relation to the domain of the product.

Given that CKB is an application that aims to allow its users to compete with other users in online code competition, it seems natural to implement it as a client-server application. More specifically, we agreed on using a three-tier architecture (presentation tier, business tier and data tier) to maintain a separation between the user interface, the logic of the system and the data, and in this way improve the maintainability, the stability and the security of the system. To further improve these aspects of the CKB application we decided to use the microservice approach to develop the system. In this way we also improve the scalability of the system, allowing it to adapt more easily and efficiently to different amounts of active users.

1.3 - Definitions, Acronyms, Abbreviations

In order to reduce ambiguity as much as possible, this document heavily uses acronyms and abbreviations to refer to entities and concepts in the CKB system. This section is aimed at defining each of these acronyms and abbreviations in a precise and concise way.

Acronym	Definition
IU	Interested Student: a Student <u>U</u> is Interested by a notification <u>N</u> : <ul style="list-style-type: none">• if <u>U</u> has selected the option to receive notification from all the users.• if <u>U</u> has selected the option to receive notification only from their friend and the user who has sent <u>N</u> is a friend of <u>U</u>.

	<ul style="list-style-type: none"> • \underline{N} is a notification of a Tournament in which \underline{U} is subscribed.
SST	Students Subscribed to a Tournament.
OTC	Original Tournament Creator.
TC	Tournament Coordinator, appointed by the OTC.
CDT	Current DateTime.
ETD	Enrollment Tournament Deadline. After this deadline, it is not possible to join the Tournament as a Student.
FTD	Final Tournament Deadline. After this deadline, the Tournament is ended and the final leaderboard is made available.
MNS	Maximum Number of Subscribers. It is the maximum number of Students that can subscribe to a specific Tournament.
EBD	Enrollment Battle Deadline. After this deadline, it is not possible to join the Battle as a Student.
FBD	Final Battle Deadline. After this deadline, the Battle is ended and the final leaderboard is made available. Moreover, the Tournament related to the Battle is updated with the new leaderboard.
OME	Optional Manual Evaluation. Personal score assigned by the Educator, who checks and evaluates the work done by students (the higher the better).
MAE	Mandatory Automated Evaluation, which includes: <ul style="list-style-type: none"> • Functional aspects, measured in terms of number of test cases that pass out of all test cases (the higher the better); • Timeliness, measured in terms of time passed between the registration deadline and the last commit (the lower the better); • Quality level of the sources, extracted through Static Analysis Tools that consider multiple aspects such as security, reliability, and maintainability (the higher the better). Aspects are selected by the educator at battle creation time.
DAS	Directly Accepted Student: a Student \underline{U} is directly accepted in a Tournament \underline{I} if: <ul style="list-style-type: none"> • \underline{I} is public; • \underline{I} is friends only and \underline{U} is a friend of the OCT of \underline{I}.
EB	End Battle:

	<ul style="list-style-type: none"> • If OME is not required, EB occurs when the FBD is reached. • If OME is required, EB is after the Educator performs it, or when the FTD is reached.
SAT	Static Analysis Tools.

1.4 - Revision History

Version	Date	Changelog
1.0	03/01/2024	First draft of the document. Completed section 1, sections 2 and 3 are work in progress.
1.1	06/01/2024	Added Table of Contents. Completed section 2, 3 and 5. Section 4 WIP.
1.2	07/01/2024	Final version of the document to be delivered.
1.3	14/02/2024	Added section 2.7.4 (sandboxing)

1.5 - Reference Documents

- The specification document Assignment RDD AY 2023-2024.pdf

1.6 - Document Structure

This document is divided in 5 main parts:

1. **Introduction:** introduces the main architectural and design choices made and briefly explains the motivations behind them.
2. **Architectural Design:** shows in detail the architectural decisions adopted and how they are developed and combined to create the global architecture of the system.
3. **User Interface Design:** shows the design adopted for the user interface, presenting a preview of the interface using the corresponding mockups.
4. **Requirements Traceability:** shows the mapping between the requirements defined in RASD and the design elements previously discussed.
5. **Implementation, Integration and Test Plan:** shows the order in which we plan to implement subsystems and components as well as the plan of the integration and test.
6. **Effort Spent:** contains information about the number of hours each group member has worked on this document.
7. **References:** contains information about the resources used to redact this document.

2 - Architectural Design

2.1 - Overview

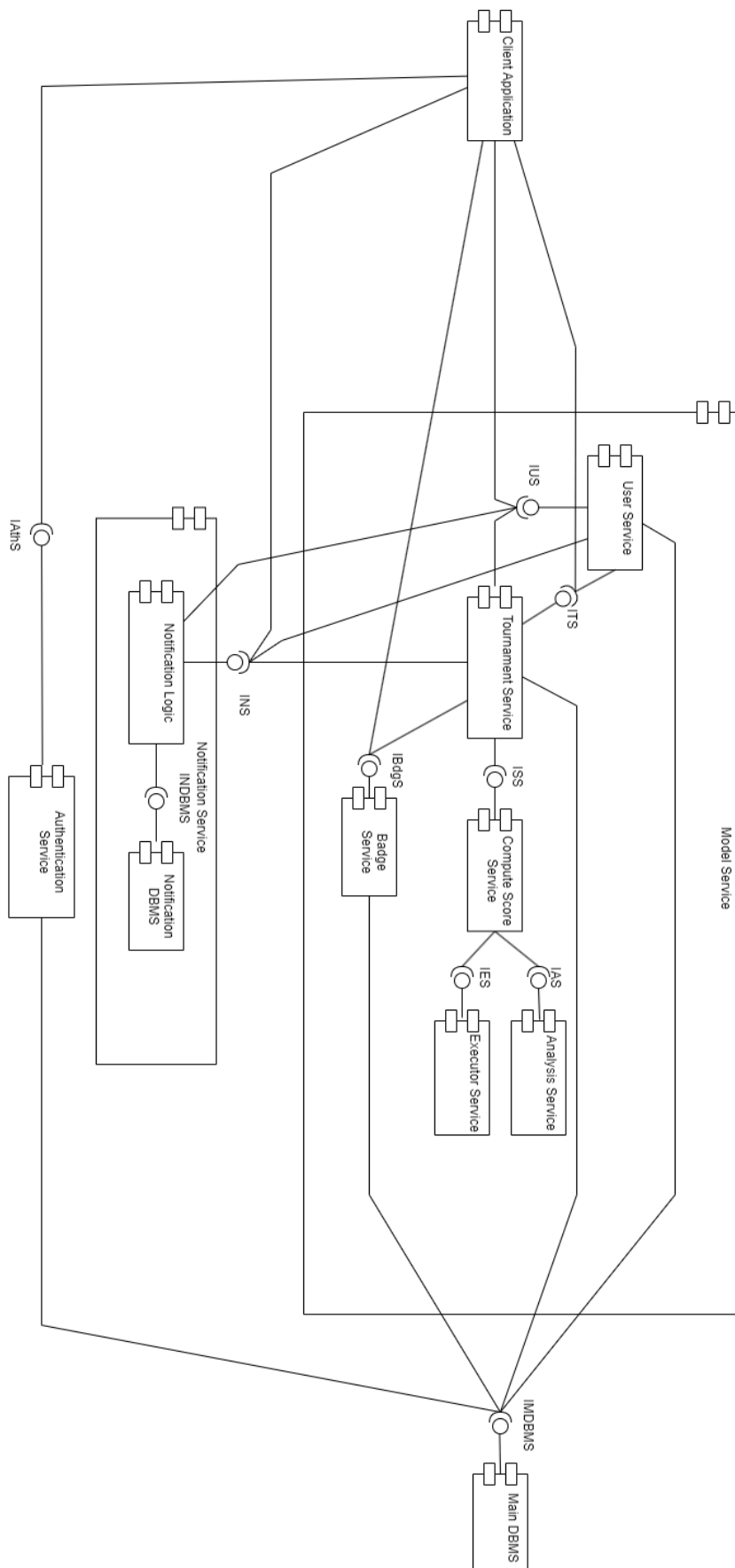
In this section, a general architecture of the CKB system is provided. The architectural choices are summarized in two main design decisions:

- **Client-Server architecture.** Consisting of a Presentation Tier, a Business Tier and a Data Tier. The Presentation Tier is the UI directly shown to the end user: its only objective is to graphically render and present the application to the user. The Business Tier handles everything related to the business logic of the system, by using the internal components (the services). The Data Tier persists the data through two databases: a PostgreSQL (Main DBMS) and a MongoDB instance (Notification DBMS). Further details are provided in section 2.3.1.
- **Microservices.** The Business Tier of the system is divided into several isolated services, in order to reduce coupling as much as possible and to improve scalability of the system as a whole.

2.2 - Component View

This section describes all the identified components and the relationships between them. The Component Diagram precisely shows the dependencies of all the interfaces. After that, a brief description of each component is shown.

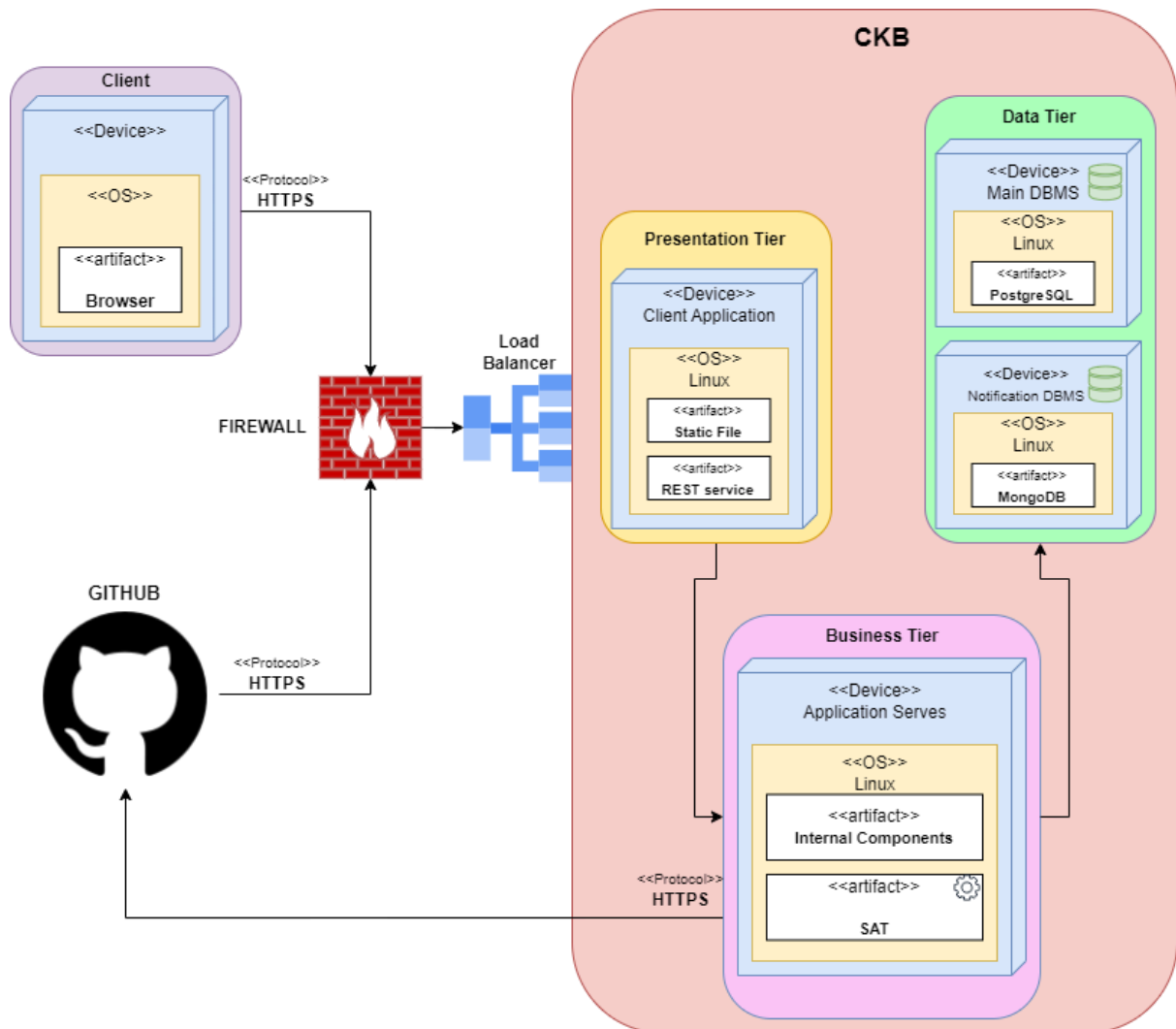
2.2.1 - Component Diagram



2.2.2 - Components Description

- **Client Application.** Represents the web application used by the final user to interface with the CKB system, so it reads the request and sends the html pages.
- **Authentication Service.** Handles authentication with GitHub following the OAuth 2.0 protocol and subsequent token verification processes.
- **User Service.** Handles registered users management, friends' list and privacy policy..
- **Tournament Service.** Handles everything related to tournaments, from creation to end of the Tournament, and creation of battle.
- **Compute Score Service.** Handles score computation.
- **Executor Service.** Handles execution of code in an isolated sandbox. It is used to run kata submissions and check their output against the provided tests.
- **Analysis Service.** Handles execution of static analysis tools (SATs) in an isolated sandbox.
- **Notification Service.** Abstracts the process of delivering and storing notifications.
- **Main DBMS:** Holds and manages data about Tournaments, Battles, Users, Badges and the relationships between them. The DBMS of choice is PostgreSQL.
- **Notification DBMS:** Holds and manages data about notifications. The DBMS of choice will be a NoSQL solution (Redis).

2.3 - Deployment View



2.3.1 - Deployment View Description

Presentation Tier: It is the frontend layer in the 3-tier architecture, it is the user interface. Communication with the client using the REST standard. The other communication is only with the Business-Tier in order to reduce the dependence. The client sends a request via Browser, this request arrives in the Presentation-Tier it responds with html pages or messages.

It is a Nginx web server.

Business Tier: This is the middle tier between Presentation and Data. It receives requests by the Presentation-Tier, computes them, if needed communicate with Data-Tier in order to receive data and at the end send the information required at the Presentation-Tier.

It also can receive the code of the repository by GitHub in order to check them and communicate with the Data-Tier to update the data, and the Business-tier communicate with GitHub to authenticate the Client.

Data Tier: This is the tier that keeps track of all data in the system, so all Tournament, Battle, User, Notification and the relationship between them.

Client: It is the client that wants to interact with the system via HTTPS request.

Firewall: It is a device that monitors the packets incoming to the system, if a packet is potentially dangerous it is not forwarder. They are placed before the load balancers, in this way the only packets that enter the network are considered safe

LoadBalancer: A load balancer is a device which performs load balancing. This is the process of distributing a set of requests over a group of resources at the presentation-tier, with the intent of increasing performance, scalability and soundness of the system. It uses Nginx.

2.4 - Component Interfaces

In this section, for each component interface, there are listed methods with their parameter types and return value. Exceptions thrown are omitted as they are not of interest.

