POLITECNICO DI MILANO

Computer Science and Engineering

Design Document

CodeKataBattle

Authors:

Manuela Marenghi
Luca Cattani
Tommaso Fellegara

Professor:

Matteo Giovanni Rossi

Table of Contents

1. Introduction

1.1 Scope

1.1.1 Product domain

The platform allows students to take part in coding tournaments, where they will have to solve coding problems in the form of battles.
A code kata consists of a project containing:

- a textual problem description
- a set of test cases the implementation must pass
- any necessary build automation tool

Each tournament is created by an educator, who can choose to allow other educators to create battles for the tournament. To create a new battle within a tournament on the platform, an educator must have been given permission to create battles for that tournament and has to provide the following data:

- a code kata
- the minimum and maximum number of students per group
- a registration deadline
- a final submission deadline
- configurations for scoring

Educators can also create gamification badges for their tournaments, these are elements in the form of individual rewards with a title and a rule about how to obtain them. Each badge can be assigned to one or more students, depending on the rules. When a student obtains a badge, it will show up on their profile, and everyone else (educators and other students) will be able to see it.

All students subscribed to the CKB platform are notified whenever a new tournament is created, and they can subscribe to the tournament by a given deadline (chosen by the tournament creator). If they subscribe, they are notified of all upcoming battles created within that tournament.
After the creation of a battle, students use the platform to form teams for that battle. In particular, each student can join a battle on their own or by inviting other students to their team (respecting the minimum and maximum number of students per group set for that battle by the creator).
When a battle's registration deadline expires, CKB creates a GitHub repository containing the code kata and sends the link to all students who are members of a valid subscribed team. In particular, students are asked to fork the GitHub repository of the code kata and set up an automated workflow through GitHub actions that informs the CKB platform (through proper API calls) as soon as a commit is pushed into the main branch of their repository.

Each commit pushed to the main branch of a group's repository must trigger the CKB platform (through GitHub actions) to pull the repository's source, analyze it by running tests on the corresponding executables, and calculate and update the battle score for that team. The score is a number between 0 and 100 and is calculated considering the following:

- number of test cases passed
- time passed between the start of the contest and the time of the submission
- quality of the code (in the matter of security, maintainability and reliability)
- a personal score assigned by the educator (optional)

At the end of each battle, the platform updates the personal tournament score of each student, that is, the sum of all battle scores received in that tournament. Thus, for each tournament, there is a rank that measures how a student's performance compares to other students in the context of that tournament. All users can see the list of ongoing tournaments as well as the corresponding tournament rank.
When an educator closes a tournament, as soon as the final tournament rank becomes available, the CKB platform notifies all students involved in the tournament.

Each user (student or educator) may also browse the list of present and past tournaments, look at tournament and battle rankings, and check out any student or educator profile.

1.1.2 Main architectural choices

The system is to be implemented using a microservices oriented architecture, this allows for independent scaling of individual components based on their specific requirement and eases the process of implementation and testing, moreover this enables efficient resource utilization and ensures that only the necessary components are scaled, optimizing performance and cost-effectiveness.
Additionally, microservices foster flexibility and maintainability by allowing each service to be developed, deployed, and updated independently by different teams, this reduces the risk of system failures, as issues within one service are less likely to impact the entire system, availability is also increased by this choice as if one component fails or has to be updated, the system is not necessarily affected as a whole but only a limited set of features is temporarily unavailable.
Furthermore, microservices promote technology diversity, enabling teams to choose the most suitable tools and frameworks for each service because components can use one another in a black-box fashion, without knowing how a component is implemented, therefore providing a higher layer of abstraction.

1.2 Definitions, acronyms, abbreviations

1.2.1 Definitions

- **Code kata** - the set of: textual description, test cases and build automation scripts that form a coding problem users on the platform have to solve.

- **Code battle** - the grouping of code kata and battle settings, described by an educator, that constitute a coding challenge on the platform. Note that code battles are also simply referred to as “battles” in this document.
- **Tournament** - a collection of code battles created by one or more educators.
- **Users** - everyone using the platform, that is, students, educators and everyone who is browsing the platform and is not logged in yet.

1.2.2 Acronyms

- **CKB** - CodeKataBattle
- **API** - Application Programming Interface
- **UML** - Unified Modeling Language
- **RASD** - Requirement Analysis and Specifications Document
- **DBMS** - DataBase Management System

1.2.3. Abbreviations

- **[Ri]** - i-th requirement

1.3 Reference documents

1.4 Overview

The bulk of the document relies in the Architectural design chapter which is structured as follows:

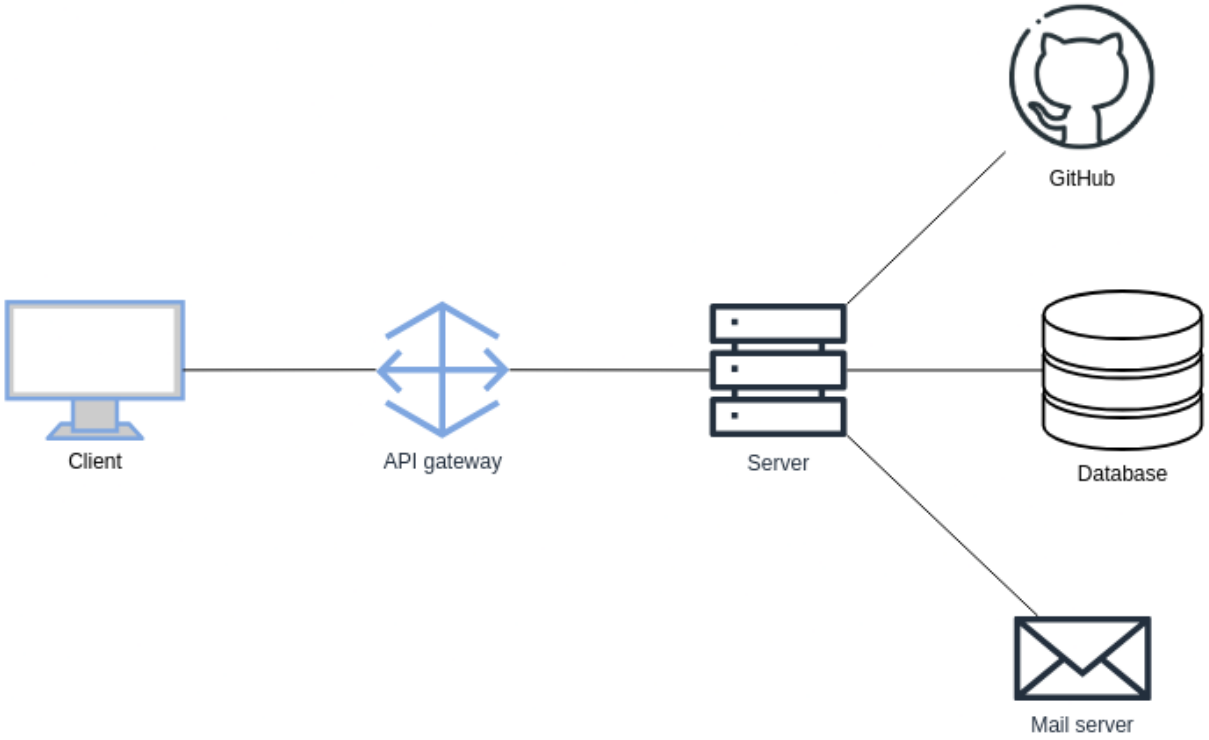
- **Overview**: general description of the main aspects of the system, not going into detail about the inner workings of the system, but defining which are the main components and how they interact with each other
- **Component view**: a more in depth view on which are the elements that make up the components described in the overview section, in particular a better description of the server components can be found here
- **Deployment view**: description of the system infrastructure displaying the role of non-logical elements and the displacement of resources
- **Component interfaces**: this section is dedicated to the description of how different components interact with each other within the system and which interfaces the system exploits and provides to allow the interaction with external components
- **Runtime view**: description through UML diagrams of how components interact with each other with respect to use cases

Another key part of the document is the requirements traceability chapter, here the relations between requirements and design elements is highlighted, providing a view of how the structure of the system is designed to satisfy requirements at best.
Finally, the Implementation, Integration and test Plan chapter provides a description on how the system should be implemented, focusing on the order of implementation of the single components, together with a plan for testing and integrating the services with each other.

2. Architectural Design

2.1 Overview: high-level components and interactions

The system will serve requests from clients, which may contact the server through the CKB platform or by exploiting the API provided by the system (which could also happen through the students triggering a GitHub actions workflow).

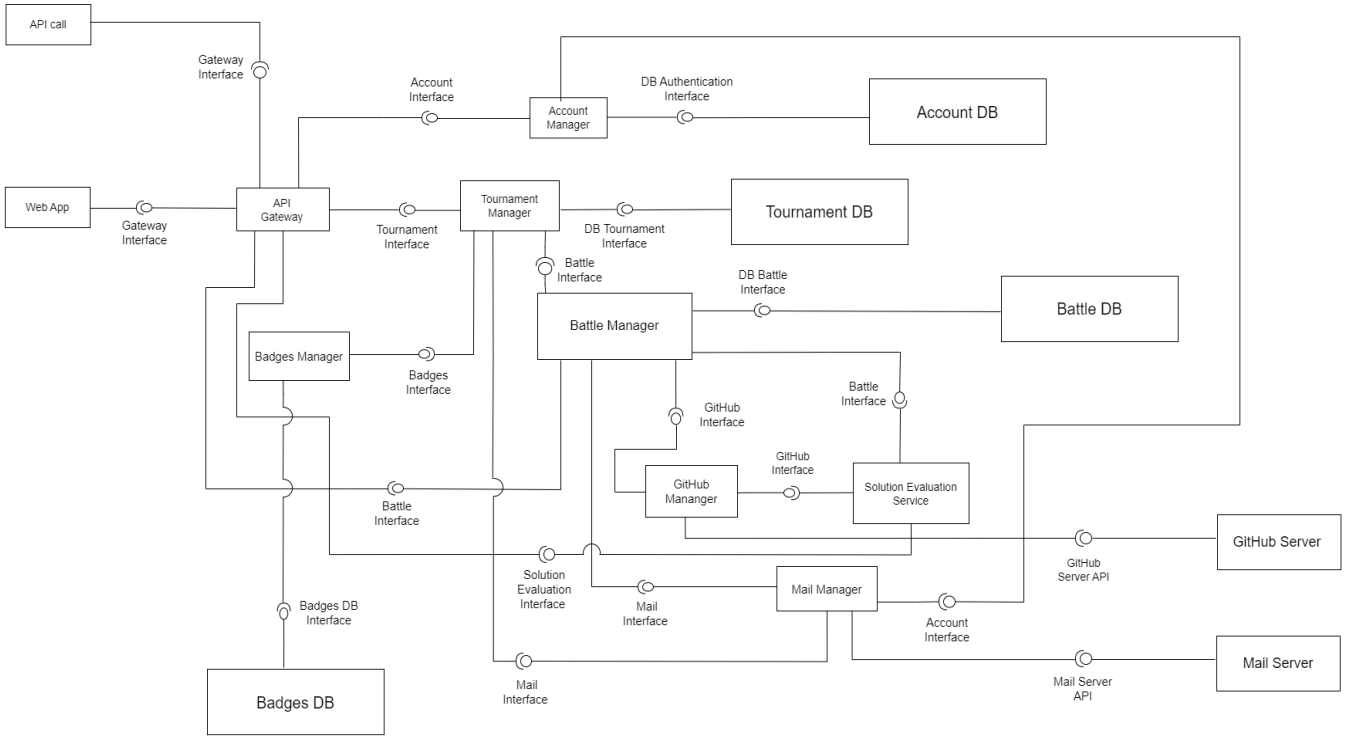


Whenever a client interacts with the system, it's request is processed by an API gateway, which directs the request to the designed service, the server may then interact with other services depending on the interaction:

- The request may cause the need to query the database
- The server may need to exploit the API provided by GitHub for the following cases:
    - Pull the sources of a repository to evaluate the student's solution to a certain code kata
    - Create a GitHub repository when an educator creates a battle
- A mail server may be used to interact with users whenever they have to be notified, that is:
    - Students get notified for relevant events regarding battle and tournaments
    - Students receive a notification every time someone invites them to a team
    - Educators are notified whenever other educators invite them to collaborate on the creation of battles for a tournament

2.2 Component view

In this section all components of the system are illustrated, explaining their roles and positions in relation to one another. The system is implemented following a microservices architecture instead of using a single server to handle application requests.