- **IUserService (IUS):**
 - addFriend(notification: JSON): void
 - updateNotificationSetting(userId: int, privacyLevel: PrivacyLevel): void
 - getUser(userId: int): User
 - getTournamentsParticipating(userId: int): List<Tournament>
 - getTournamentsCoordinating(userId: int): List<Tournament>
 - getFriends(userId: int): List<User>
 - getBadges(userId: int): List<Badge>
 - getNotificationSetting(userId: int): PrivacyLevel
 - getInterestedUser(creatorId: int, privacy: TournamentPrivacyLevel): List<Int>
 - canSend(senderId: int, receiverId: int): bool
 - areFriends(userId1: int, userId2: int): bool
- **IAuthService (IAthS):**
 - getOAuthToken(accessCode: string): string
 - getUserProfile(token: string): GHUserProfile
- **ITournamentService (ITS):**
 - createTournament(tournament: Tournament): Tournament
 - joinTournament(notification: JSON): void
 - @overload joinTournament(tournamentId: int, userId: int): void
 - addCoordinator(notification: JSON): void
 - banParticipant(clientId: int, participantId: int, tournamentId: int): void
 - updateETD(date: DateTime, tournamentId: int, userId: int): void
 - updateFTD(date: DateTime, tournamentId: int, userId: int): void
 - areFriends(userId1: int, userId2: int): bool
 - createBattle(battle: Battle): Battle
 - joinBattle(notification: JSON): void

- @overload joinBattle(battleId: int, userId: int): void ??
- performOME(battleId: int ,teamId: int, userId: int, score: float): void
- updateEBD(date: DateTime, battleId: int, userId: int): void
- updateFBD(date: DateTime, battleId: int, userId: int): void
- getRepository(teamId: int): String
- addRepository(battleId: int, userId: int, link: String): void
- updateLeaderboard(battleId: int, tournamentId: int):void
- Getters/Setters
- **IScoreService (ISS):**
 - computeScore(kata: Kata, code: String , evaluationSetting: EvaluationSetting): float
- **IExecutorService (IES):**
 - runTests(test: List<KataTest>, code: String): List<KataTestResult>
- **IAnalysisService (IAS):**
 - runAnalysis(code: String, evaluationSetting: EvaluationSetting): List<float>
- **INotificationService (INS):**
 - sendFriendRequestNotify(senderId: int, receiverId: int): void
 - interactFriendRequestNotify(receiverId: int, notificationId: int, accepted: bool): void
 - sendTeamInvitationNotify(battleId: int, teamId: int, senderId: int, receiverId:int): void
 - interactTeamInvitationNotify(receiverId: int, notificationId: int, accepted: bool): void
 - sendBattleCreationNotify(battleId: int, participantIds:List<int>): void
 - interactBattleCreationNotify(receiverId: int, notificationId: int, accepted: bool): void
 - sendBattleOMENotify(battleId: int, participantIds:List<int>): void
 - sendBattleEndNotify(battleId: int, participantIds:List<int>): void
 - sendTournamentCreationNotify(tournament: Tournament): void
 - interactTournamentCreationNotify(receiverId: int, notificationId: int, accepted: bool): void
 - sendTournamentCoordinatorNotify(senderId: int, receiverId: int): void
 - interactTournamentCoordinatorNotify(receiverId: int, notificationId: int, accepted: bool): void
 - sendTournamentEndNotify(tournamentId: int, participantIds: List<int>): void
 - sendParticipantAcceptation(tournamentId: int, userId: int): void
 - interactParticipantAcceptation(receiverId: int, tournamentId: int, userId: int, accepted: bool): void
 - openNotification(userId: int): List<Notification>
 - sendEditTournamentDeadlineNotify(userIds: List<Int>,tournamentId: int): void
 - sendEditBattleDeadlineNotify(userIds: List<Int>,battleId: int): void

- **IBadgeService (IBdgS):**
 - getBadge(badgetId: int): Badge
 - createBadge(badge: Badge): Badge
 - applyBadges(tournamentId: int): void
 - getBadges(userId: int): List<Badge>
- **IMainDBMS (IMDBMS):**
 - persistScore(teamId: int, score: float): void
 - storeUser(userProfile: GHUserProfile): void
 - isFullTeam(teamId: int): bool
 - addTeam(battleId: int, userId: int): int
 - getTeam(battleId: int, userId: int): int
 - deleteTeamParticipant(teamId: int, userId: int): void
 - addTeamParticipant(teamId: int, userId: int): void
 - isFullTournament(tournamentId: int): bool
 - addTournamentParticipant(tournamentId: int, userId: int): void
 - getUser(userId: int): User
 - getTournamentsParticipating(userId: int): List<Tournament>
 - getTournamentsCoordinating(userId: int): List<Tournament>
 - getFriends(userId: int): List<User>
 - getBadges(userId: int): List<Badge>
 - createTournament(tournament: Tournament): Tournament
 - writeCoordinator(tournamentId: int, coordinatorId: int): void
 - createBadge(badge: Badge): Badge
 - getBadge(badgetId: int): Badge
 - createBattle(battle: Battle): Battle
 - updateScoreOme(battleId: int, teamId: int, userId: int, score: float): void
 - updateLeaderboard(battleId: int, tournamentId: int): void
 - updateEBD(date: DateTime, battleId: int, userId: int): void
 - updateFBD(date: DateTime, battleId: int, userId: int): void
 - updateETD(date: DateTime, tournamentId: int, userId: int): void
 - updateFTD(date: DateTime, tournamentId: int, userId: int): void
 - deleteParticipant(clientId: int, participantId: int, tournamentId: int)
 - endTournament(tournamentId: int): Tournament
 - isEBDexpired(battleId: int): bool
 - getRepository(teamId: int): String
 - addRepository(battleId: int, userId: int, link: String): void
 - updateNotificationSetting(userId: int, privacyLevel: PrivacyLevel): void
 - addFriend(notification: JSON): void
 - getNotificationSetting(userId: int): PrivacyLevel
 - updateBadges(assignedBadges: List<assignedBadge>): void

2.5 - Runtime View

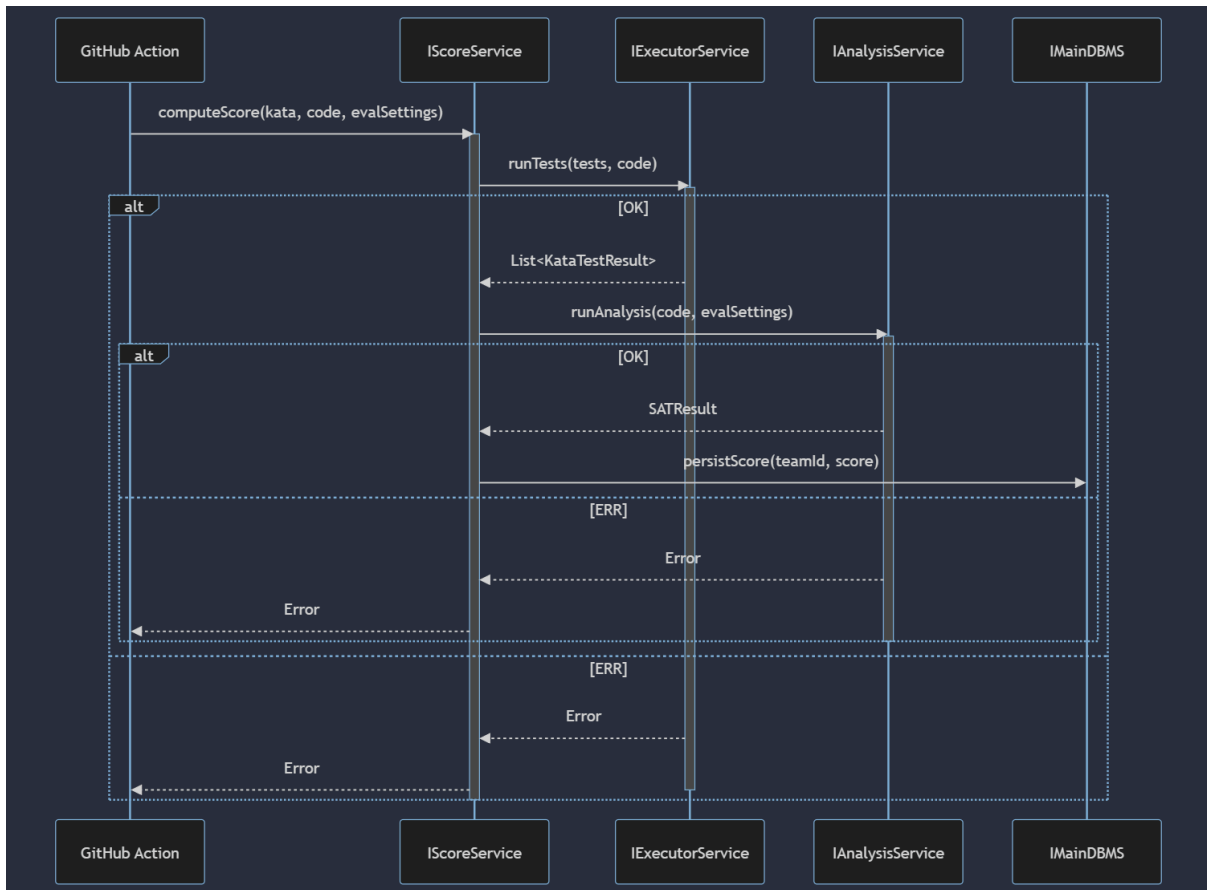
In this section, there is a list of important use cases and the corresponding sequence diagram to perform them using the previously described component interfaces.

All sequence diagrams are made using Mermaid. As such, it is possible to render them graphically using the appropriate command line application.

UC: Calculate Total Score

```
sequenceDiagram
    participant GHA as GitHub Action
    participant ISS as IScoreService
    participant IES as IExecutorService
    participant IAS as IAnalysisService
    participant IMDBMS as IMainDBMS

    GHA->>ISS: computeScore(kata, code, evalSettings)
    activate ISS
    ISS->>IES: runTests(tests, code)
    activate IES
    alt OK
        IES-->>ISS: List<KataTestResult>
        ISS->>IAS: runAnalysis(code, evalSettings)
        activate IAS
        alt OK
            IAS-->>ISS: SATResult
            ISS->>IMDBMS: persistScore(teamId, score)
        else ERR
            IAS-->>ISS: Error
            ISS-->>GHA: Error
        end
        deactivate IAS
    else ERR
        IES-->>ISS: Error
        ISS-->>GHA: Error
        deactivate IES
    end
    deactivate ISS
```

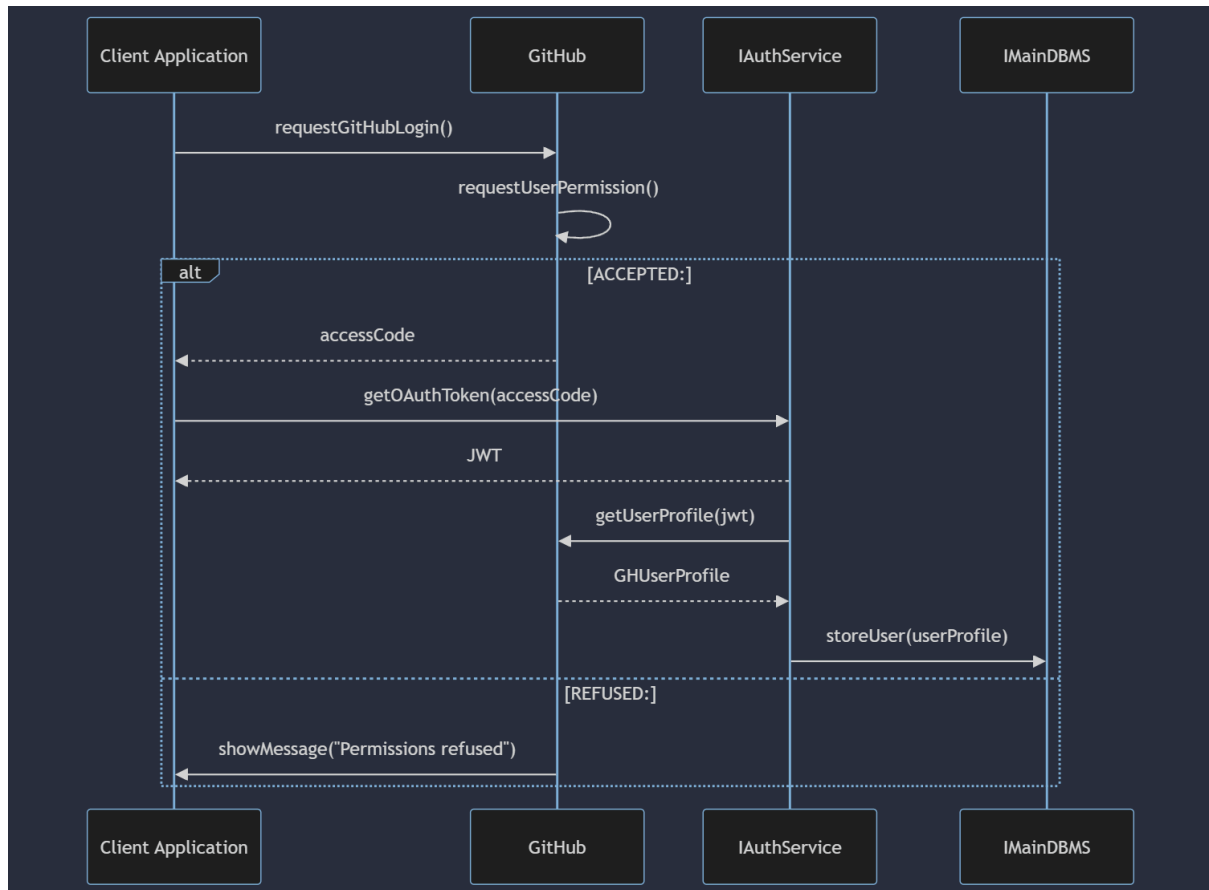


UC: Registration

sequenceDiagram

```

participant CA as Client Application
participant GH as GitHub
participant IAUTH as IAuthService
participant IMDBMS as IMainDBMS
CA->>GH: requestGitHubLogin()
GH->>GH: requestUserPermission()
alt ACCEPTED:
    GH-->>CA: accessCode
    CA->>IAUTH: getOAuthToken(accessCode)
    IAUTH-->>CA: JWT
    IAUTH->>GH: getUserProfile(jwt)
    GH-->>IAUTH: GHUserProfile
    IAUTH->>IMDBMS : storeUser(userProfile)
else REFUSED:
    GH->>CA: showMessage("Permissions refused")
end
  
```



UC: Visualize User Profile

sequenceDiagram

```

participant CA as Client Application
participant IUS as IUserService
participant IMDBMS as IMainDBMS
CA->>+IUS: getUser(userId)
IUS->>+IMDBMS: getUser(userId)
IMDBMS-->>-IUS: User
IUS-->>-CA: User
CA->>+IUS: getBadges(userId)
IUS->>+IMDBMS: getBadges(userId)
IMDBMS-->>-IUS: List<Badge>
IUS-->>-CA: List<Badge>
CA->>+IUS: getFriends(userId)
IUS->>+IMDBMS: getFriends(userId)
IMDBMS-->>-IUS: List<User>
IUS-->>-CA: List<User>
CA->>+IUS: getTournamentsParticipating(userId)
IUS->>+IMDBMS: getTournamentsParticipating(userId)
IMDBMS-->>-IUS: List<Tournament>
IUS-->>-CA: List<Tournament>
  
```

```

CA->>+IUS: getTournamentsCoordinating(userId)
IUS->>+IMDBMS: getTournamentsCoordinating(userId)
IMDBMS-->>-IUS: List<Tournament>
IUS-->>-CA: List<Tournament>

```



UC: Send Friend Request

sequenceDiagram

```

participant CAs as Client Application
participant INS as INotificationService

```



```
CAs->>INS: sendFriendRequestNotify(senderId, receiverId)
```



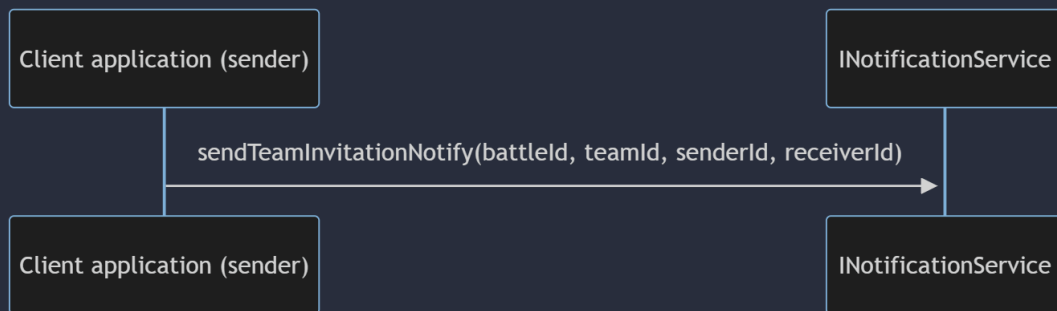
UC: Send team request

```
sequenceDiagram
```

```
participant CAs as Client application (sender)
```

```
participant INS as INotificationService
```

```
CAs->>+INS: sendTeamInvitationNotify(battleId, teamId, senderId, receiverId)
```



UC: Accept/Reject team request

sequenceDiagram

```
participant CAr as Client application (receiver)
participant INS as INotificationService
participant ITS as ITournamentService
participant IMDBMS as IMainDBMS
```

```
CAr->>+INS: openNotification(userId)
INS->>+ CAr: List<Notification>
CAr->>+INS: interactTeamInvitationNotify(reciverId, notificationId, accepted)
alt accepted == true:
    INS->>+ITS: joinBattle(notification)
    ITS->>+IMDBMS: isFullTeam(teamId)
    IMDBMS-->>+ITS: outcome
    alt outcome == true:
        ITS->>+INS: exceptionTournamentFull
        INS->>+CAr: exceptionTournamentFull
    else

```

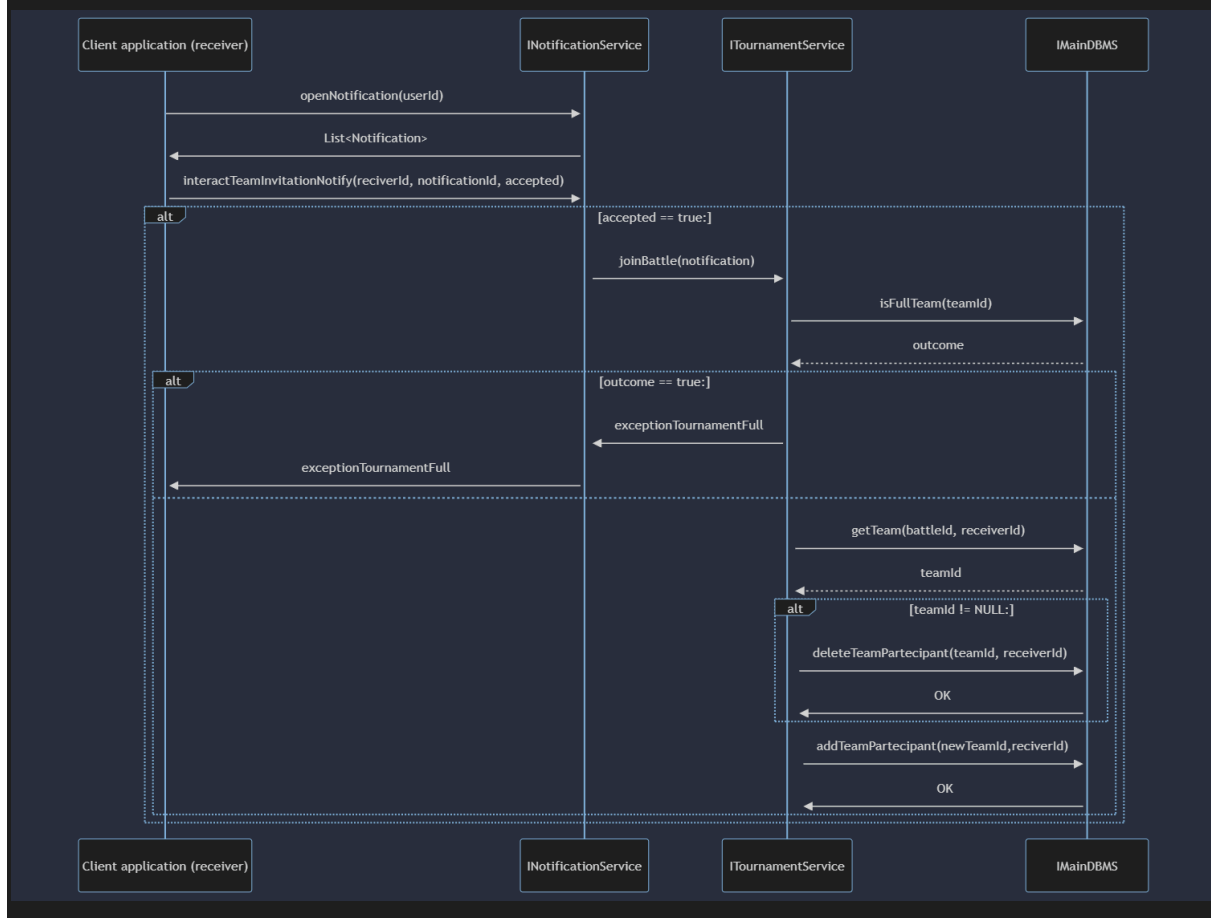
```

ITS->>+IMDBMS: getTeam(battleId, receiverId)
IMDBMS-->>+ITS: teamId
alt teamId != NULL:
    ITS->>+IMDBMS: deleteTeamPartecipant(teamId,
receiverId)

    IMDBMS-->>+ITS: OK
end

ITS->>+IMDBMS: addTeamPartecipant(newTeamId,reciverId)
IMDBMS-->>+ITS: OK
end
end

```



This sequence diagram shows the process of accepting or rejecting an invitation to a team. The interaction with the notification happens through the method `interactTeamInvitation` exposed by the interface of the Notification Service. If the request is accepted, the Notification Service calls the method `JoinBattle` of the Tournament Service, passing the notification. The tournament service immediately checks if the Team is already full (in this case an exception is thrown). Then checks if the user is already in a team, using the method `getTeam` exposed by the interface of the MainDBMS. If the return value of the method is True, then the user leaves the current Team, and in any case it is added to the new team. With an internal operation, Notification Service deletes the notification.

UC: Join battle via notification

sequenceDiagram

```
participant CA as Client application
participant INS as INotificationService
participant ITS as ITournamentService
participant IMDBMS as IMainDBMS
```

```
CA->>+INS: openNotification(userId)
```

```
INS->>+ CA: List<Notification>
```

```
CA->>+INS: interactBattleCreationNotify(receiverId, notificationId,
accepted)
```

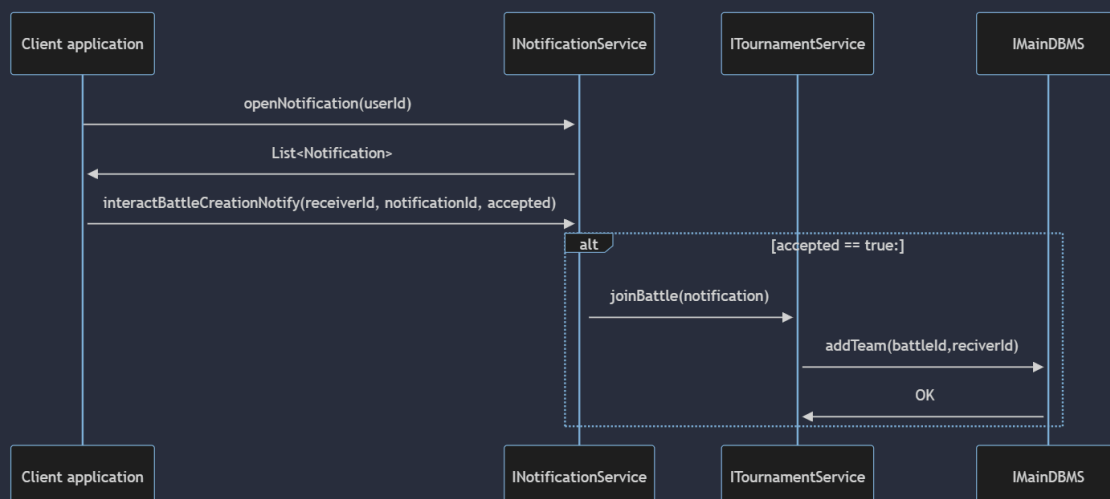
```
alt accepted == true:
```

```
    INS->>+ITS: joinBattle(notification)
```

```
    ITS->>+IMDBMS: addTeam(battleId,reciverId)
```

```
    IMDBMS->>+ITS: OK
```

```
end
```



This sequence diagram shows the process of joining a tournament through notification. The first step is for the Client Application to get the notification list using the method openNotification. Then the Client Application calls

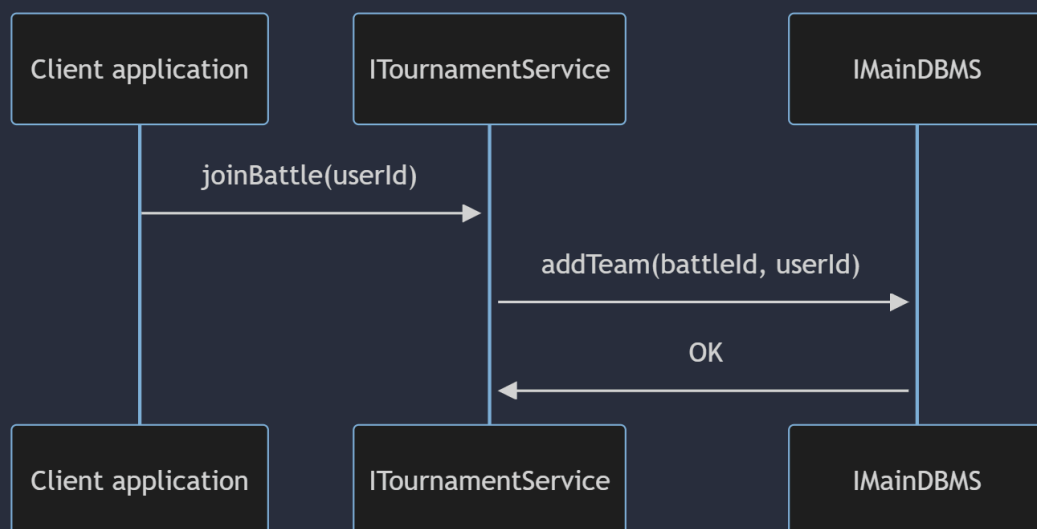
interactBattleCreationNotify to accept or reject the request. If accepted, the user is added to the battle through the cascade calling of joinBattle and addTeam. The notification is then eliminated by the Notification service in an inner operation.

UC: Join battle via webapp

sequenceDiagram

```
participant CA as Client application
participant ITS as ITournamentService
participant IMDBMS as IMainDBMS

CA->>ITS: joinBattle(userId)
ITS->>IMDBMS: addTeam(battleId, userId)
IMDBMS->>ITS: OK
```



UC: Join tournament via notification

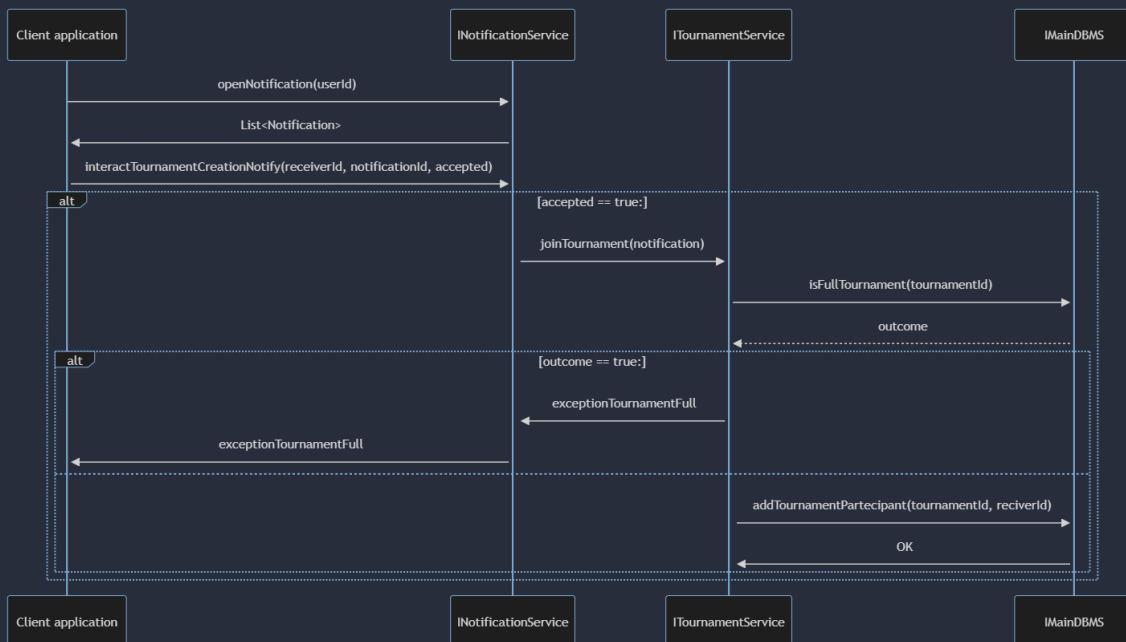
sequenceDiagram

```
participant CA as Client application
participant INS as INotificationService
participant ITS as ITournamentService
participant IMDBMS as IMainDBMS
```

```

CA->>+INS: openNotification(userId)
INS->>+ CA: List<Notification>
CA->>+INS: interactTournamentCreationNotify(receiverId, notificationId,
accepted)
alt accepted == true:
    INS->>+ITS: joinTournament(notification)
    ITS->>+IMDBMS: isFullTournament(tournamentId)
    IMDBMS-->>+ITS: outcome
    alt outcome == true:
        ITS->>+INS: exceptionTournamentFull
        INS->>+CA: exceptionTournamentFull
    else
        ITS->>+IMDBMS: addTournamentPartecipant(tournamentId,
receiverId)
        IMDBMS->>+ITS: OK
    end
end
end

```



UC: Join tournament via webapp or link p1

sequenceDiagram

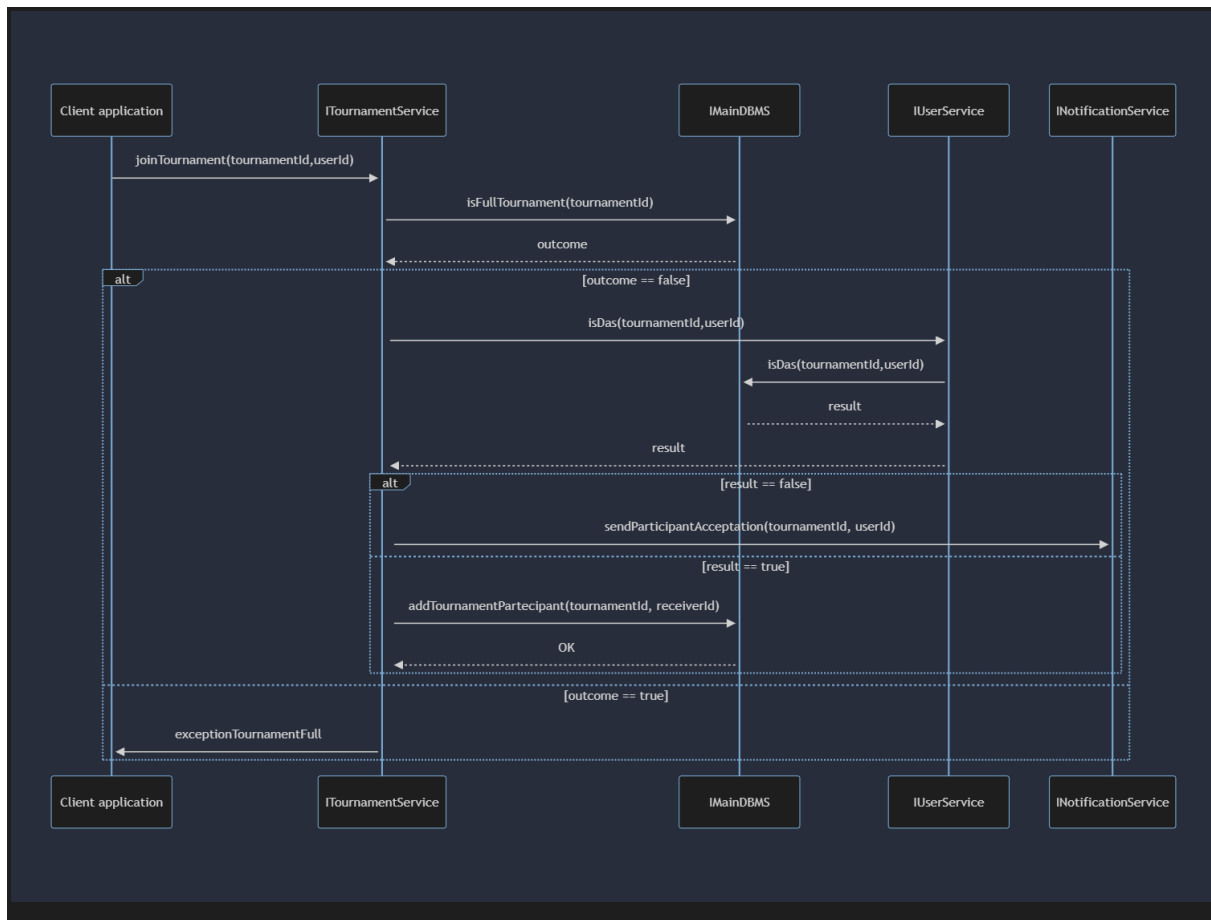
participant CAs as Client application (student)

```

participant ITS as ITournamentService
participant IMDBMS as IMainDBMS
participant INS as INotificationService
participant CAo as Client application (OCT)

CAs->>+ITS: joinTournament(tournamentId,userId)
ITS->>+IMDBMS : isFullTournament(tournamentId)
IMDBMS-->>+ITS: outcome
alt outcome == false
    ITS->>+IUS: isDas(tournamentId,userId)
    IUS->>+IMDBMS: isDas(tournamentId,userId)
    IMDBMS-->>+IUS: result
    IUS-->>+ITS: result
    alt result == false
        ITS->>+INS: sendParticipantAcceptation(tournamentId, userId)
    else result == true
        ITS->>+IMDBMS: addTournamentPartecipant(tournamentId,
receiverId)
        IMDBMS->>+ITS: OK
    end
else outcome == true
    ITS->>+CAs: exceptionTournamentFull
end

```



When a user tries to join a tournament via webapp or link they can access directly, if they are a DAS for the tournament. Otherwise, the OCT of the tournament must accept the user. This verification is made through the cascade calls of the methods isDas.