The components in the diagram are explained in detail as follows:

- **Web App** : represents the website used by each type of user of the system. By using the **Gateway Interface** it sends requests to the **API Gateway** that has the responsibility of redirecting each request to the right microservice that can handle it. Each user would have to use an arbitrary web browser to access it.

- **API call** : represents a call starting from GitHub by using CKB API when a push on repositories occurs. It uses the **API Gateway** in order to reach the **Solution Evaluation Service** that is the microservice that performs the static analysis of the project.

- **API Gateway** : this component is the dispatcher above all microservices. Each request is at first handled by this component that redirects it to the appropriate microservice. It interacts with all the interfaces of each microservice.

- **Account Manager** : this component is the microservice that handles information about the account of each user. Whenever a user wants to log in it asks this service to custom the Web App based on its type of account. Requests arrive from the **API Gateway** by using the **Account Interface**. It also uses the **DB Authentication Interface** to interact with its database where it stores information about accounts.

- **Tournament Manager** : this component is about the microservice that handles tournaments. As for the others it receives requests from the **API Gateway** and it stores its information in its database by using the **DB Tournament Interface**. In particular, those information are about tournaments, their battle list and participants. It also interacts with **Badges Interface** in order to send information about badges and their rules, **Battle interface** and **Mail Interface** to send notifications when a tournament is created.

- **Badges Manager** : this component is about the microservice that handles badge information. It receives information from the **Tournament Manager** and stores it in its database using the **Badges DB Interface**.

- **Battle Manager** : this component is about the microservice that handles information about battles that stores in its database by using **DB Battle Interface**. It receives requests from the **API Gateway** and from the **Tournament Manager** and information about points from the **Solution Evaluation Service** whenever a static analysis occurs. It also sends notifications about teams by using the **Mail Interface** and gets team repositories from the **GitHub Manager**.

- **GitHub Manager** : this component provides the microservice about the creation of GitHub repositories when a battle is added to a tournament. This allows the system to retrieve the link at the repository, by using the **GitHub Interface** and to pass it to the **Solution Evaluation Service** through the **GitHub Interface**.

- **Solution Evaluation Service** : this component is about the microservice that performs the static analysis on students solutions using the **Solution Evaluation Interface**. Then it sends points updated to the **Battle Manager** using the **Battle Interface**.

- **Mail Manager** : this component handles the service about notifications that happen by sending mail. It receives requests from **Tournament Manager** and **Battle Manager** , both using the **Mail Interface**. Then it interacts with the **Mail Server** of each specific receiver and with the **Account Manager** to check if the insert email has an account associated .

- **Badges DB** : this component represents the badges database that stores badges of each battle, their descriptions and rules and which ones are assigned and their owner.

- **Account DB** : this component is about the database of the **Account Manager** and it stores information about users, their accounts and their personal information given to the system.

- **Battle DB** : this component represents the **Battle Manager** database. It stores different tables about battles, their description, participants and teams.

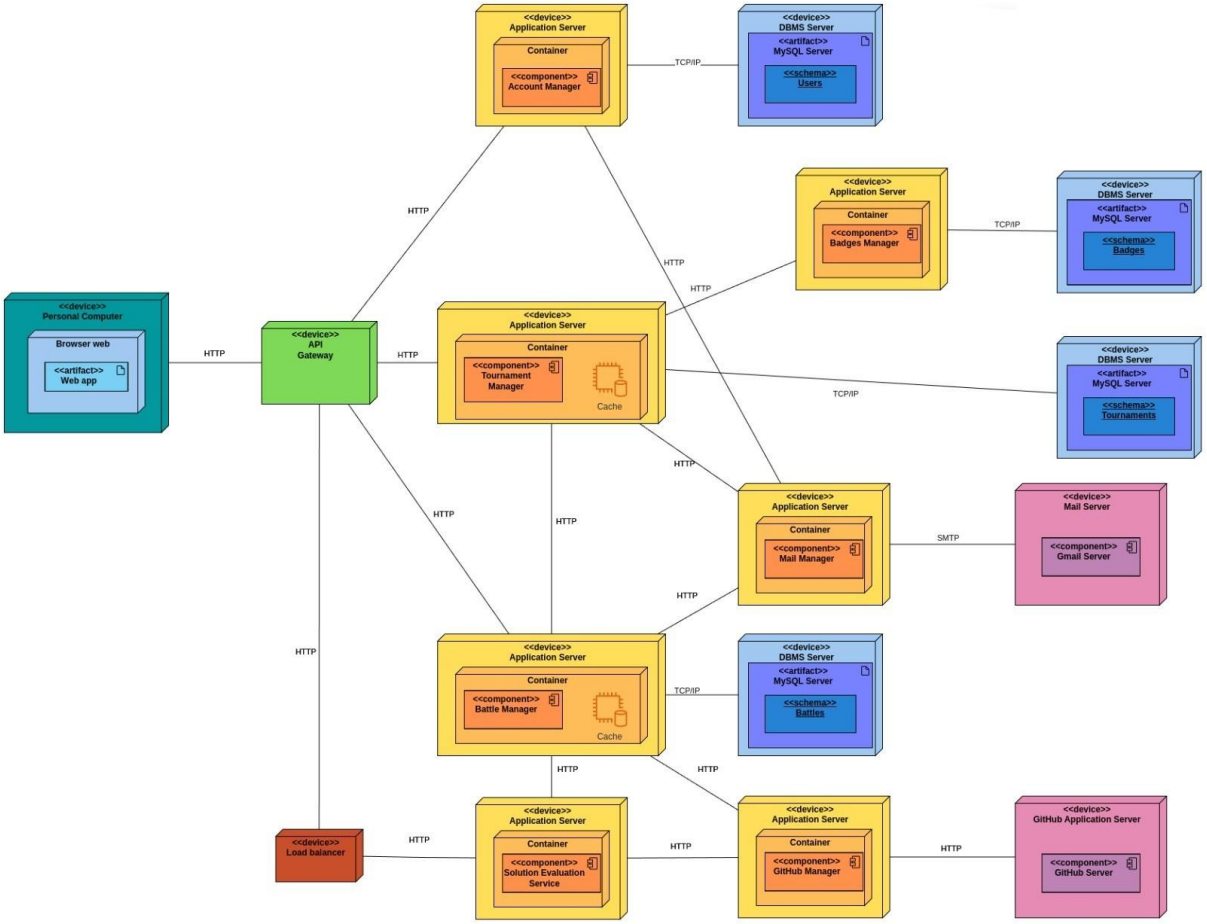
- **Tournament DB** : this component represents the **Tournament Manager** database and keeps information about tournaments, their battles and creators, permissions to educators, rankings and all descriptions related to tournaments.

- **GitHub Server** : this component represents the server of GitHub that receives requests from the **GitHub Manager** in order to retrieve links of repositories about each battle.

- **Mail Server** : this component provides the interface to the **Mail Manager** to deal with the email receiving services needed. Whenever a notification occurs, that means an mail is sent, it happens through the **Mail Interface** that contacts the specific **Mail Server** of the receiver.

2.3 Deployment view

In this chapter the deployment view for CodeKataBattle is described. This view describes the execution environment of the system, together with the physical distribution of the hardware components that executes the software.



Since the architecture chosen for this application is a microservices architecture, all the microservices work independently in different devices with their own MySQL database and exchange information through API calls.
In particular to avoid congestion of data and to speed up the application, a load balancer has been put before the Solution Evaluation Service since it is the most time consuming microservice.
Moreover, since there can be a lot of equal requests to Tournament Manager and Battle Manager, a cache has been provided in order to increase the performance and the efficiency of these microservices.

2.4 Component interfaces

This section is a summary of all the methods that each component provides to the rest of the system, including names, return types and required arguments.

2.4.1 Account Manager

Account Interface

void createNewUser(String fullName, String email, String password, AccountType type)
void logUser(String email, String password)
User getUser(Integer id)
void changeInformations(String userID, String fullName, String email, String password)
String getMail(integer userID)

2.4.2 Badges Manager

Badges Interface

void createBadges(List<Badge> badges)
List<Badge> getBadges(owner_id)
void createNewVariable(Variable variable)

2.4.3 Tournament Manager

Tournament Interface

void createTournament(Tournament tournament)
void closeTournament(Integer id)
Tournament getTournament(Integer id)
void grantPermission(Integer id_tournament, Integer id_new_educator)
List<Pair<User, Integer>> getRanking(Integer id_tournament)
void register(Integer id, String fullName)

2.4.4 Mail Manager

Mail Interface

void sendMail(Integer userID, String content)
void sendMassMail(List<Integer> userID, String content)
void mailAllStudents(String content)

2.4.5 Battle Manager

Battle Interface

Battle getBattle(Integer id)
void createBattle(Battle battle, Integer id_tournament)
void joinBattle(Integer id_battle, Integer id_student)

void leaveBattle(Integer id_battle, Integer id_student)
Team getTeam(Integer id)
List<Team> getTeams(Long id_battle)
List<Pair<User, Integer>> getAllUsersPoints(Tournament id)
void registerTeam(Integer id_team, Integer id_user)
void inviteStudent(Integer id_team,Integer id_user)
void createTeam(Integer id_user)
void assignScore(Integer id_team, Integer points)
void assignPersonalScore(Integer id_team, score)

2.4.6 Solution Evaluation Service

Solution Evaluation Interface

void evaluateSolution(Integer battleID, Integer id_team)

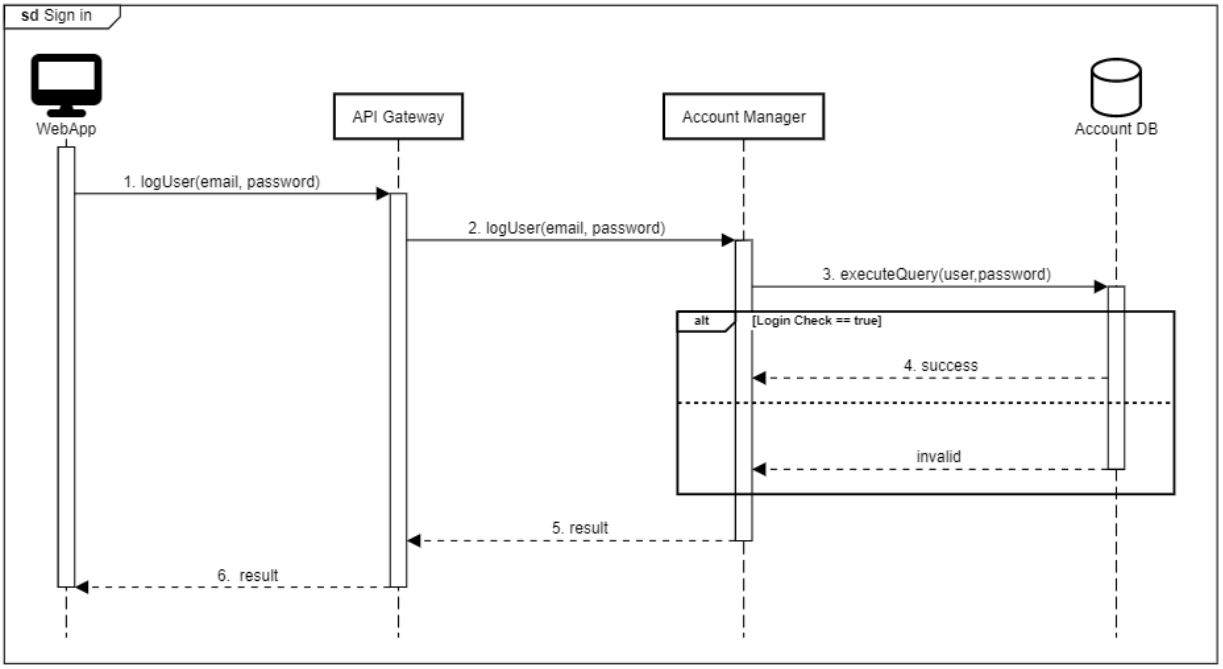
2.4.7 GitHub Manager

GitHub Interface

File fetchSolution(Integer battleID, Integer teamID)
void createBattleRepo(CodeKata codeKata, Integer battleID)