UC: Join tournament via webapp or link p2

```

sequenceDiagram
    participant CA as Client application
    participant INS as INotificationService
    participant ITS as ITournamentService
    participant IMDBMS as IMainDBMS

    CA->>INS: interactParticipantAcceptation(reciverId,notificationId,accepted)

    alt NotificationOK
    alt accepted == true:
        INS->>ITS: joinTournament(notification)
        ITS->>IMDBMS: addTournamentPartecipant(tournamentId, receiverId)

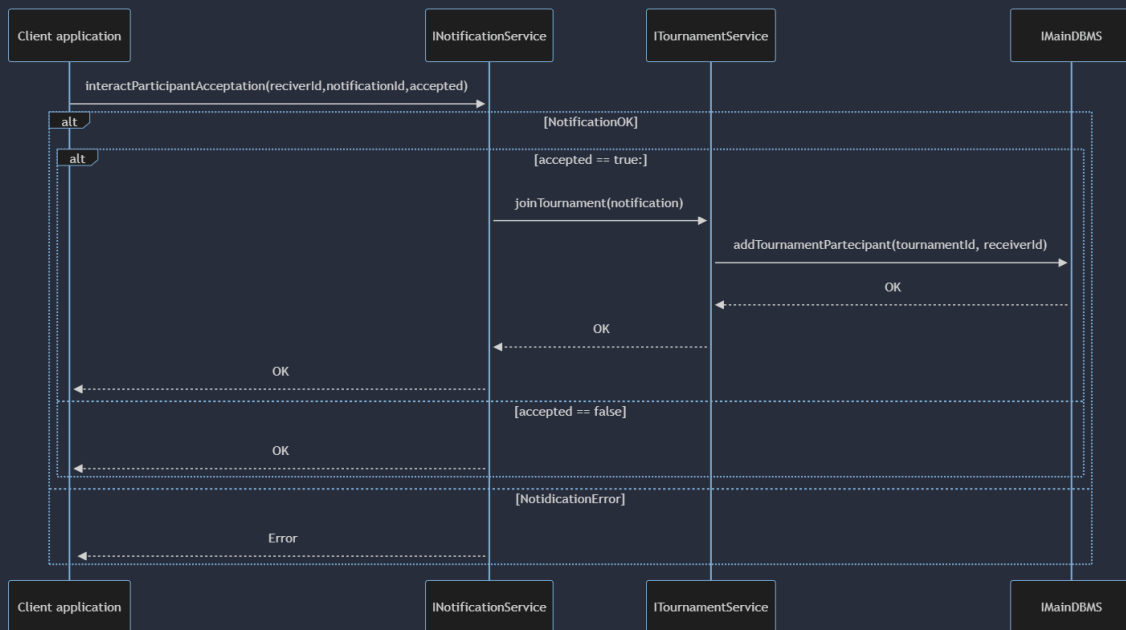
        IMDBMS-->>ITS: OK
        ITS-->>INS: OK
        INS-->>CA: OK
    end
  
```



```

        else accepted == false
            INS-->>CA: OK
        end
        else NotidicationError
            INS-->>CA:Error
        end
    end
end

```



The system return Error in Client Application when the client accepts a notification that aren't allowed to accept\reject.

UC: createTournament

```

sequenceDiagram
    participant CA as Client application
    participant ITS as ITournamentService
    participant IMDBMS as IMainDBMS
    participant INS as INotificationService
    participant IUS as IUserService

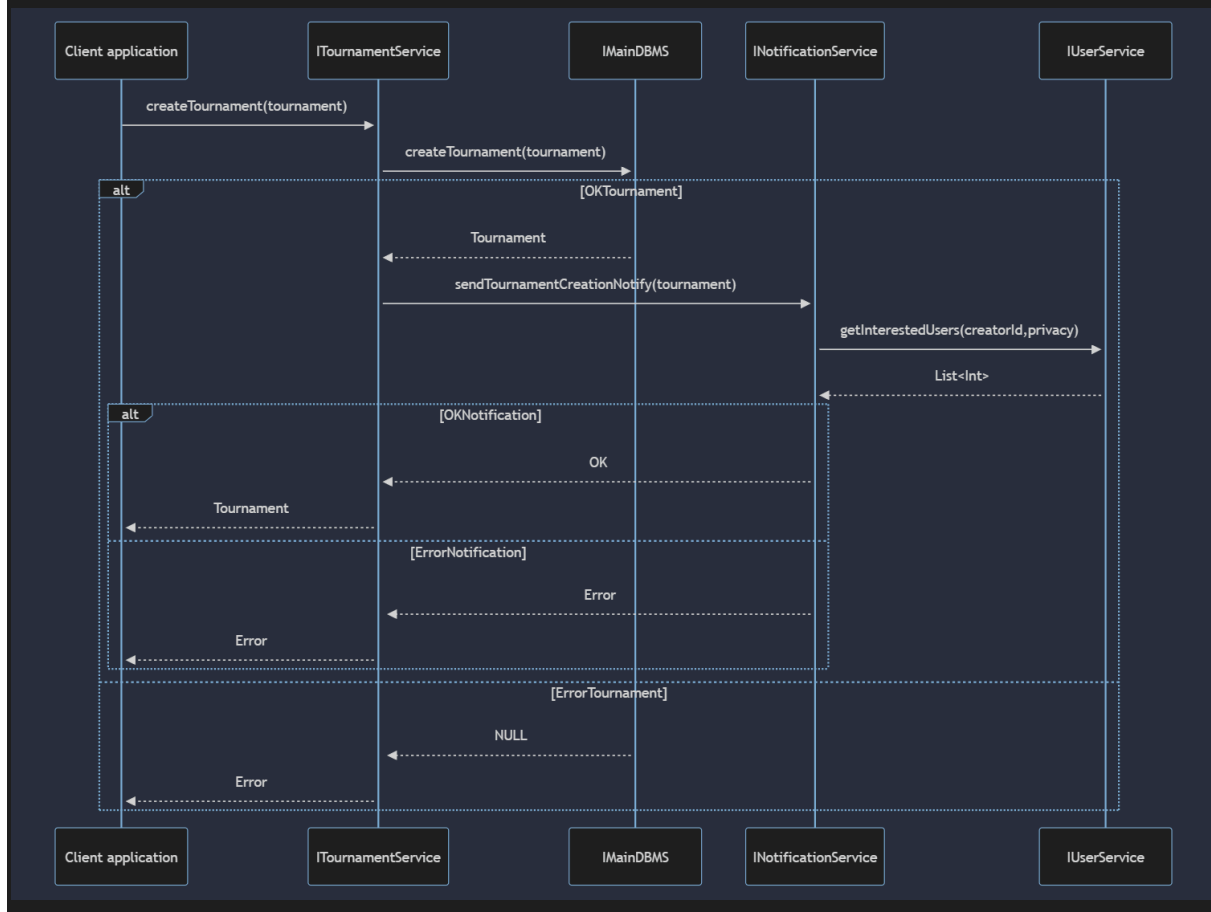
    CA->>ITS: createTournament(tournament)
    ITS->>IMDBMS: createTournament(tournament)
    alt OKTournament

```

```

IMDBMS-->>ITS: Tournament
ITS-->>INS: sendTournamentCreationNotify(tournament)
INS-->>IUS: getInterestedUsers(creatorId,privacy)
IUS-->>INS: List<Int>
alt OKNotification
INS-->>ITS: OK
ITS-->>CA: Tournament
else ErrorNotification
INS-->>ITS: Error
ITS-->>CA: Error
end
else ErrorTournament
IMDBMS-->>ITS: NULL
ITS-->>CA: Error
end

```



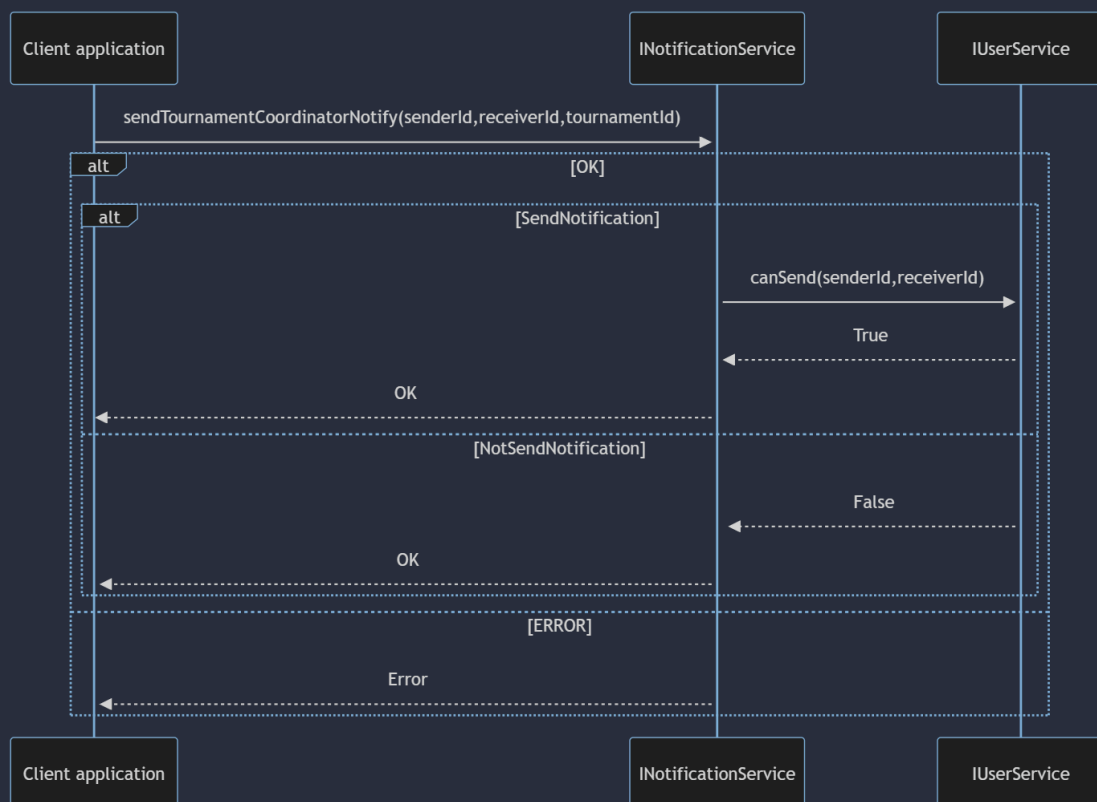
The creation of the tournament is made through the cascade calls of the methods createTournament, while the rest of the sequence diagram models the sending of the notifications for the creation of the tournament.

UC: AddingCoordinator

```

sequenceDiagram
    participant CA as Client application
    participant INS as INotificationService
    participant IUS as IUserService
    CA->>+INS:
sendTournamentCoordinatorNotify(senderId,receiverId,tournamentId)
    alt OK
        alt SendNotification
            INS->>+IUS: canSend(senderId,receiverId)
            IUS-->>+INS: True
            INS-->>CA: OK
        else NotSendNotification
            IUS-->>+INS: False
            INS-->>CA: OK
        end
    else ERROR
        INS-->>CA: Error
    end
end

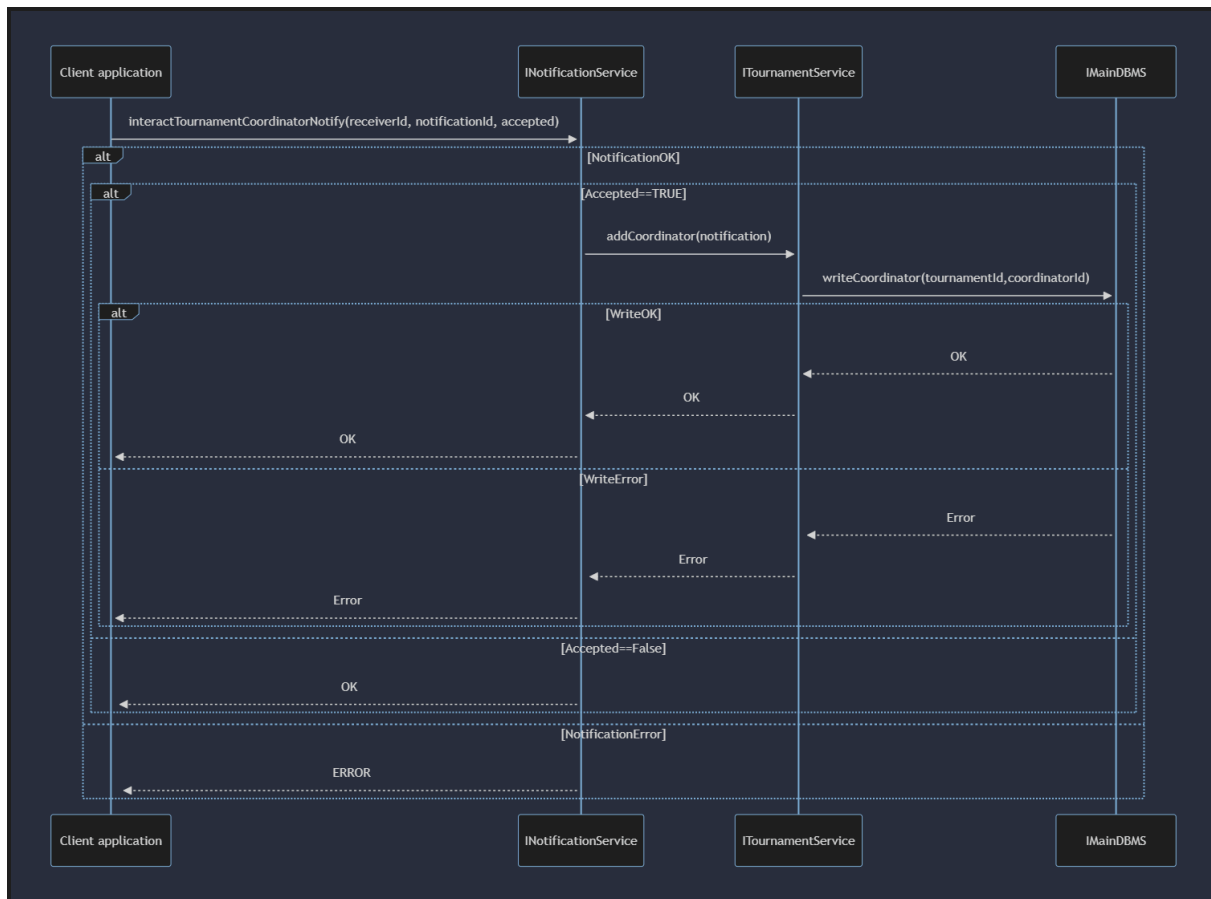
```



The notification doesn't arrive if the sender wants send notification that aren't allowed to send, or when the receiver are a policy that don't allow received the notification.

UC: Accept/Reject TC request

```
sequenceDiagram
    participant CA as Client application
    participant INS as INotificationService
    participant ITS as ITournamentService
    participant IMDBMS as IMainDBMS
    CA->>+INS: interactTournamentCoordinatorNotify(receiverId,
notificationId, accepted)
    alt NotificationOK
        alt Accepted==TRUE
            INS->>+ITS:addCoordinator(notification)
            ITS->>+IMDBMS:writeCoordinator(tournamentId,coordinatorId)
            alt WriteOK
                IMDBMS-->>+ITS:OK
                ITS-->>+INS:OK
                INS-->>+CA:OK
            else WriteError
                IMDBMS-->>+ITS:Error
                ITS-->>+INS:Error
                INS-->>+CA:Error
            end
        else Accepted==False
            INS-->>+CA:OK
        end
    else NotificationError
        INS-->>+CA:ERROR
    end
end
```



UC: Accept/reject friend request

sequenceDiagram

```

sequenceDiagram
    participant CA as Client application
    participant INS as INotificationService
    participant IUS as IUserService
    participant IMDBMS as IMainDBMS

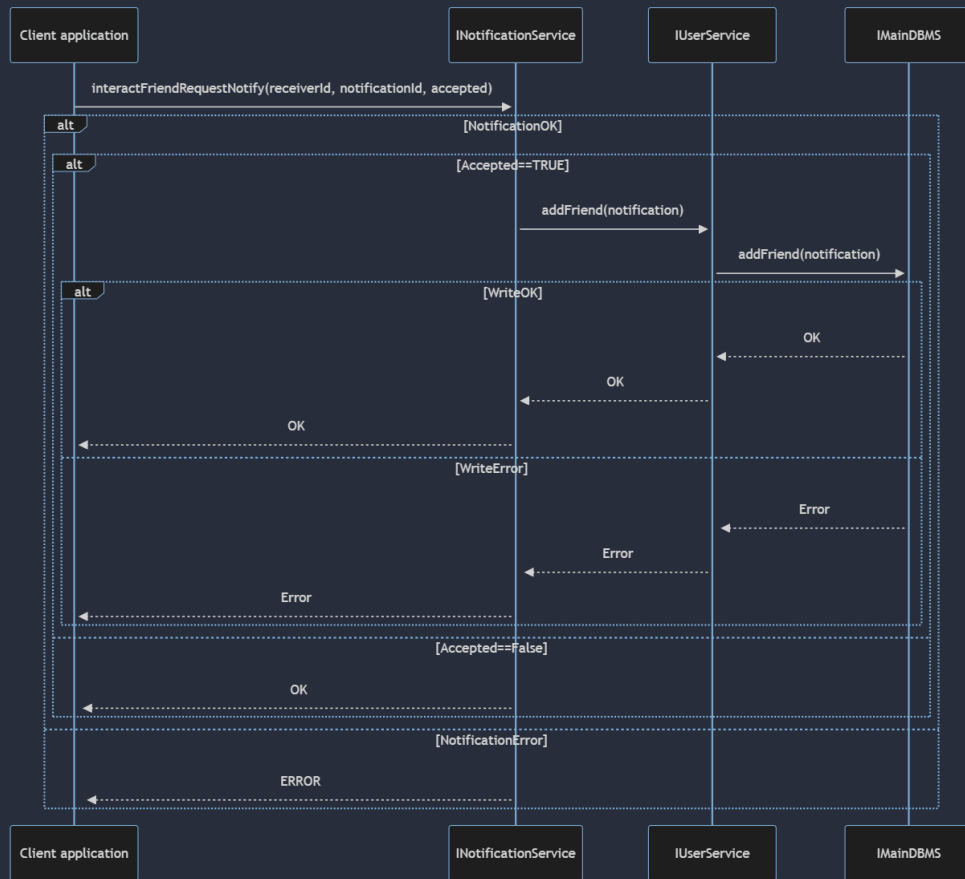
    CA->>INS: interactFriendRequestNotify(receiverId, notificationId,
accepted)
    alt NotificationOK
        alt Accepted==TRUE
            INS->>IUS: addFriend(notification)
            IUS->>IMDBMS: addFriend(notification)
            alt WriteOK
                IMDBMS-->>IUS: OK
                IUS-->>INS: OK
                INS-->>CA: OK
            else WriteError
                IMDBMS-->>IUS: Error
                IUS-->>INS: Error
                INS-->>CA: Error
            end
        else Accepted==False
            INS-->>CA: ERROR
        end
    end