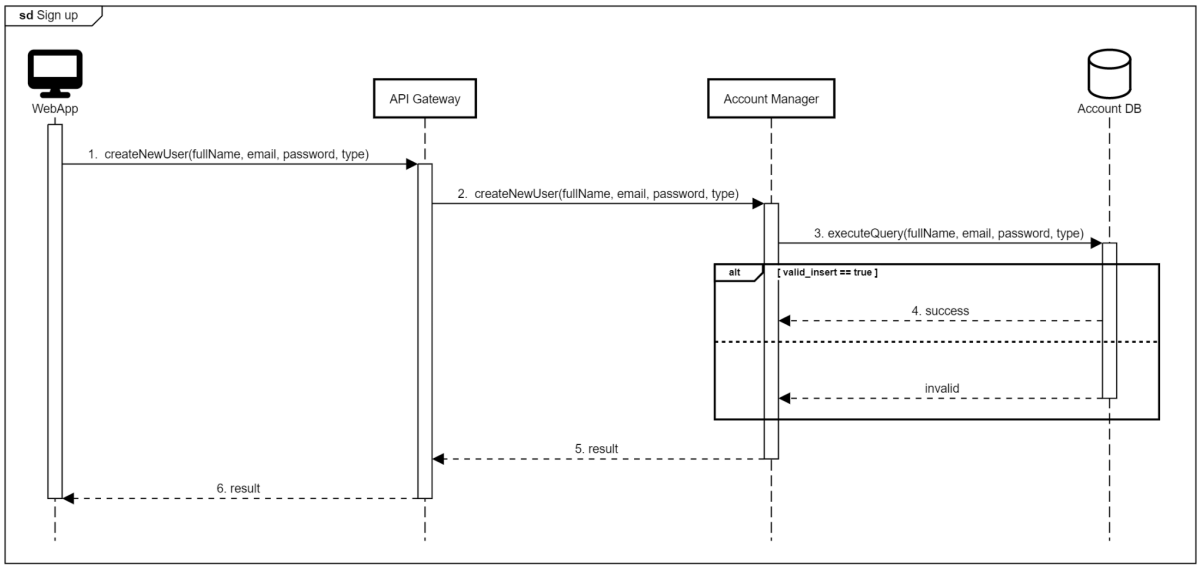
2.5 Runtime view

[Sign in]



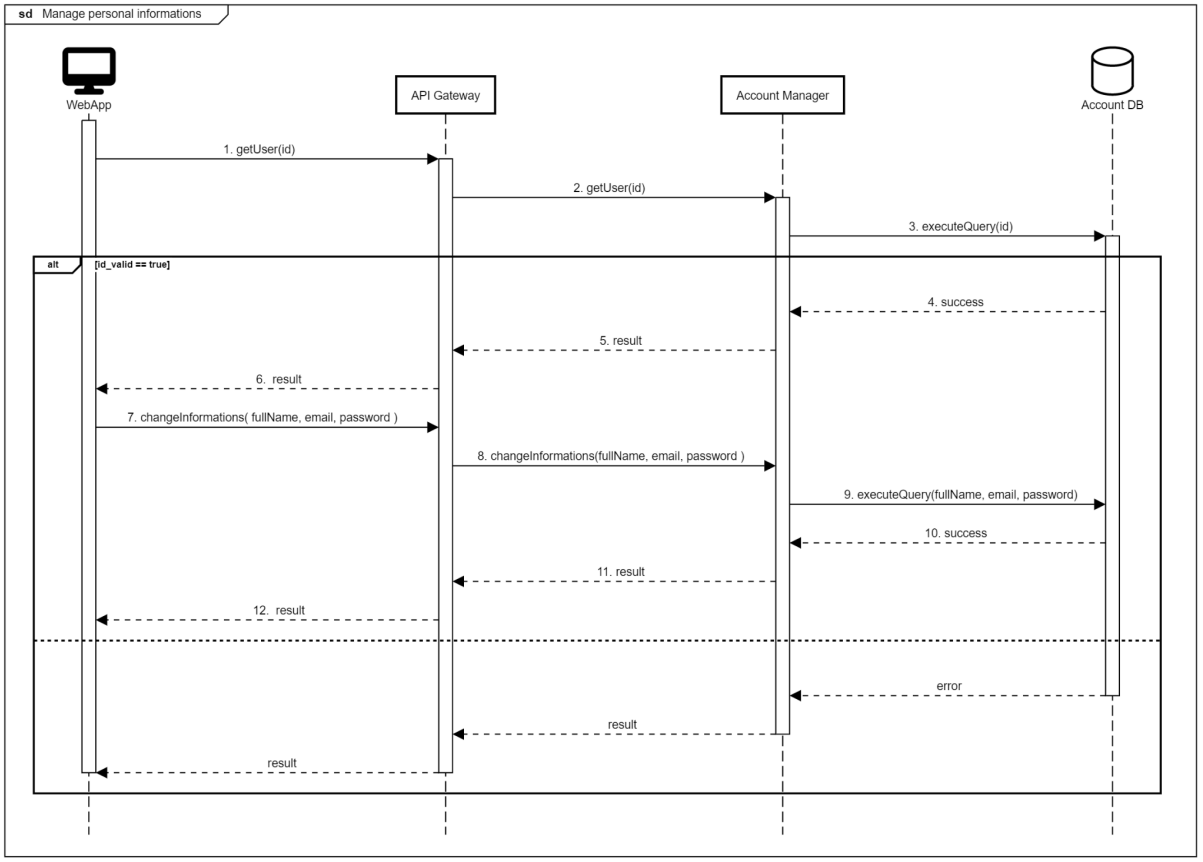
This sequence diagram represents the interaction that happens when a user wants to sign in to the CKB application assuming that the student is not.

[Sign up]



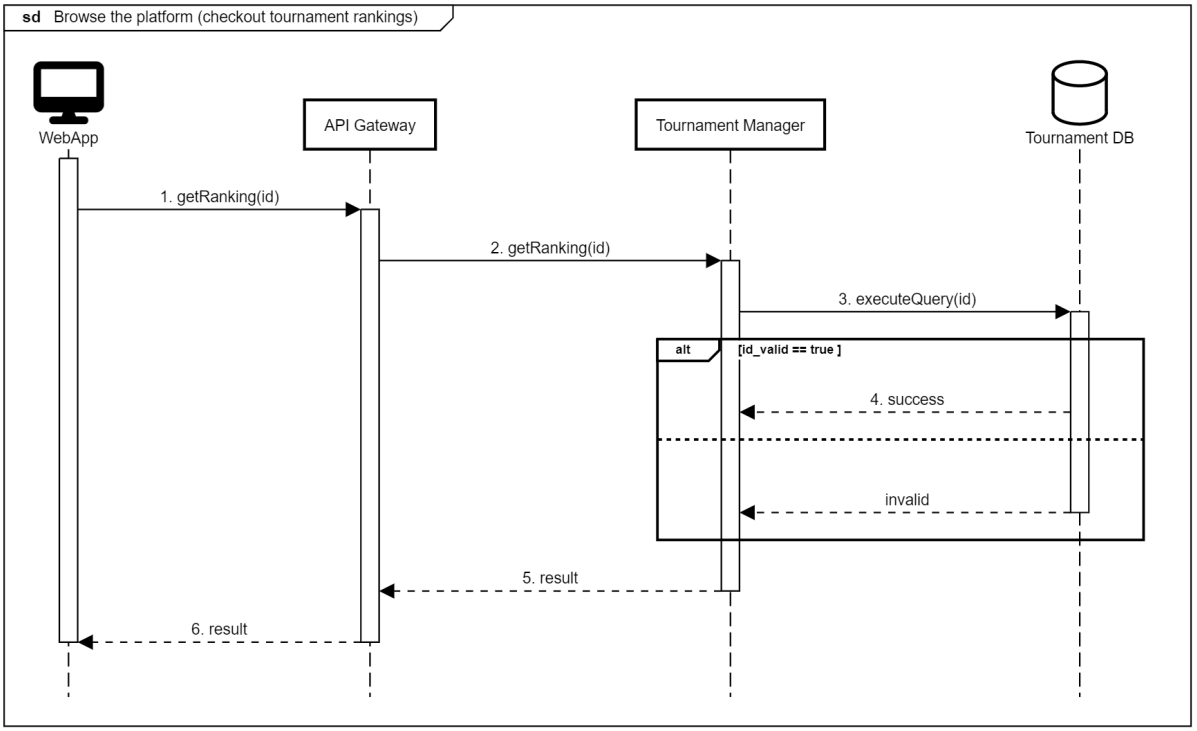
This sequence diagram represents the interaction that happens when a user wants to sign up to the CKB application assuming that the student is not.

[Manage personal informations]



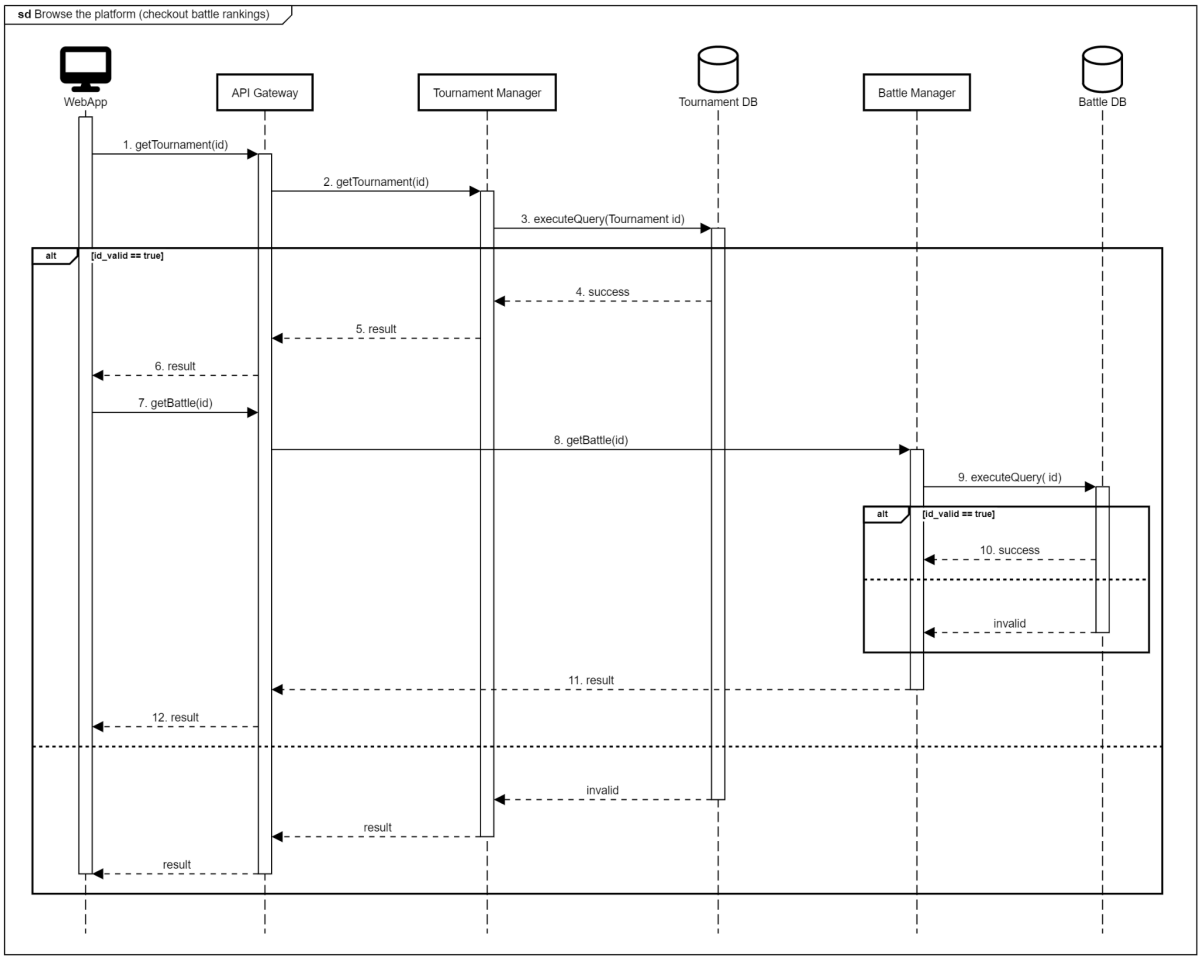
This sequence diagram represents the interaction that happens when a user wants to manage its personal information in the CKB application.

[Browse the platform (checkout tournament rankings)]



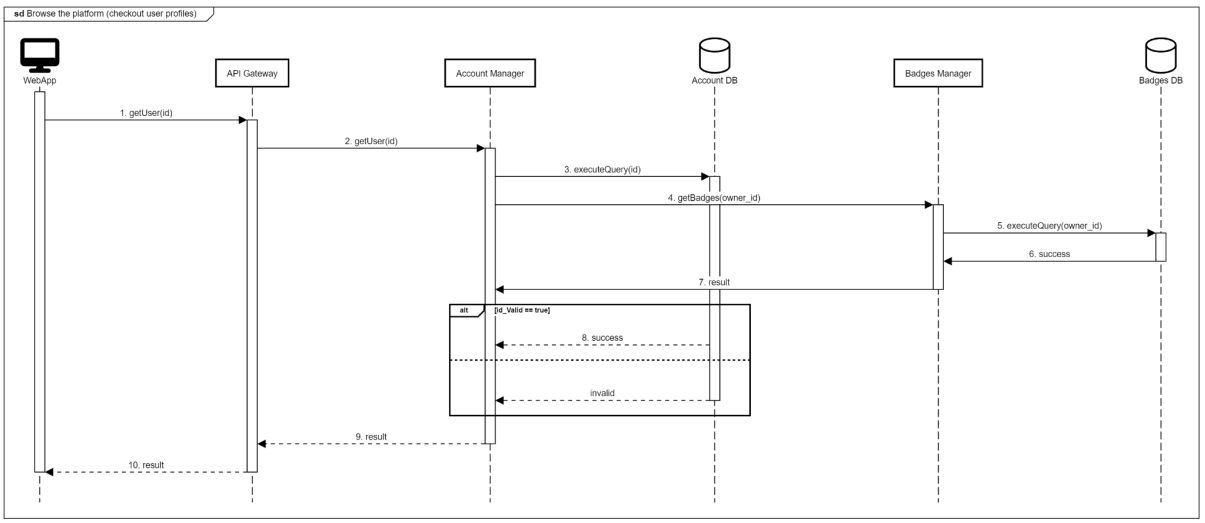
This sequence diagram represents the interaction that happens when a user wants to checkout tournament rankings that equals reaching the specific tournament page.

[Browse the platform (checkout battle rankings)]



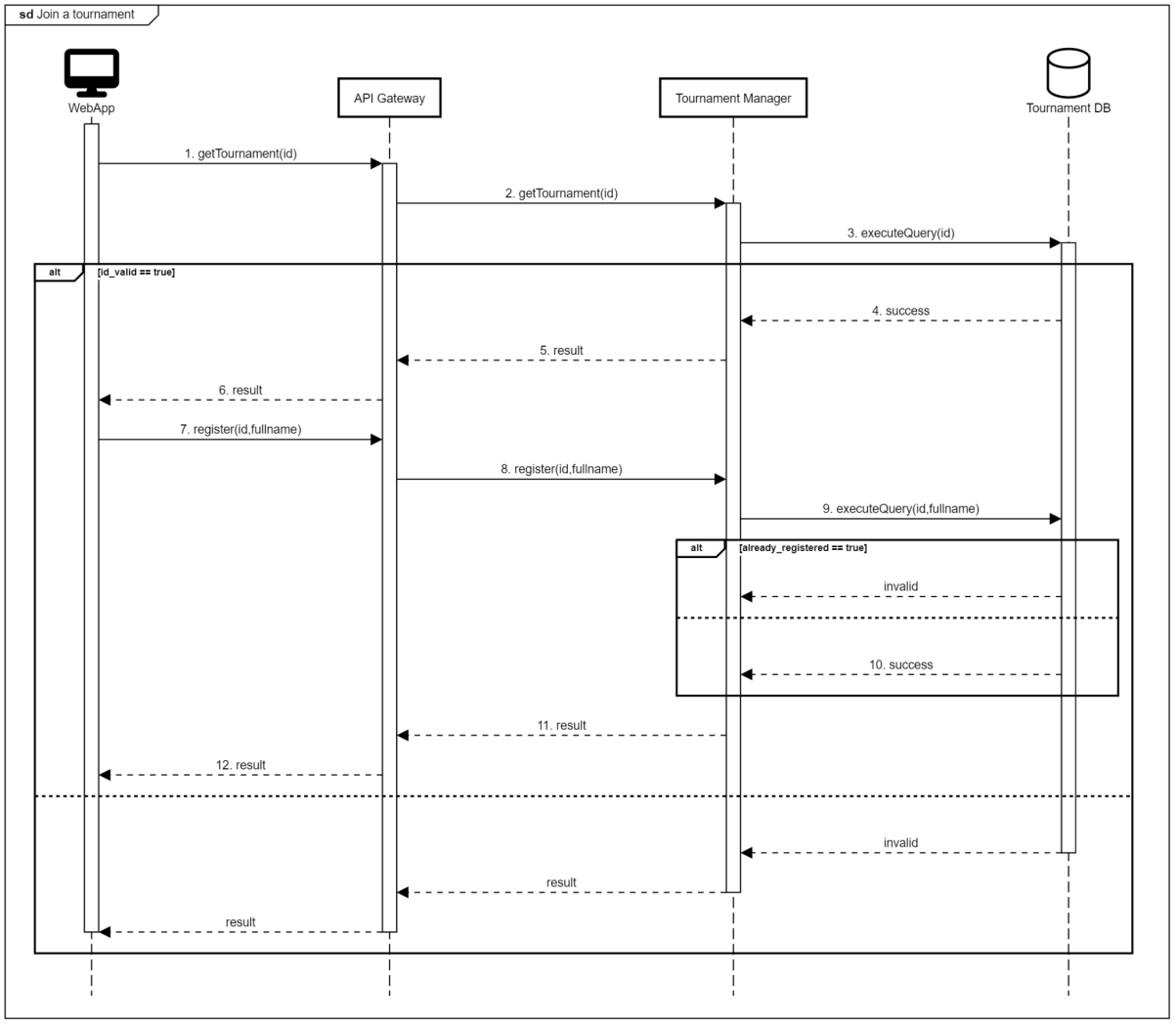
This sequence diagram represents the interaction that happens when a user wants to checkout battle rankings that equals reaching the specific battle page.

[Browse the platform (checkout user profiles)]



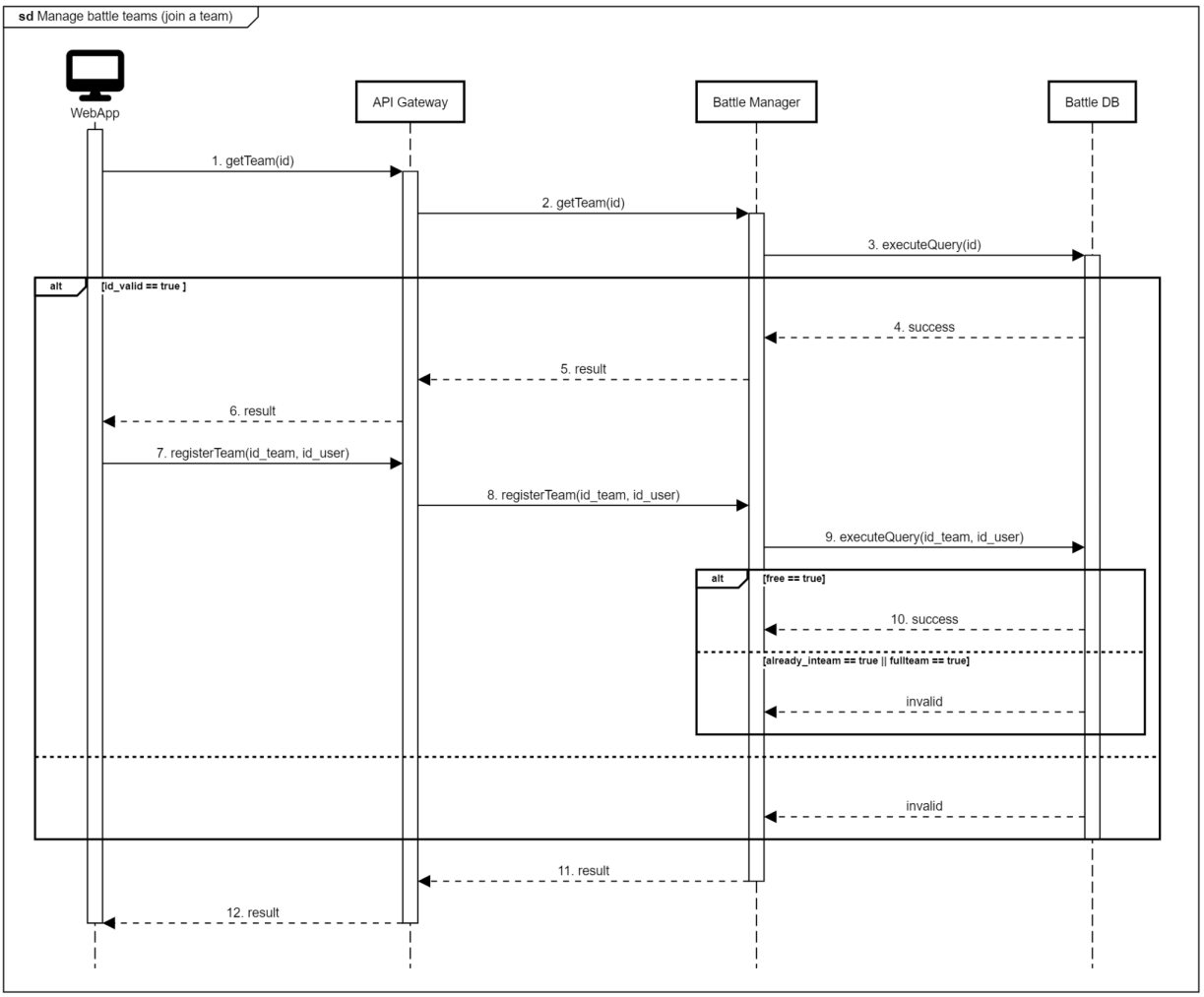
This sequence diagram represents the interaction that happens when a user wants to search for another user.