```

```

    INS-->>+CA:OK
  end
else NotificationError
  INS-->>+CA:ERROR
end

```



Like Accept/Reject TC request

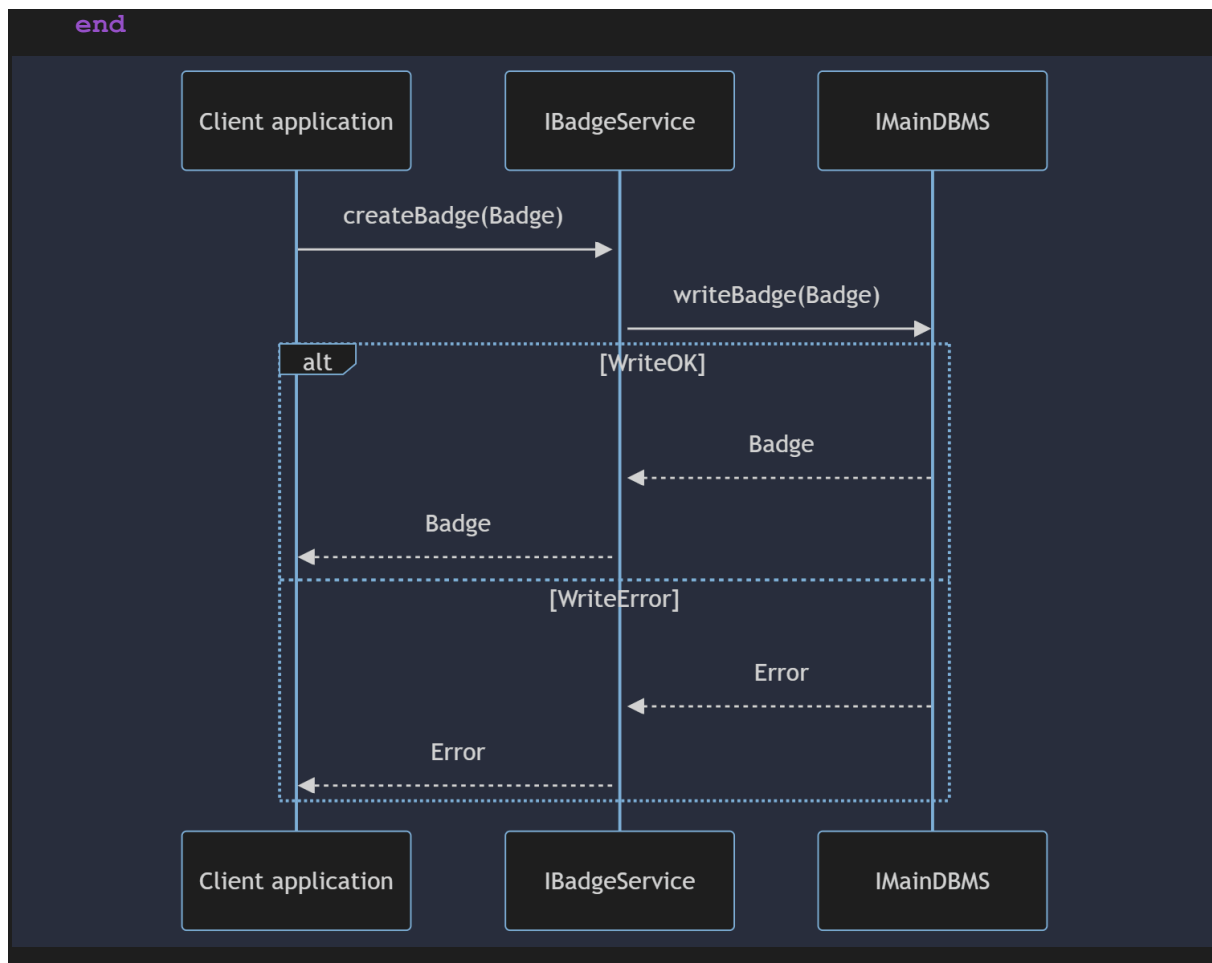
UC: Create Badge

sequenceDiagram

```

participant CA as Client application
participant IBdgS as IBadgeService
participant IMDBMS as IMainDBMS
CA->>+IBdgS: createBadge(Badge)
IBdgS->>IMDBMS: writeBadge(Badge)
alt WriteOK
  IMDBMS-->>IBdgS: Badge
  IBdgS-->>CA: Badge
else WriteError
  IMDBMS-->>IBdgS: Error
  IBdgS-->>CA: Error
end

```

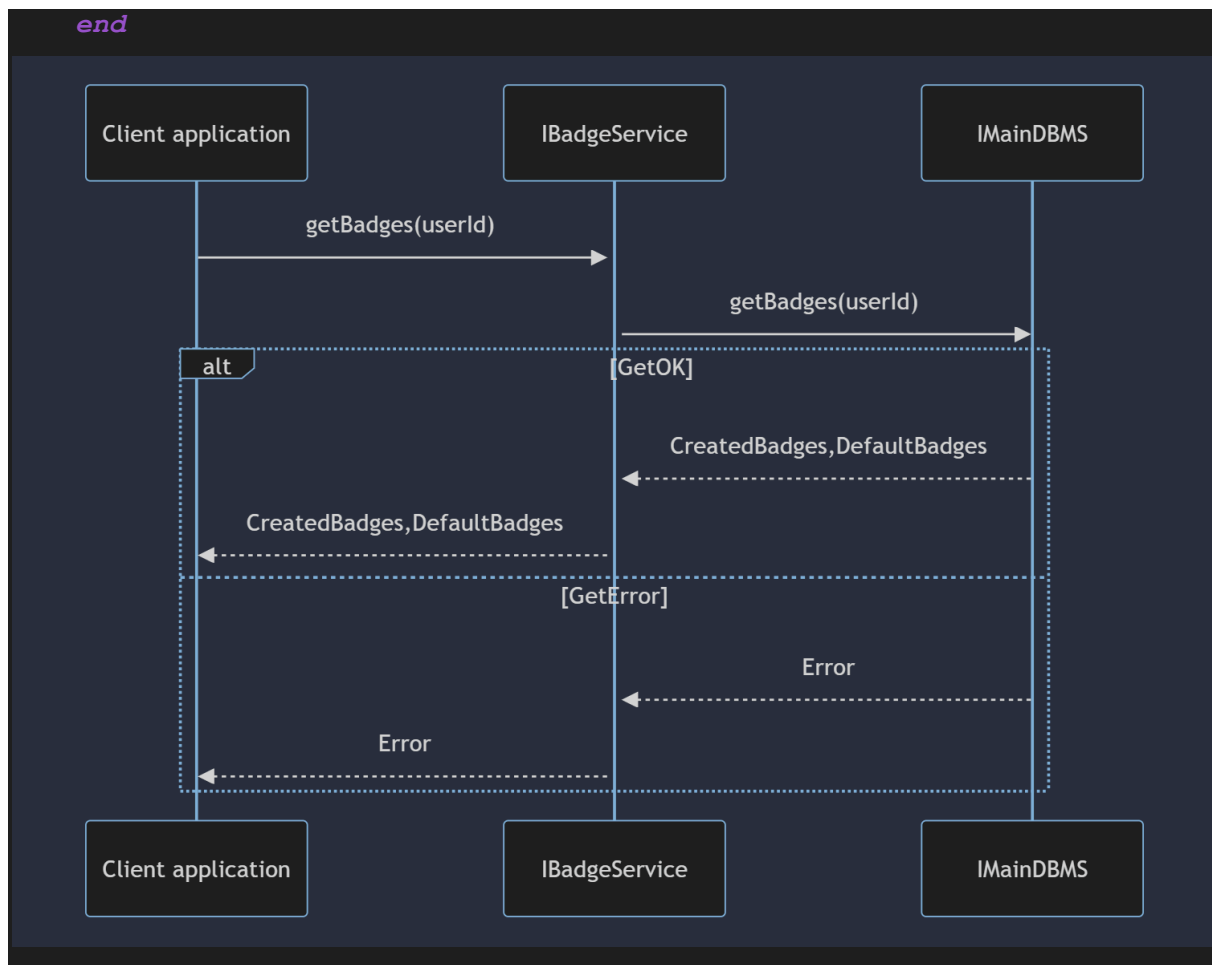


UC: getBadge

sequenceDiagram

```

participant CA as Client application
participant IBdgS as IBadgeService
participant IMDBMS as IMainDBMS
CA->>+IBdgS: getBadges(userId)
IBdgS->>IMDBMS: getBadges(userId)
alt GetOK
IMDBMS-->>IBdgS: CreatedBadges,DefaultBadges
IBdgS-->>CA: CreatedBadges,DefaultBadges
else GetError
IMDBMS-->>IBdgS: Error
IBdgS-->>CA: Error
end
  
```

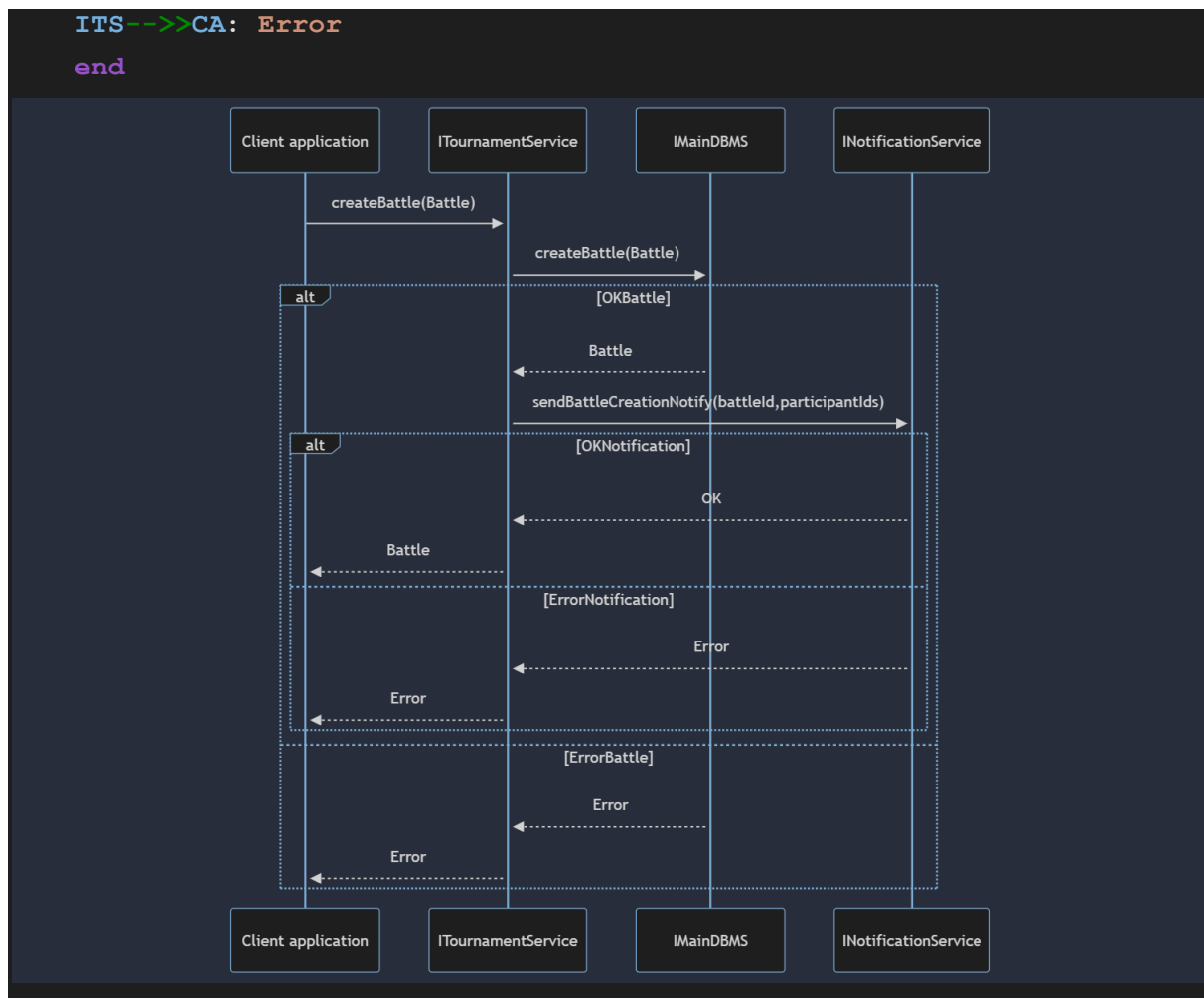


UC: CreateBattle

sequenceDiagram

```

participant CA as Client application
participant ITS as ITournamentService
participant IMDBMS as IMainDBMS
participant INS as INotificationService
CA->>+ITS: createBattle(Battle)
ITS->>+IMDBMS: createBattle(Battle)
alt OKBattle
IMDBMS-->>ITS: Battle
ITS->>+INS: sendBattleCreationNotify(battleId,participantIds)
alt OKNotification
INS-->>ITS: OK
ITS-->>+CA: Battle
else ErrorNotification
INS-->>ITS: Error
ITS-->>CA: Error
end
else ErrorBattle
IMDBMS-->>ITS: Error
  
```

The creation of the battle is made through the cascade calls of the methods createBattle, while the rest of the sequence diagram models the sending of the notifications for the creation of the battle. The ErrorBattle refers to the case where the user is not authorized to create the battle

UC: Perform OME

sequenceDiagram

```

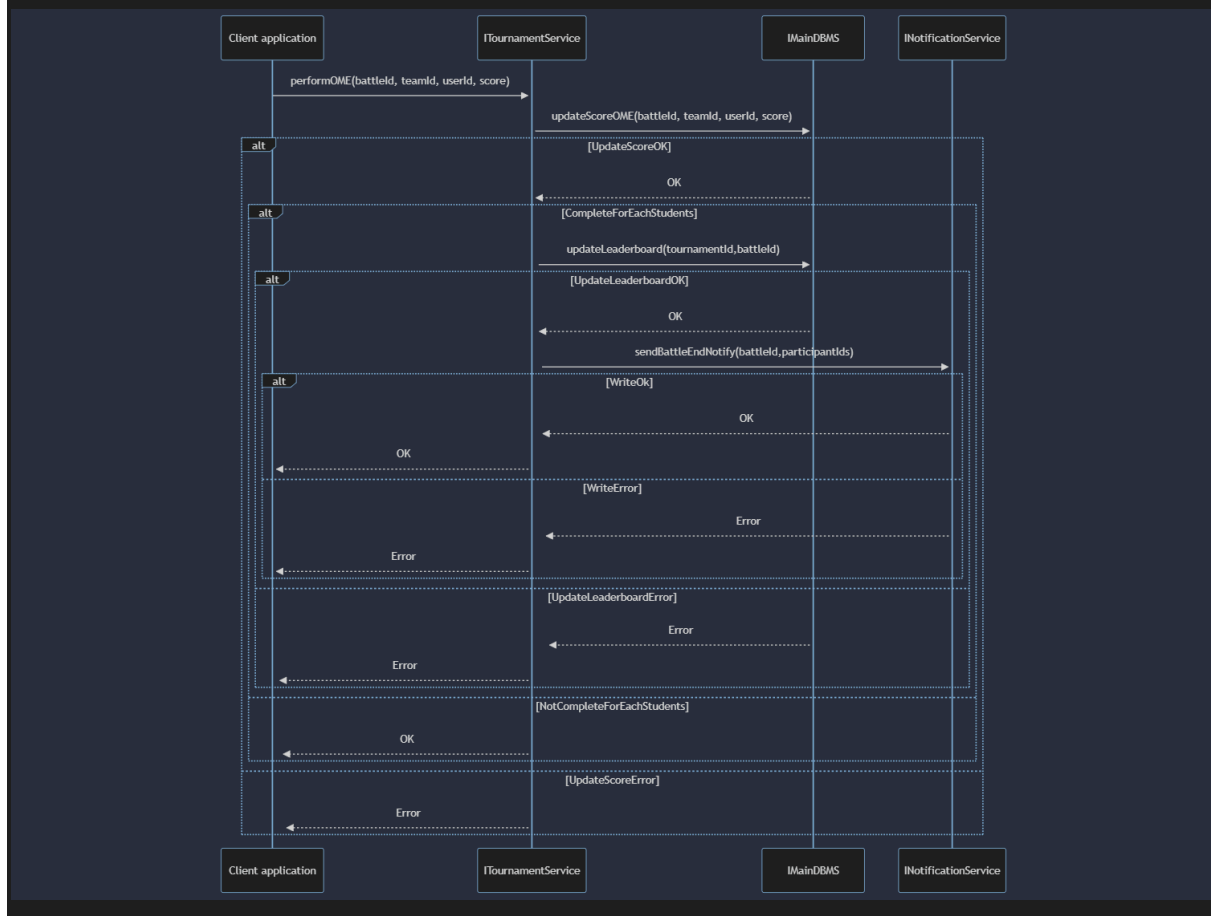
sequenceDiagram
    participant CA as Client application
    participant ITS as ITournamentService
    participant IMDBMS as IMainDBMS
    participant INS as INotificationService

    CA->>ITS: performOME(battleId, teamId, userId, score)
    ITS->>IMDBMS: updateScoreOME(battleId, teamId, userId, score)
    alt UpdateScoreOK
        IMDBMS-->>ITS: OK
    alt CompleteForEachStudents
        ITS->>IMDBMS: updateLeaderboard(tournamentId,battleId)
        alt UpdateLeaderboardOK
            IMDBMS-->>ITS: OK
        end
    end
  