[Join a tournament]



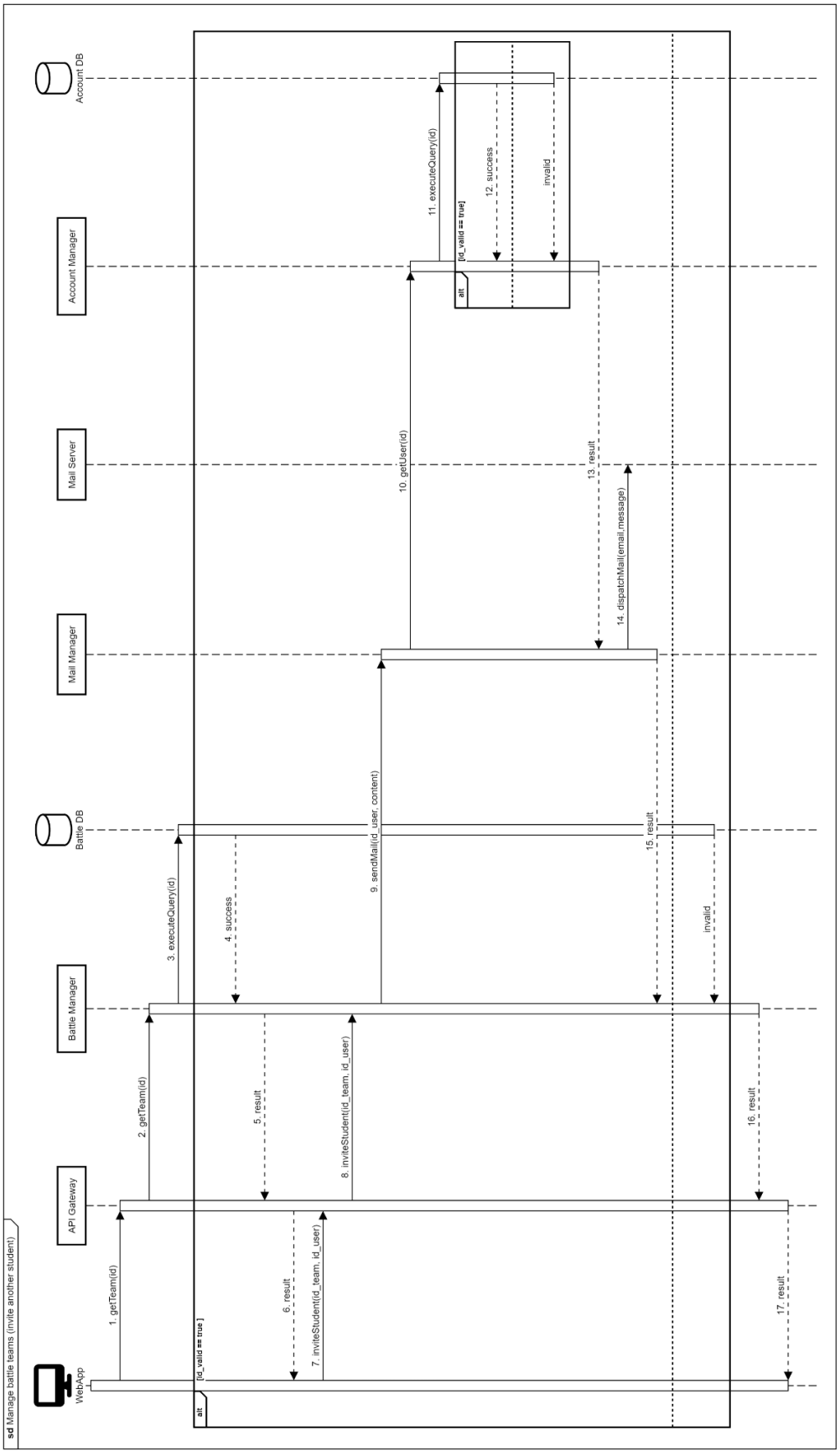
This sequence diagram represents the interaction that happens when a student wants to register to a specific tournament assuming that it's already logged.

[Manage battle teams (join a team)]



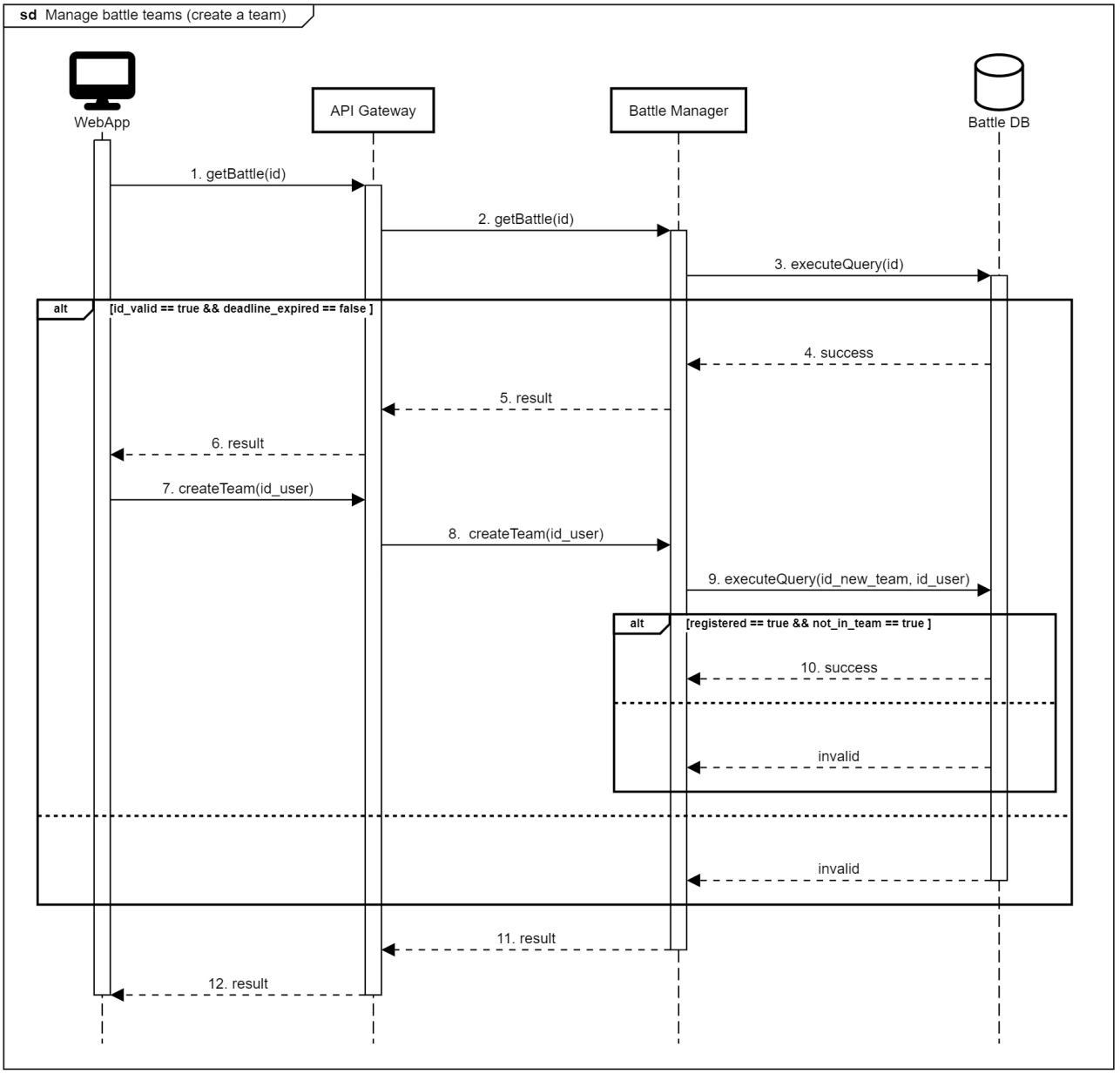
This sequence diagram represents the interaction that happens when a student wants to join a team.

[Manage battle teams (invite another student)]



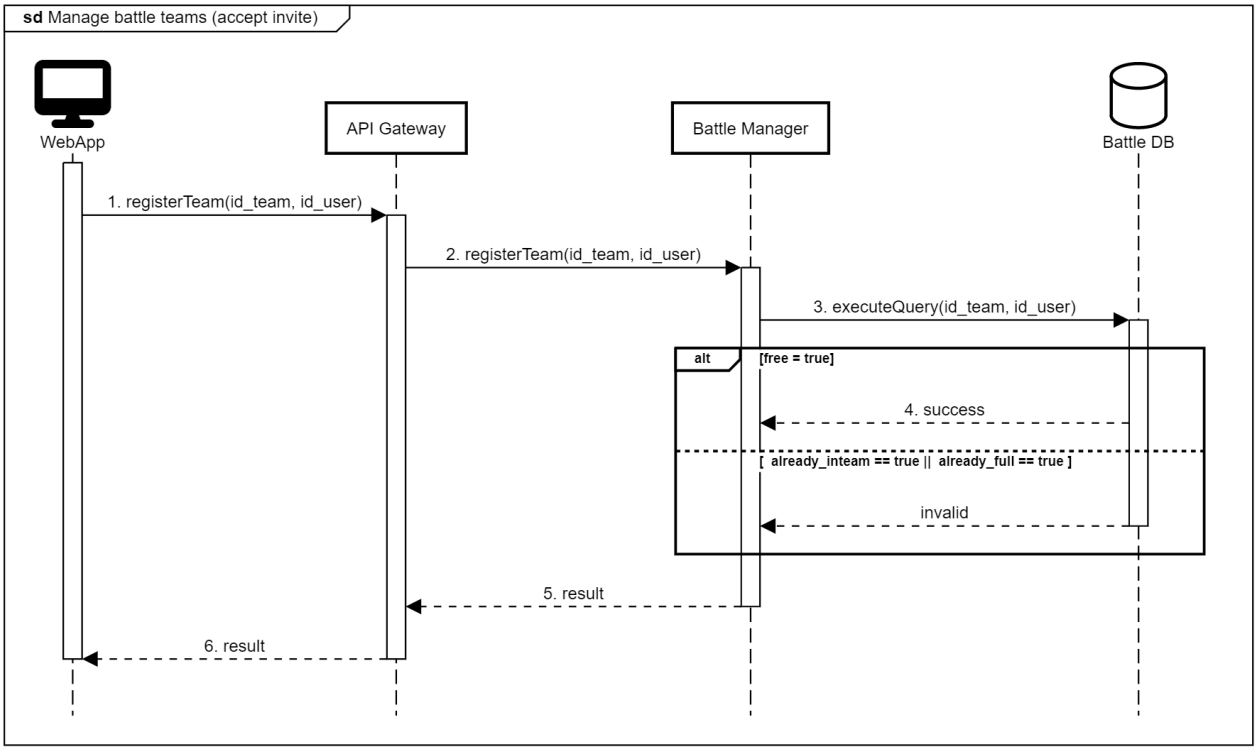
This sequence diagram represents the interaction that happens when a student wants to invite another student to its team.

[Manage battle teams (create a team)]



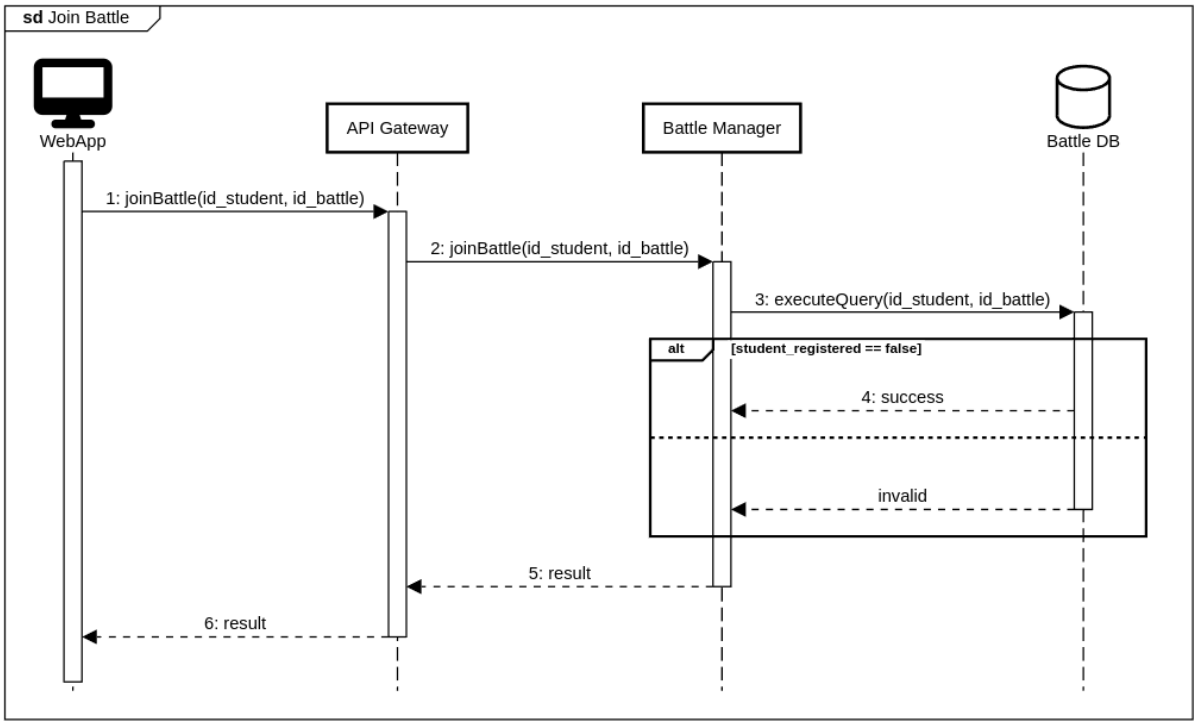
This sequence diagram represents the interaction that happens when a student wants to create a team.

[Manage battle teams (accept invite)]



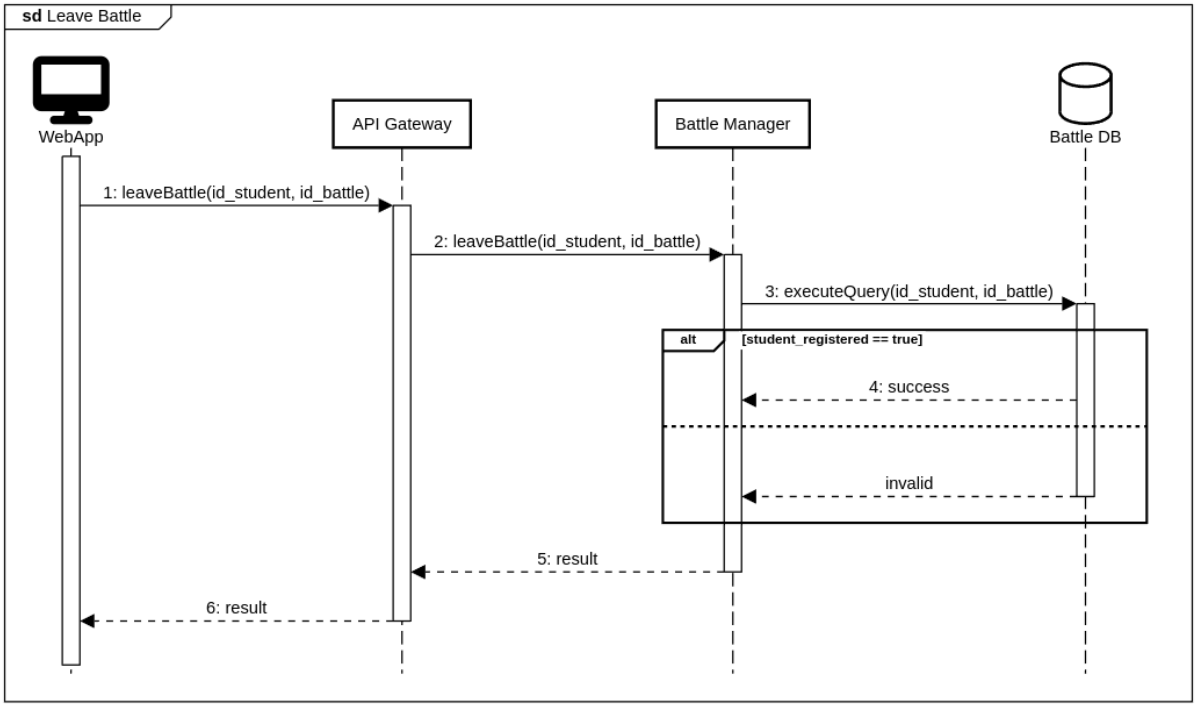
This sequence diagram represents the interaction that happens when a student wants to accept an invite received from another student.

[Join Battle]



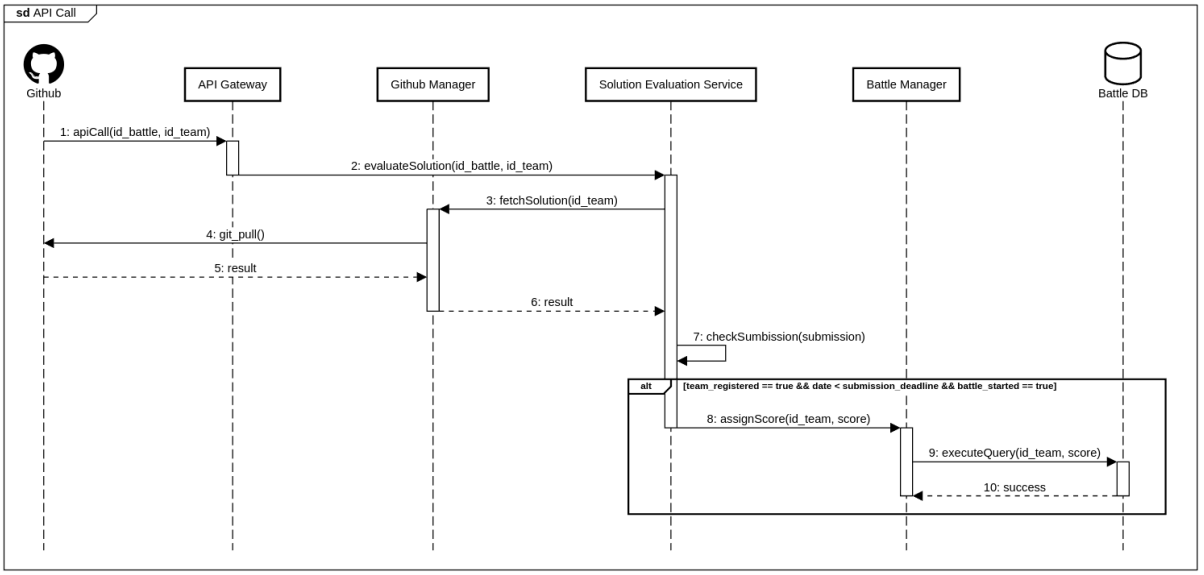
This sequence diagram represents the interaction that happens when a student wants to join a battle in the CKB application.

[Leave Battle]



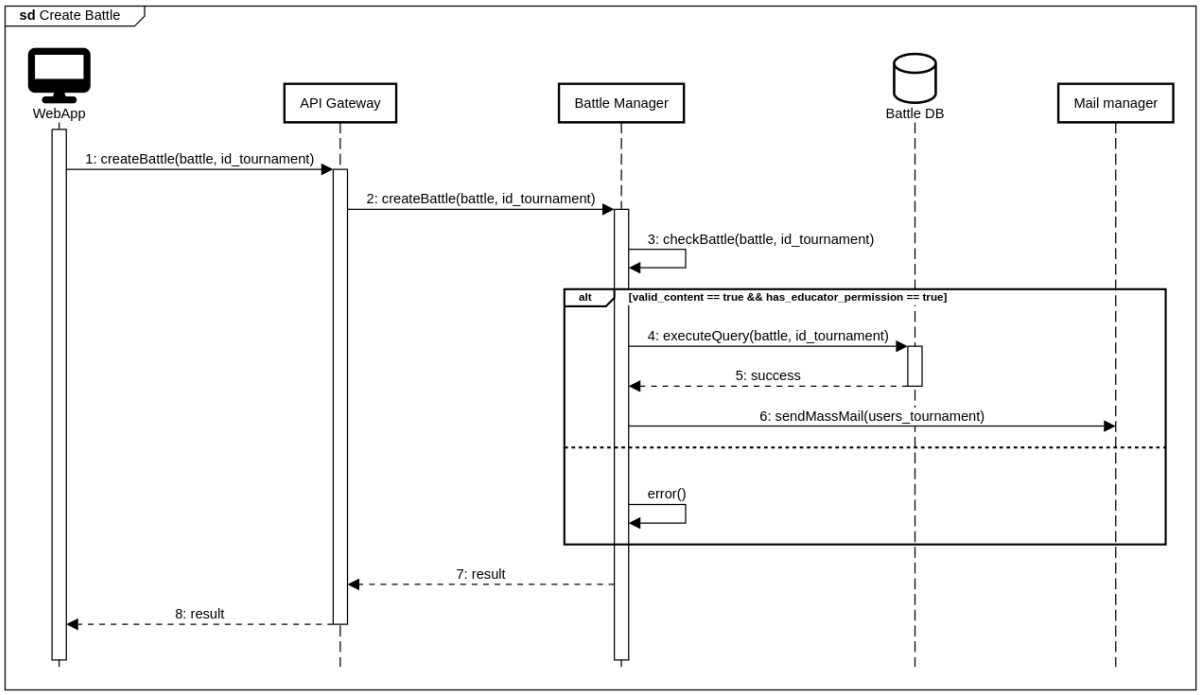
This sequence diagram represents the interaction that happens when a student wants to leave a battle in the CKB application.