```

```

ITS-->>+INS:sendBattleEndNotify(battleId,participantIds)
alt WriteOk
INS-->>+ITS:OK
ITS-->>+CA: OK
else WriteError
INS-->>+ITS:Error
ITS-->>+CA: Error
end
else UpdateLeaderboardError
IMDBMS-->>+ITS: Error
ITS-->>+CA: Error
end
else NotCompleteForEachStudents
ITS-->>+CA: OK
end
else UpdateScoreError
ITS-->>+CA: Error
end
end

```



UC: Edit deadline

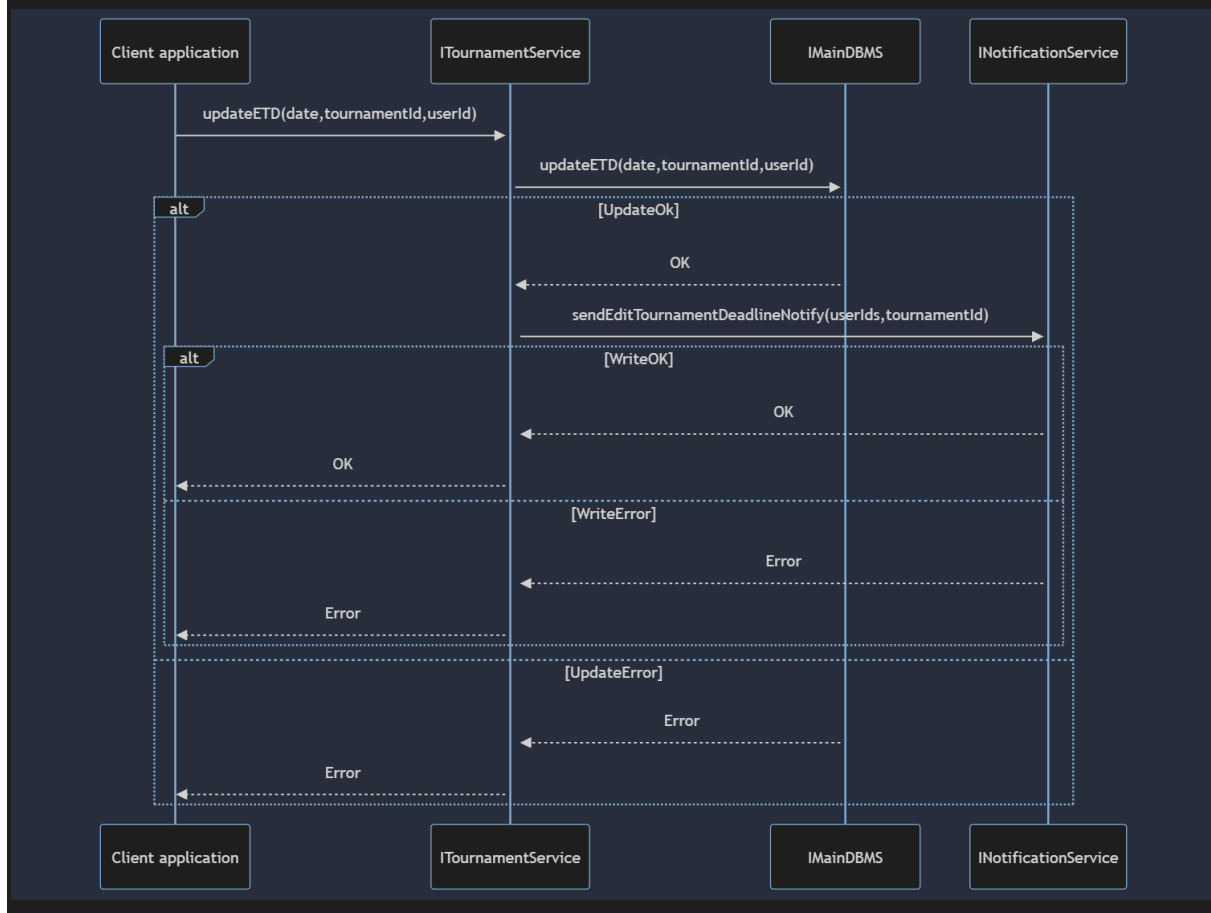
sequenceDiagram

participant CA as Client application

```

participant ITS as ITournamentService
participant IMDBMS as IMainDBMS
participant INS as INotificationService
CA->>+ITS: updateETD(date,tournamentId,userId)
ITS->>+IMDBMS: updateETD(date,tournamentId,userId)
alt UpdateOk
IMDBMS-->>+ITS: OK
ITS->>+INS: sendEditTournamentDeadlineNotify(userIds,tournamentId)
alt WriteOK
INS-->>ITS: OK
ITS-->>CA: OK
else WriteError
INS-->>ITS: Error
ITS-->>CA: Error
end
else UpdateError
IMDBMS-->>+ITS: Error
ITS-->>CA: Error
end
end

```



(This is ETD is equal for FTD, EBD and FBD)

UC: Ban Student

sequenceDiagram

participant CA as Client application

participant ITS as ITournamentService

participant IMDBMS as IMainDBMS

CA->>+ITS: banParticipant(clientId,participantId,tournamentId)

ITS->>+IMDBMS: deleteParticipant(clientId,participantId,tournamentId)

alt DeleteOk

IMDBMS-->>+ITS: OK

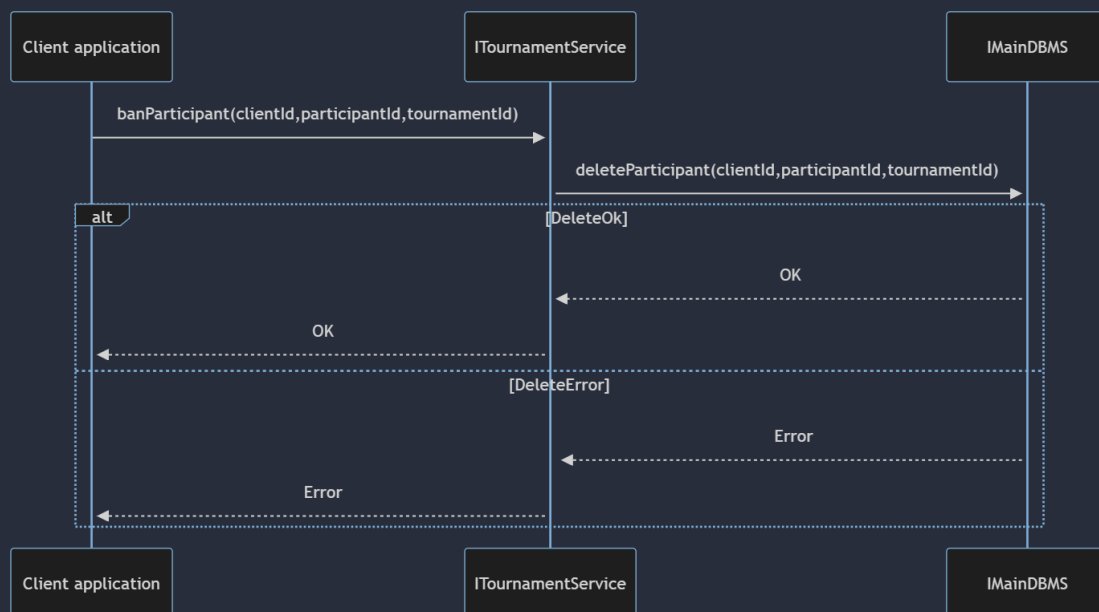
ITS-->>+CA: OK

else DeleteError

IMDBMS-->>+ITS: Error

ITS-->>+CA: Error

end



UC: FBD expired

sequenceDiagram

participant ModelService

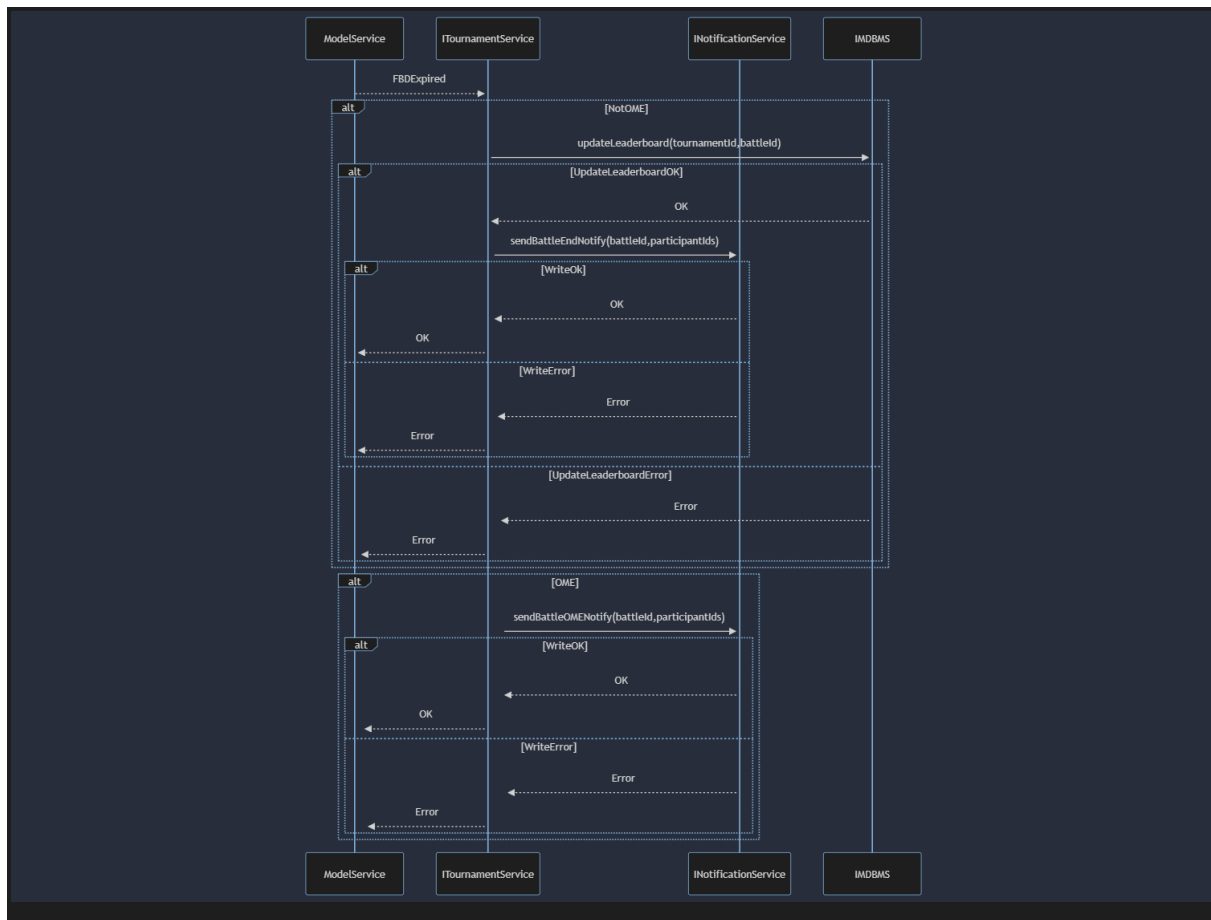
participant ITS as ITournamentService

participant INS as INotificationService

```

ModelService-->>+ITS: FBDEexpired
alt NotOME
ITS->>+IMDBMS:updateLeaderboard(tournamentId,battleId)
    alt UpdateLeaderboardOK
        IMDBMS-->>+ITS: OK
        ITS->>+INS:sendBattleEndNotify(battleId,participantIds)
            alt WriteOk
                INS-->>+ITS:OK
                ITS-->>+ModelService: OK
            else WriteError
                INS-->>+ITS:Error
                ITS-->>+ModelService: Error
            end
        else UpdateLeaderboardError
            IMDBMS-->>+ITS: Error
            ITS-->>+ModelService: Error
        end
    end
end
alt OME
ITS->>+INS:sendBattleOMENotify(battleId,participantIds)
    alt WriteOK
        INS-->>+ITS:OK
        ITS-->>+ModelService:OK
    else WriteError
        INS-->>+ITS:Error
        ITS-->>+ModelService:Error
    end
end
end

```



The behavior changes if the OME is present or not. In the first case both the creator and the participant are notified through the call of sendBattleOMENotify and no score is added to the tournament leaderboard through the call of updateLeaderboard. In the second case the tournament leaderboard is updated and the battle ends, so the end battle notification is sent through the call of sendBattleEndNotify.

UC: FTD expired so EndTournament

sequenceDiagram

```

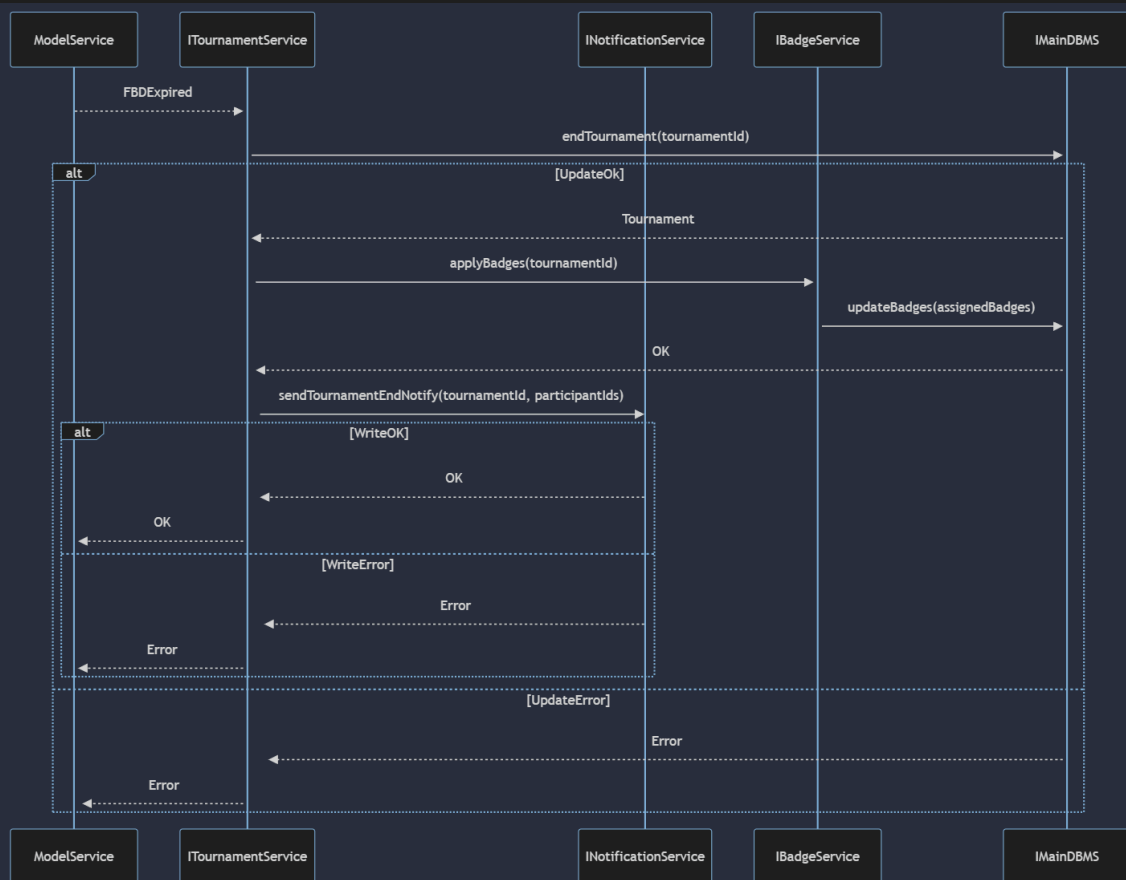
participant MS as ModelService
participant ITS as ITournamentService
participant INS as INotificationService
participant IBdgS as IBadgeService
participant IMDBMS as IMainDBMS
ModelService->>ITS: FBDEExpired
ITS->>IMDBMS: endTournament(tournamentId)
alt UpdateOk
IMDBMS->>ITS: Tournament
ITS->>IBdgS: applyBadges(tournamentId)
IBdgS->>IMDBMS: updateBadges(assignedBadges)
IMDBMS->>ITS: OK
ITS->>INS: sendTournamentEndNotify(tournamentId, participantIds)

```

```

alt WriteOK
INS-->>+ITS: OK
ITS-->>+ModelService:OK
else WriteError
INS-->>+ITS: Error
ITS-->>+ModelService:Error
end
else UpdateError
IMDBMS-->>+ITS:Error
ITS-->>+ModelService:Error
end

```



When the tournament FBD is expired the ModelService send a message with the tournamentId at the TournamentService, this with endTournament, close all battle ongoing and update the final leaderboard of the Tournament.