[API Call]



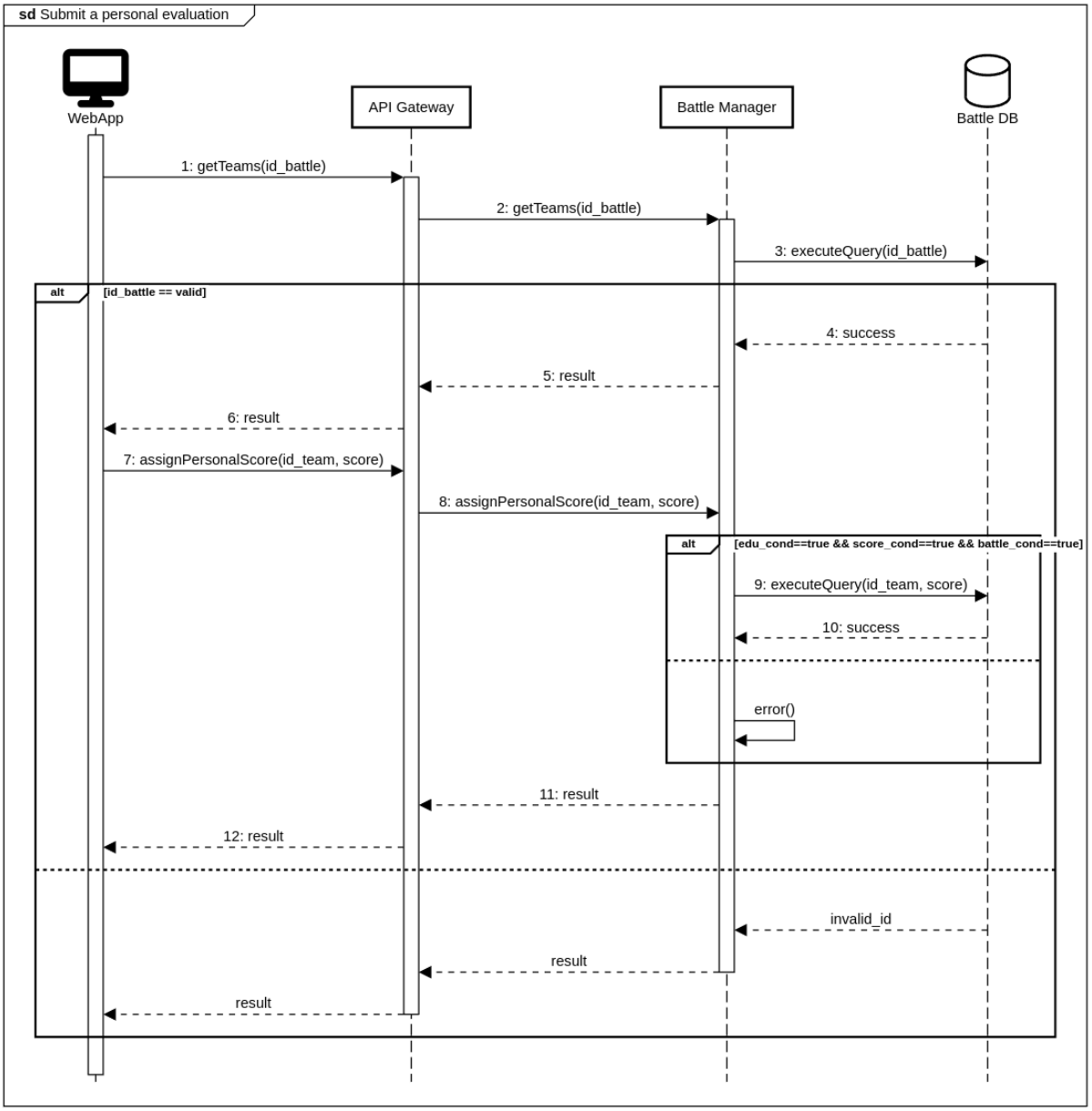
This sequence diagram represents the interaction that happens when the CKB application receives an API call from github by a team that wants to submit a solution to a battle.

[Create Battle]



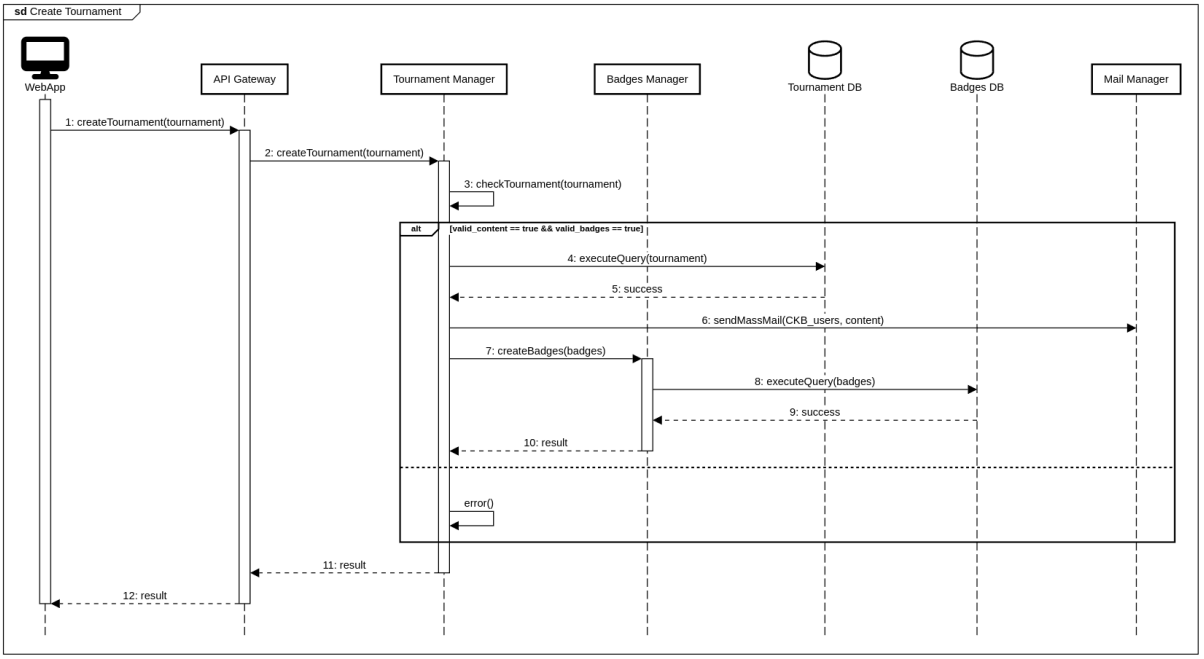
This sequence diagram represents the interaction that happens when an educator wants to create a battle in the CKB application.

[Submit a personal evaluation]



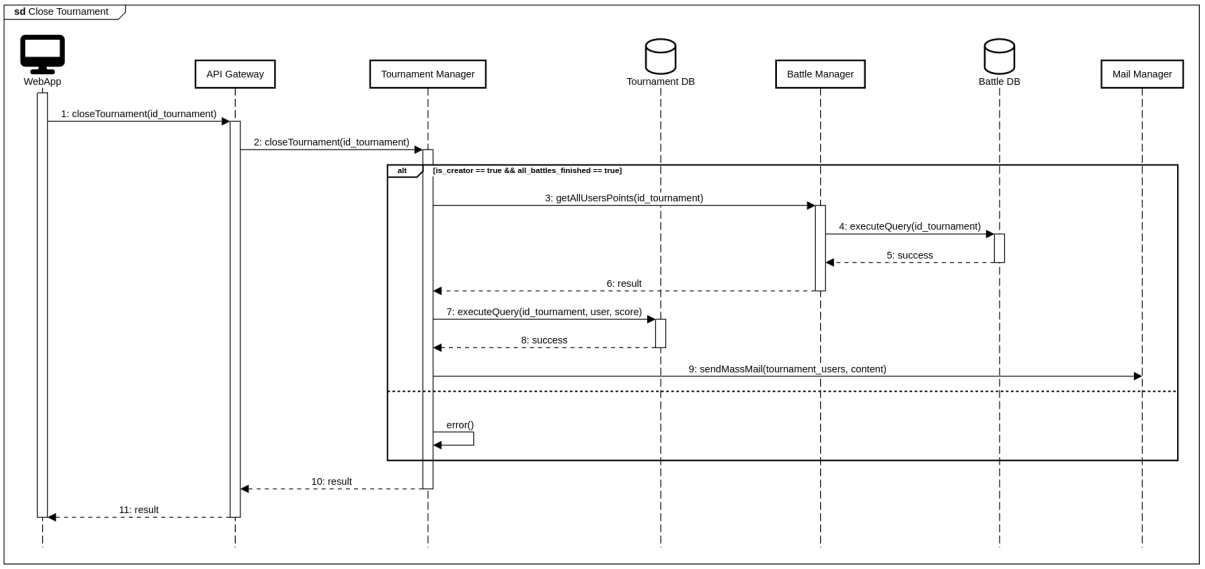
This sequence diagram represents the interaction that happens when an educator wants to assign a personal evaluation to a team in the CKB application.
The edu_cond are that the educator included the possibility to assign a personal score to students' solutions and that he/she created the battle.
The score_cond is that the score must be between 0 and 100.
The battle_cond is that the battle must be in the consolidation phase

[Create Tournament]



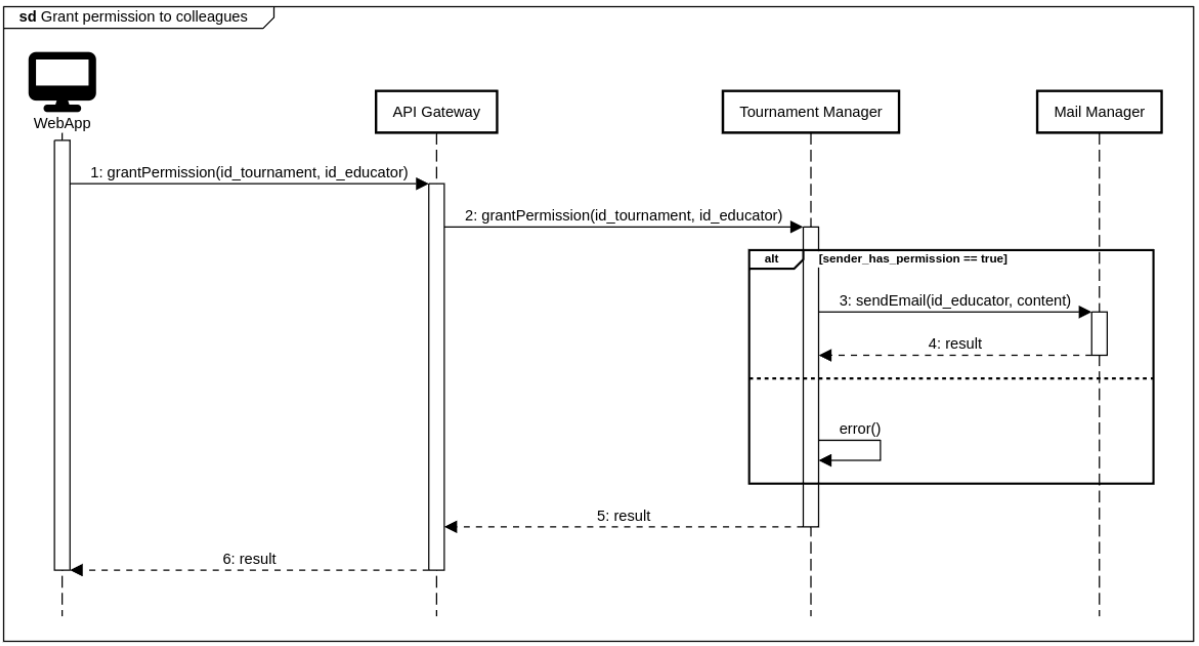
This sequence diagram represents the interaction that happens when an educator wants to create a tournament in the CKB application.