After that with applyBadges the BadgeService check all the rules of badge and assign them at the correct participants.

At the end with sendTournamentEndNotify the NotificationService send the End Tournament Notification at all participants in the tournaments.

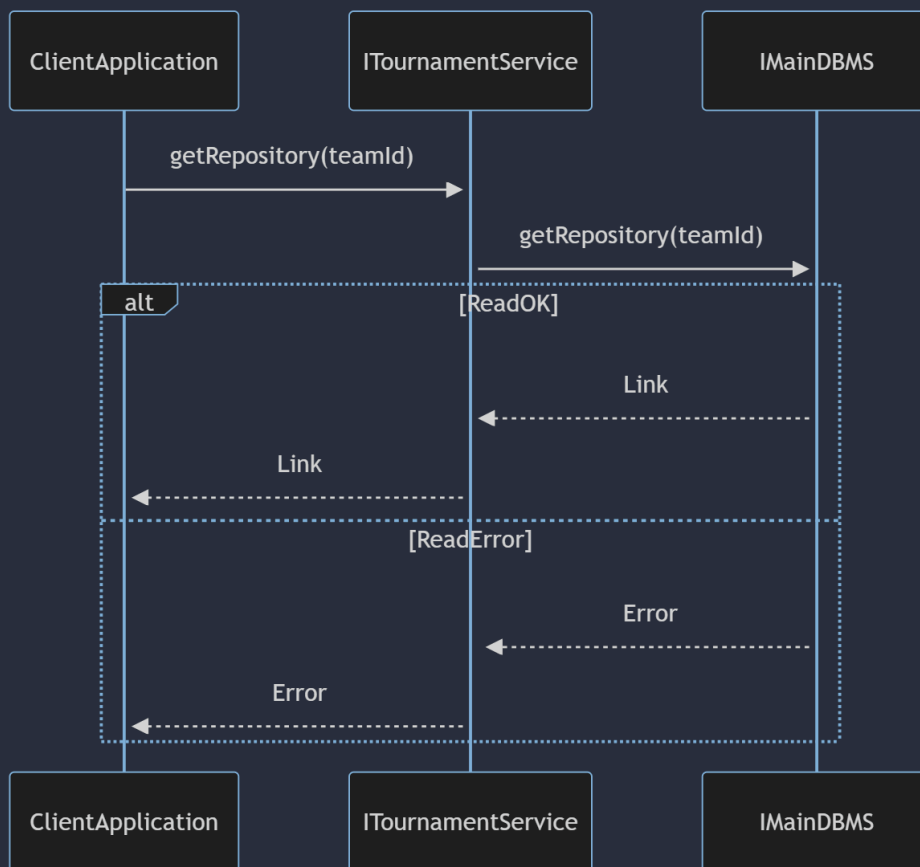
UC: Educator wants a repository link by team

sequenceDiagram

```

participant CA as ClientApplication
participant ITS as ITournamentService
participant IMDBMS as IMainDBMS
CA->>+ITS:getRepository(teamId)
ITS->>+IMDBMS:getRepository(teamId)
alt ReadOK
IMDBMS-->>+ITS:Link
ITS-->>+CA:Link
else ReadError
IMDBMS-->>+ITS:Error
ITS-->>+CA:Error
end
end

```



GitHub share repository link for a battle

sequenceDiagram

```

participant GHA as GitHub Action
participant ITS as ITournamentService
participant IMDBMS as IMainDBMS

GHA->>+ITS: addRepository(battleId, userId, link)

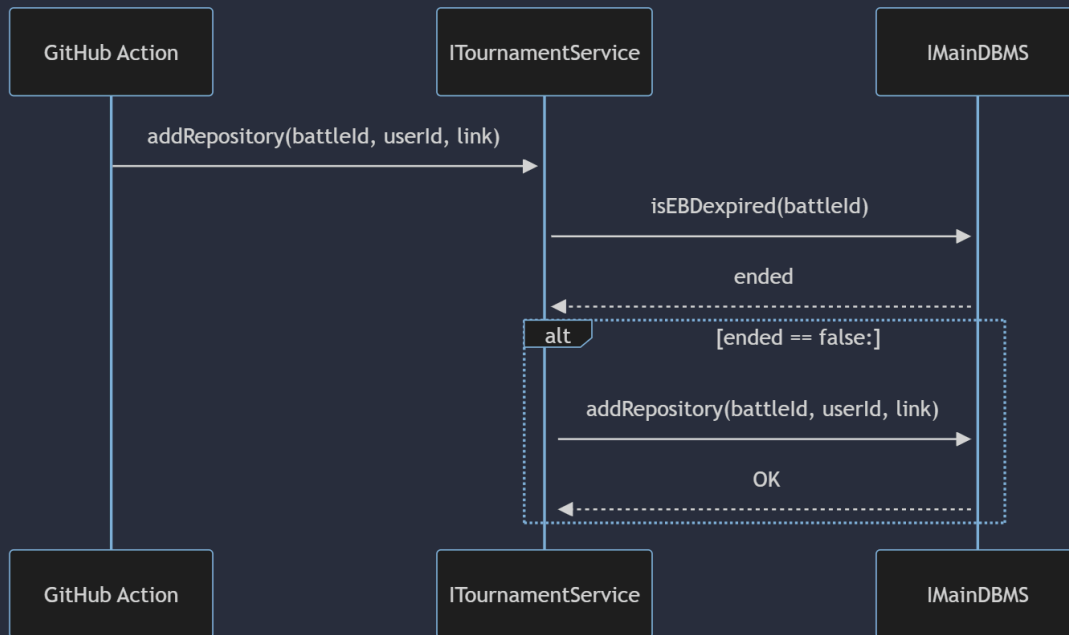
```



```

ITS->>IMDBMS : isEBDexpired(battleId)
IMDBMS-->>ITS: ended
alt ended == false:
    ITS->>IMDBMS: addRepository(battleId, userId, link)
    IMDBMS-->>ITS: OK
end

```



Accept/reject FriendRequest

2.6 - Selected Architectural Styles and Patterns

The main architectural style adopted for the system is Microservices Architecture. It was chosen for the following main reasons:

- **Scalability:** Microservices can be independently deployed on the cloud. This means that it is possible to dynamically scale the number of instances of each services, depending on the current demand of the users.
- **Availability:** If one or more services fail, the entire system keeps working while autoheal procedures are running to restore the failed services.
- **Maintainability:** Since each service is implemented separately in its own repository, this increases maintainability by reducing coupling and allows multiple teams to work in parallel.

- **Security:** Each microservice communicates with the others only through methods exposed on the interface, in this way a microservice can't access inner methods and parameters of other microservices, and can communicate with DBMSs only through the corresponding microservices.

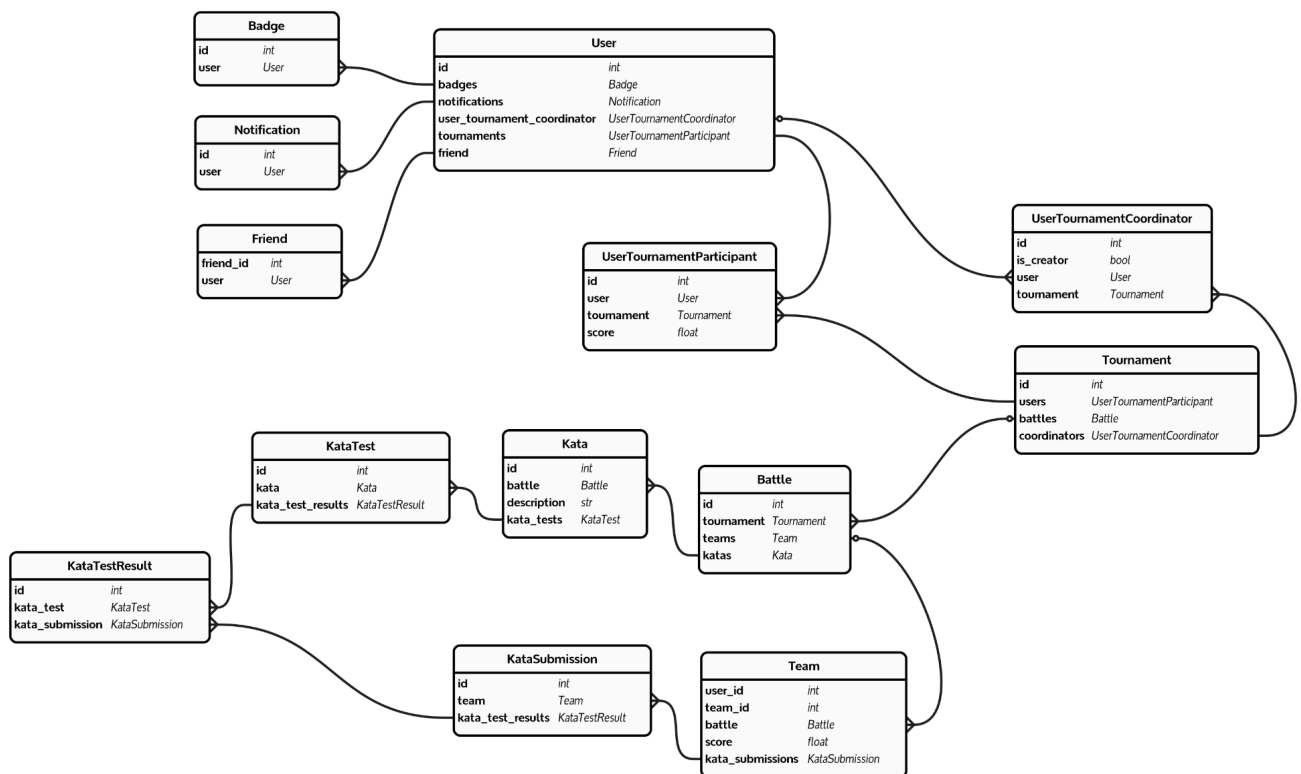
2.7 - Other Design Decisions

2.7.1 - API-first Development

The entire system must be usable only through the REST API interface. Before developing the frontend application, there will be a standard REST API available to be consumed in OpenAPI 3.0 format, as well as a documentation page (Swagger) to more easily explore and experiment with the endpoints. The frontend itself will just be a thin wrapper around the REST API, providing ease of use for the final user. This also implicitly allows for future integrations with other platforms to be implemented in an easier way, because the system is API-centric and does not even need the client to work properly.

2.7.2 - Entity-Relationship Diagram

We decided to build an Entity-Relationship Diagram (ERD) in order to clarify and accelerate the development of database schema. This ERD is not definitive and might change once the implementation gets going. Furthermore, for simplicity sake, this ERD does not have any property other than the relationships between entities. The updated ERD is available [here](#).



2.7.3 - Separate databases: Main DBMS and Notification DBMS

The Main DBMS will be one or more instances of PostgreSQL running with the same database schema. Notification DBMS will be one or more instances of MongoDB. This was decided because MongoDB is way cheaper to host, and notifications do not need to have a static schema enforced upon them. This decision was also made for scalability concerns, because MongoDB is capable of storing millions of records in a more economic and horizontally scalable manner.

2.7.4 - Sandboxing for arbitrary code execution

One of the main features of the application is automatic execution for the provided tests for a given battle based on submitted commits (called “entries”). This strictly implies RCE (Remote Code Execution) which is a common security vulnerability. This situation is unavoidable given the requirement.

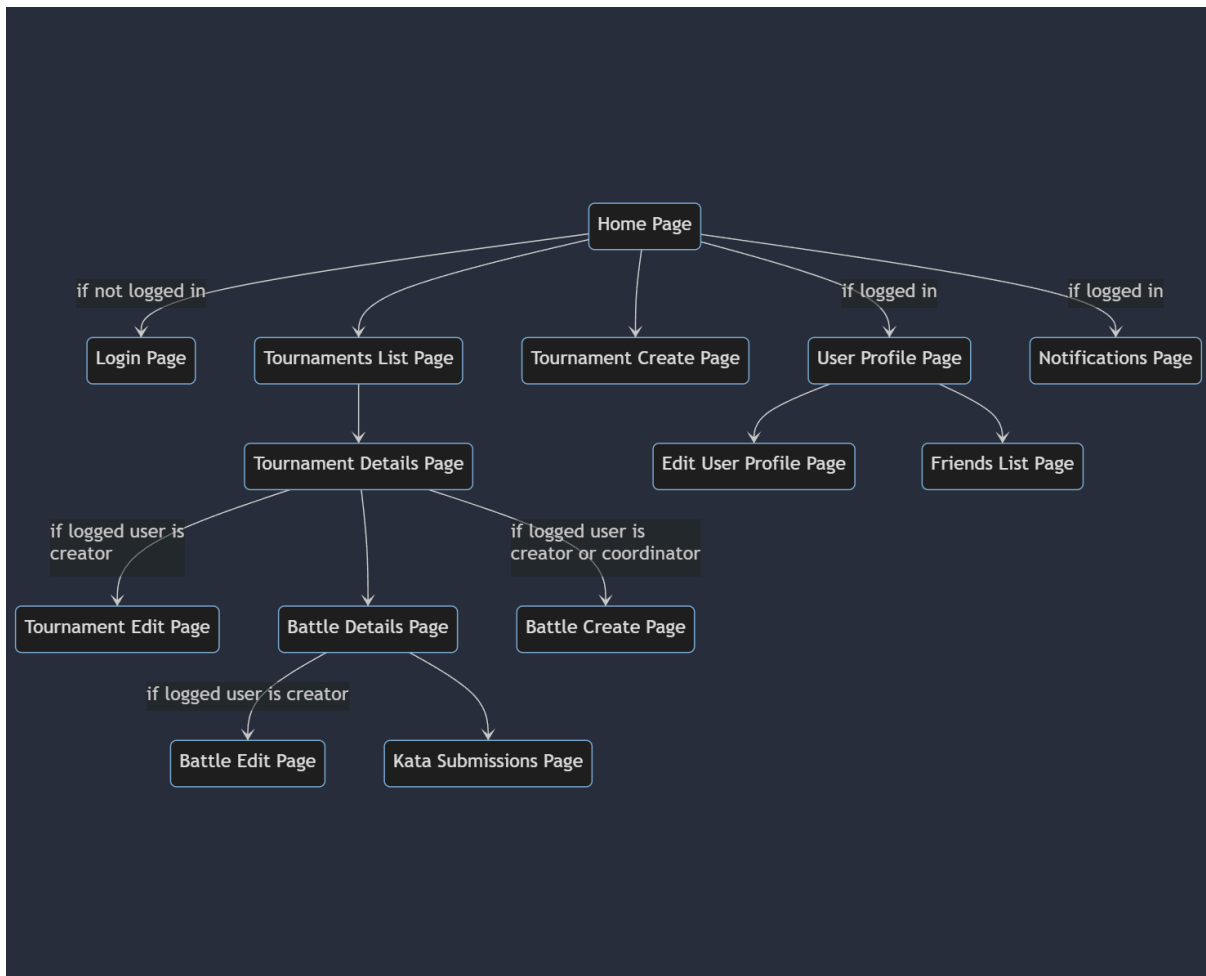
However, it is possible to reduce the attack surface by adopting a technique called “sandboxing”: the custom code submitted by participants is executed in a “sandboxed” environment (which is an execution environment completely separated and closed off from the main server where the backend is running).

3 - User Interface Design

This section is dedicated to presenting a general overview of the user interface of the CodeKataBattle system, as well as showing some mockups mainly focused on UX rather than UI, since that aspect will be handled by focus groups.

3.1 - Overview

The following is a state diagram indicating a general sitemap of the platform.



stateDiagram-v2

```
home: Home Page
login: Login Page
tournament_list: Tournaments List Page
tournament_create: Tournament Create Page
tournament_details: Tournament Details Page
tournament_edit: Tournament Edit Page
battle_details: Battle Details Page
battle_edit: Battle Edit Page
battle_create: Battle Create Page
kata_submission: Kata Submissions Page
```

```
user_profile: User Profile Page
edit_profile: Edit User Profile Page
notifications: Notifications Page
friends_list: Friends List Page
home --> login: if not logged in
home --> tournament_list
home --> user_profile: if logged in
home --> notifications: if logged in
home --> tournament_create
tournament_list --> tournament_details
tournament_details --> battle_details
tournament_details --> tournament_edit: if logged user is\ncreator
tournament_details --> battle_create: if logged user is\ncreator or
coordinator
battle_details --> battle_edit: if logged user is creator
battle_details --> kata_submission
user_profile --> edit_profile
user_profile --> friends_list
```

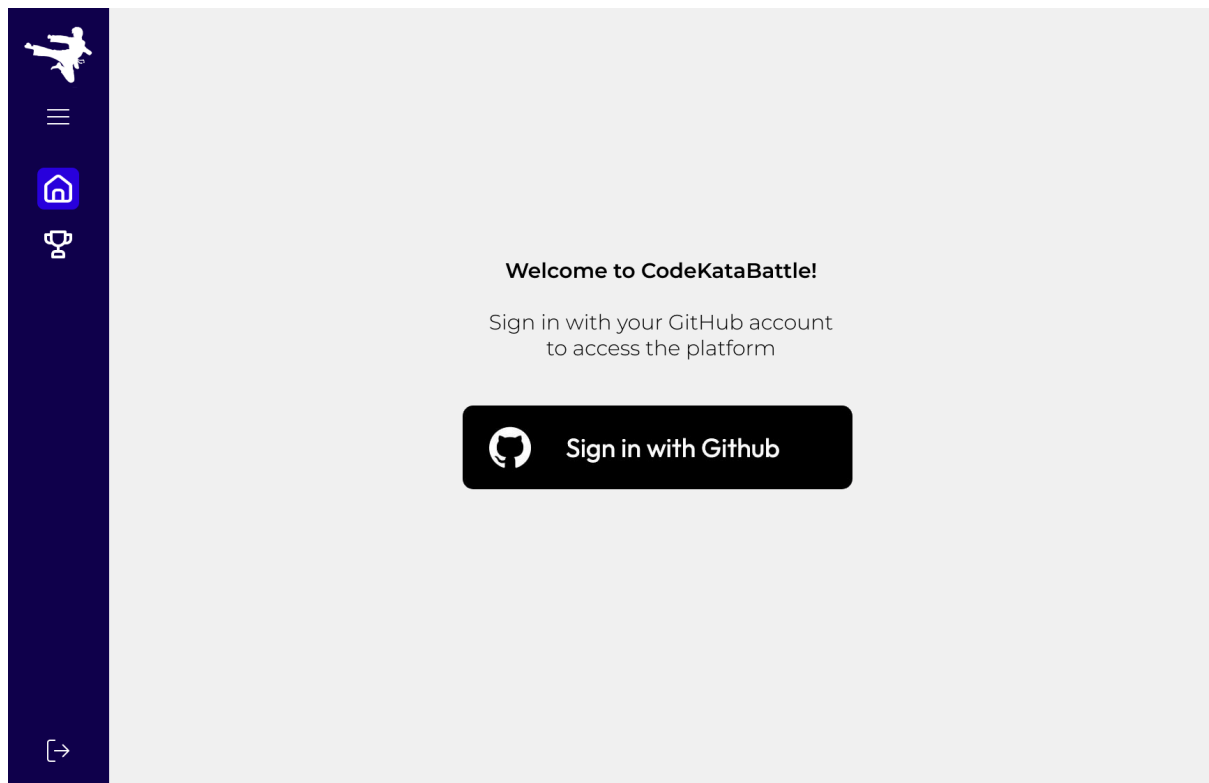
3.2 - Mockups

The mock-ups serve to indicate to the team developing the Client Application a general direction for the UX and style of the application (regarding inputs and typography). Not all pages are shown, as those will be developed by focus groups.

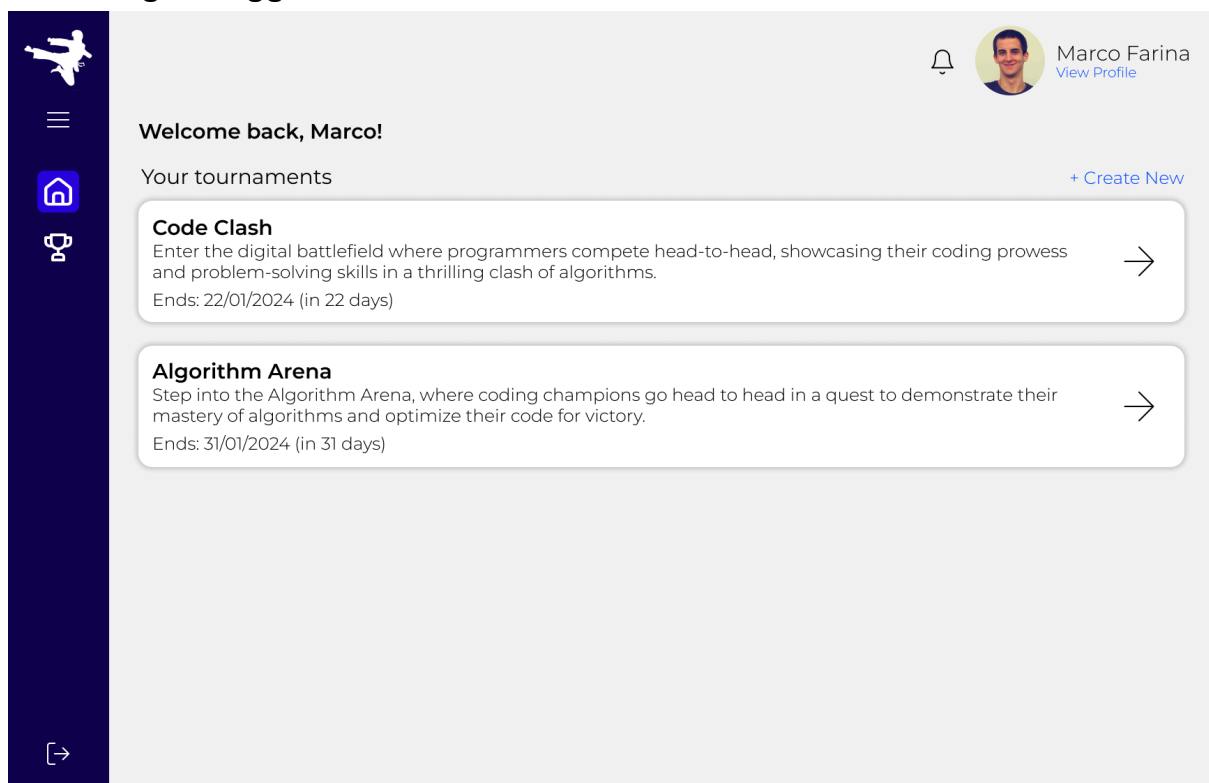
The structure of the application is composed of a simple sidebar and navbar. The sidebar allows users to navigate between different sections of the application, while the navbar allows users to visit the logged in user profile and view unread notifications.

The following images are rendered from [Figma](#):





Home Page - Not logged in





Home Page - Logged in



Upcoming Tournaments





Marco Farina
[View Profile](#)

Upcoming Tournaments

Advanced Python Trials
Show off your Python skills!
Enrollment Deadline: 02/01/2024 (in 2 days)
104 students subscribed (max 150)

→

The Ultimate Go Challenge
Improve your Go skills by competing with others
Enrollment Deadline: 05/01/2024 (in 5 days)
55 students subscribed (max 100)

→

Problem Solving Trials
Solve problems with various languages and compete to improve your programming skills
Enrollment Deadline: 10/01/2024 (in 10 days)
21 students subscribed (max 200)





→



10 Days with JavaScript
Compete with other students to improve your knowledge of JavaScript
Enrollment Deadline: 14/01/2024 (in 14 days)
10 students subscribed (max 500)

→

→

Create New Tournament





Marco Farina
[View Profile](#)

Create Tournament




Title
Code Clash


Description
Enter the digital battlefield where programmers compete head-to-head, showcasing their coding prowess and problem-solving skills in a thrilling clash of algorithms.

Enrollment Deadline
25/01/2024

Closing Deadline
26/02/2024






Privacy Level
Public



Coordinators:  [+ Invite coordinators](#)

Badges:  [+ Add badges](#)

Create

Tournament Details





Marco Farina
[View Profile](#)

Code Clash

Enter the digital battlefield where programmers compete head-to-head, showcasing their coding prowess and problem-solving skills in a thrilling clash of algorithms.

Battles **Leaderboard**

Battle #1 - Status: In Progress
Language: Python
Final Deadline: 02/01/2024 (in 2 days)
20 teams subscribed (min 2 per team, max 4 per team) →

Battle #2 - Status: Enrolling
Language: JavaScript
Enrollment Deadline: 10/01/2024 (in 10 days)
30 teams subscribed (min 1 per team, max 5 per team) →

Battle #3 - Status: Enrolling
Language: JavaScript
Enrollment Deadline: 20/01/2024 (in 20 days)
45 teams subscribed (min 1 per team, max 5 per team) →

4 - Requirements Traceability

	Client Application	User Service	Tournament Service	Battle Service	Badge Service	Compute Score Service	Executor Service	Analysis Service	Notification Service	Authentication Service	Notification DBMS	Main DBMS
SR1	X		X						X		X	
SR2	X			X					X		X	
SR3				X		X	X	X				X
SR4	X			X								X
SR5	X			X					X		X	
SR6	X								X		X	
SR7	X		X						X	X	X	X
SR8	X			X					X		X	
SR9	X		X									
SR10	X			X					X		X	X
SR11	X		X	X								X
ER1	X		X									X
ER2	X		X									X
ER3	X								X		X	
ER4	X		X									X
ER5	X				X							X
ER6	X		X	X								X
ER7	X			X								X
ER8	X			X								X
ER9	X		X	X								
ER10	X			X					X		X	
ER11	X			X								X
ER12	X			X					X		X	
ER13	X		X						X		X	X
ER14	X		X									X

	Client Application	User Service	Tournament Service	Battle Service	Badge Service	Compute Score Service	Executor Service	Analysis Service	Notification Service	Authentication Service	Notification DBMS	Main DBMS
SR1	X		X						X		X	
SR2	X			X					X		X	
SR3				X		X	X	X				X
SR4	X			X								X
SR5	X			X					X		X	
SR6	X								X		X	
SR7	X		X						X	X	X	X
SR8	X			X					X		X	
SR9	X		X									
SR10	X			X					X		X	X
SR11	X		X	X								X
ER1	X		X									X
ER2	X		X									X
ER3	X								X		X	
ER4	X		X									X
ER5	X				X							X
ER16	X		X									X
ER17	X		X									X
ER18	X		X						X		X	X
ER19												
ER20	X		X									X
ER21	X		X									X
ER22	X			X								X
OR1				X								
OR2	X		X									X
OR3	X		X						X		X	

	Client Application	User Service	Tournament Service	Battle Service	Badge Service	Compute Score Service	Executor Service	Analysis Service	Notification Service	Authentication Service	Notification DBMS	Main DBMS
SR1	X		X						X		X	
SR2	X			X					X		X	
SR3				X		X	X	X				X
SR4	X			X								X
SR5	X			X					X		X	
SR6	X								X		X	
SR7	X		X						X	X	X	X
SR8	X			X					X		X	
SR9	X		X									
SR10	X			X					X		X	X
SR11	X		X	X								X
ER1	X		X									X
ER2	X		X									X
ER3	X								X		X	
ER4	X		X									X
ER5	X				X							X
OR4	X	X							X		X	
OR5	X	X							X		X	X
OR6	X	X							X			X
OR7	X	X										X
OR8	X	X								X		X
OR9	X	X								X		X
OR10				X		X	X	X				X
OR11			X	X								X
OR12			X		X							X
OR13	X		X									X

	Client Application	User Service	Tournament Service	Battle Service	Badge Service	Compute Score Service	Executor Service	Analysis Service	Notification Service	Authentication Service	Notification DBMS	Main DBMS
SR1	X		X						X		X	
SR2	X			X					X		X	
SR3				X		X	X	X				X
SR4	X			X								X
SR5	X			X					X		X	
SR6	X								X		X	
SR7	X		X						X	X	X	X
SR8	X			X					X		X	
SR9	X		X									
SR10	X			X					X		X	X
SR11	X		X	X								X
ER1	X		X									X
ER2	X		X									X
ER3	X								X		X	
ER4	X		X									X
ER5	X				X							X
OR14	X		X									X
OR15		X	X						X		X	X
OR16			X	X					X		X	X
OR17		X		X								

5 - Implementation, Integration and Test Plan

5.1 - Implementation and Integration

In the implementation phase it is important that the Product Owner follows each step of the implementation in order to guarantee that requirements are being fulfilled correctly.

Regarding the implementation itself, first it is needed to deploy MainDBMS and NotificationDBMS (as they are external applications), after that the implementation can start.

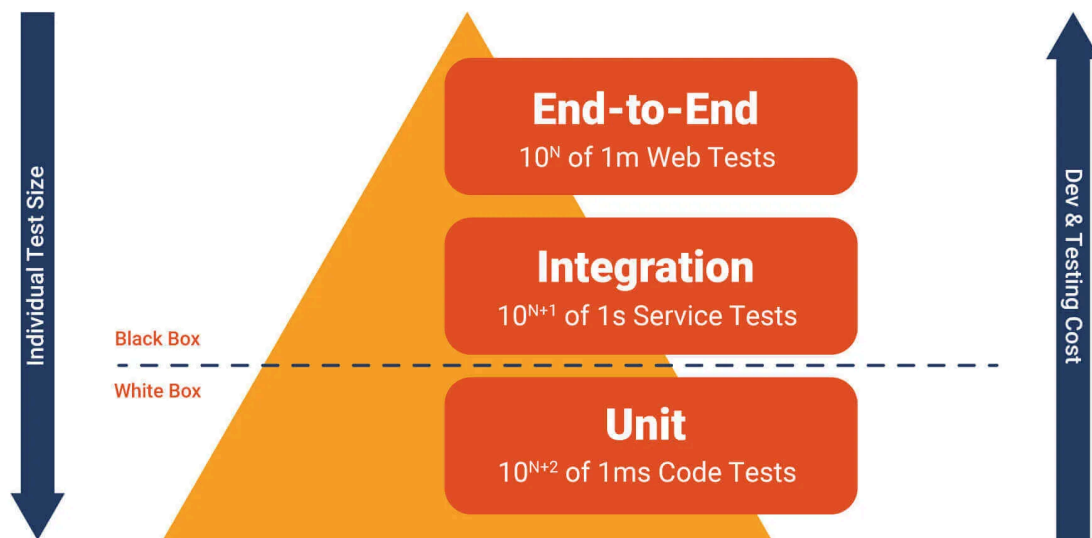
The initial approach for the implementation is Bottom-up. Then, after the decomposition of the model in small units, it is possible to proceed with the identification of the independence, so the units independence can be implemented in parallel.

Order of implementation of services:

- 1) UserService, AuthenticationService, NotificationService;
- 2) TournamentService, ExecutorService, AnalysisService, BadgeService, ComputeScoreService;
- 3) ClientApplication.

5.2 - Test Plan

The approach that will be followed for writing tests is the famous pyramidal approach, where there are many unit tests followed by fewer integration tests and, lastly, only a very small amount of E2E tests.



5.2.1 - Functional Testing (Unit + Integration)

First, the CKB system will be tested by using the properly written unit tests for each module. These tests aim to show the exactness and soundness of the codebase at the level of singular functions, by mocking every dependency. These tests are very short and their execution should be as fast as possible, in order to let the developer iterate quickly without waiting for unit tests to execute.

The next step will be integration testing. These tests span across multiple components, are longer and slower compared to unit tests, and aim to show that components are communicating correctly (following the correct format of inputs and outputs) and no unexpected errors are being thrown.

5.2.2 - End to End Testing (E2E)

The last type of tests to be implemented are End to End tests (E2E), which aim to test end user interactions with the entire system through the UI (Client Application component). These tests require the emulation of an entire browser environment and simulation of user interactions. As such, they are very slow and consume a lot of resources, so there must be a small number of them implemented. They show the correctness of entire application flows at once.

5.2.3 - MVP testing

Lastly, the MVP of the application will be tested by a small, selected group of schools interested in adopting the solution for their students. These tests will be performed manually and will aim to identify possible sources of confusion in the UI/UX of the program, and solve them before releasing the final version of the product.

6 - Effort Spent

Group Member	Effort Spent (hours)
Nicolò Giallongo	Introduction: 4h Architectural Design: 12h User Interface Design: 2h Requirements Traceability: 6h Implementation, Integration and Test Plan: 2h
Giovanni Orciuolo	Introduction: 1h Architectural Design: 8h User Interface Design: 14h Requirements Traceability: 3h Implementation, Integration and Test Plan: 6h
Giuseppe Vitello	Introduction: 1h Architectural Design: 13h User Interface Design: 2h Requirements Traceability: 2h Implementation, Integration and Test Plan: 6h

7 - References

- Version Control System: GitHub (<https://github.com>)
- State diagrams made with: Mermaid (<https://mermaid.js.org>)
- Sequence diagrams made with: Mermaid (<https://mermaid.js.org>)
- Mock-ups made with: Figma (<https://www.figma.com/>)
- Document written with: Google Documents (<https://docs.google.com>)