[Close Tournament]



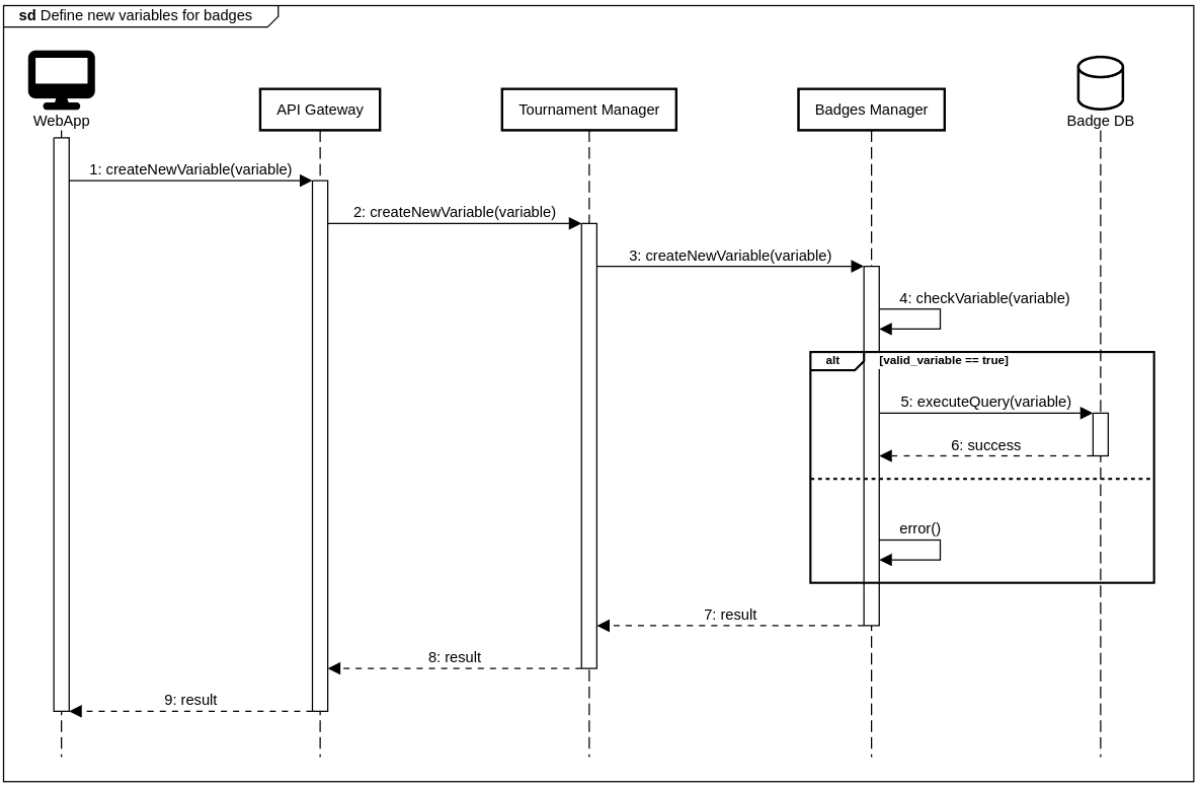
This sequence diagram represents the interaction that happens when an educator wants to close a tournament in the CKB application.

[Grant Permission]



This sequence diagram represents the interaction that happens when an educator wants to grant permission to a colleague to create a battle in a tournament in the CKB application.

[Define Variable]



This sequence diagram represents the interaction that happens when an educator wants to define a new variable for badges in the CKB application.

2.6 Selected architectural styles and patterns

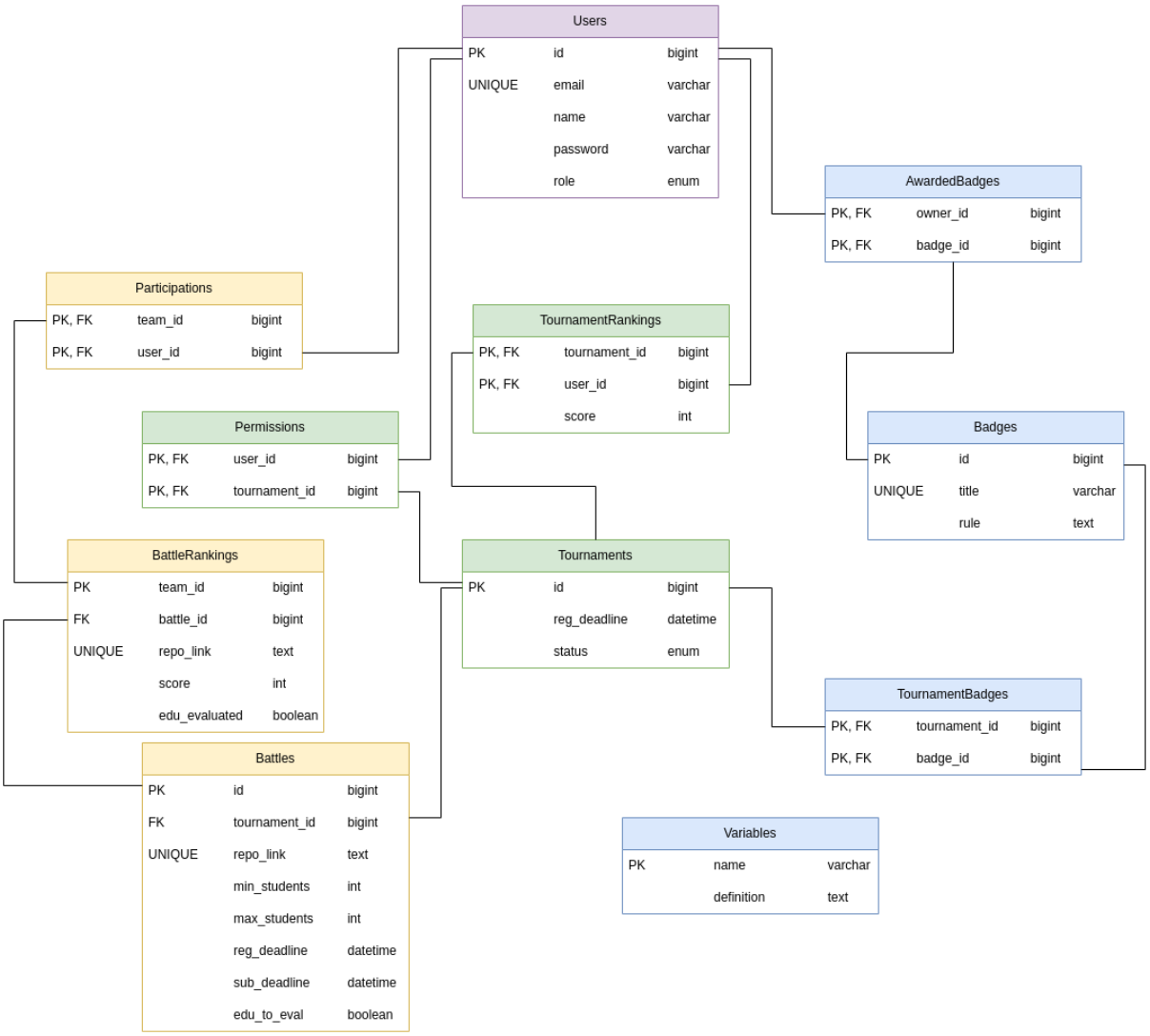
- **Microservices architecture**: the system is designed using a microservices styled architecture, to provide high decoupling among components and a high degree of scalability. This architectural style splits the system in multiple services, each focusing on satisfying a smaller set of requirements, allowing for reduced teams synchronization overhead, reduction in the size of the development teams and multiple smaller codebases, which lead to easier development, testing and debugging.
- **REST API**: the system provides a set of RESTful APIs for lightweight communication that users can exploit when interacting with the system. This choice goes very well with a microservices architecture because it provides a technology-neutral communication primitive, making underlying technical implementation of the single services irrelevant.
- **API gateway**: this design pattern acts a mediator between the users and the system, providing a common interface for users and also working as a gatekeeper for all traffic to microservices. The API gateway essentially abstracts away the internal composition of the system and could also act as a load balancer, forwarding requests evenly among the machines that provide the same back-end services.
- **Server-side service discovery**: all microservices contact the discovery service (whose ip address and port are well known) as soon as they start to communicate what service they offer and what their ip and open port is. When a service has to contact another one, it contacts the discovery service to get the port and address of the machine that provides the required service, then the communication happens directly. This design pattern increases decoupling among components because the addresses of the services do not need to be hard-coded in the services that need them, horizontal scalability is also made easier as the discovery service could also act as an internal load balancer and adding another machine to the system is as easy as contacting the discovery server.

2.7 Other design decisions

2.7.1 Database Structure

The following diagram represents the general relations between the tables present in the databases. It's important to note that this is just a logical representation and, although all tables are represented in the same diagram, they may be stored in physically different databases.
Moreover it’s a common occurrence in this architecture that foreign keys be stored in databases where the corresponding primary key is not present, this is done to increase redundancy, improving performance for requests among services.
Tables are represented in different colors depending on the actual physical database where they will be stored, more precisely:

- Purple tables belong in Account Manager
- Blue tables belong in Badges Service
- Green tables belong in Tournament Manager
- Yellow tables belong in Battle Manager


Users
PK id bigint
UNIQUE email varchar
name varchar
password varchar
role enum
AwardedBadges
PK, FK owner_id bigint
PK, FK badge_id bigint
Participations
PK, FK team_id bigint
PK, FK user_id bigint
TournamentRankings
PK, FK tournament_id bigint
PK, FK user_id bigint
score int
Permissions
PK, FK user_id bigint
PK, FK tournament_id bigint
Badges
PK id bigint
UNIQUE title varchar
rule text
BattleRankings
PK team_id bigint
FK battle_id bigint
UNIQUE repo_link text
score int
edu_evaluated boolean
Tournaments
PK id bigint
reg_deadline datetime
status enum
TournamentBadges
PK, FK tournament_id bigint
PK, FK badge_id bigint
Battles
PK id bigint
FK tournament_id bigint
UNIQUE repo_link text
min_students int
max_students int
reg_deadline datetime
sub_deadline datetime
edu_to_eval boolean
Variables
PK name varchar
definition text

3. User Interface Design

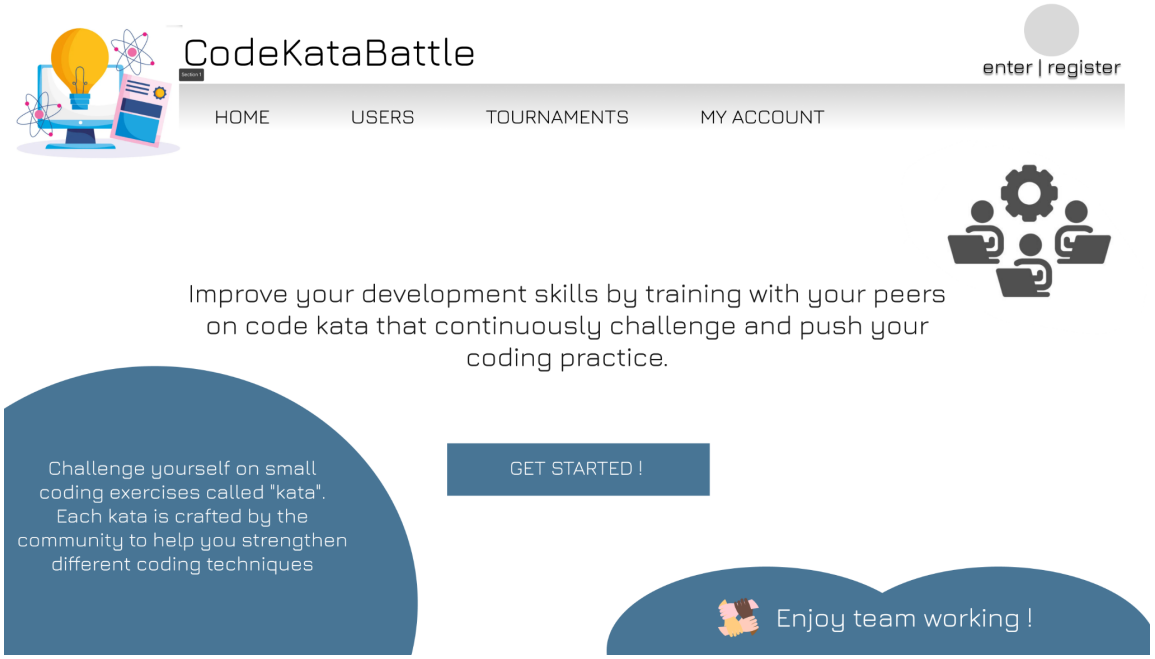


Fig.1 Home page

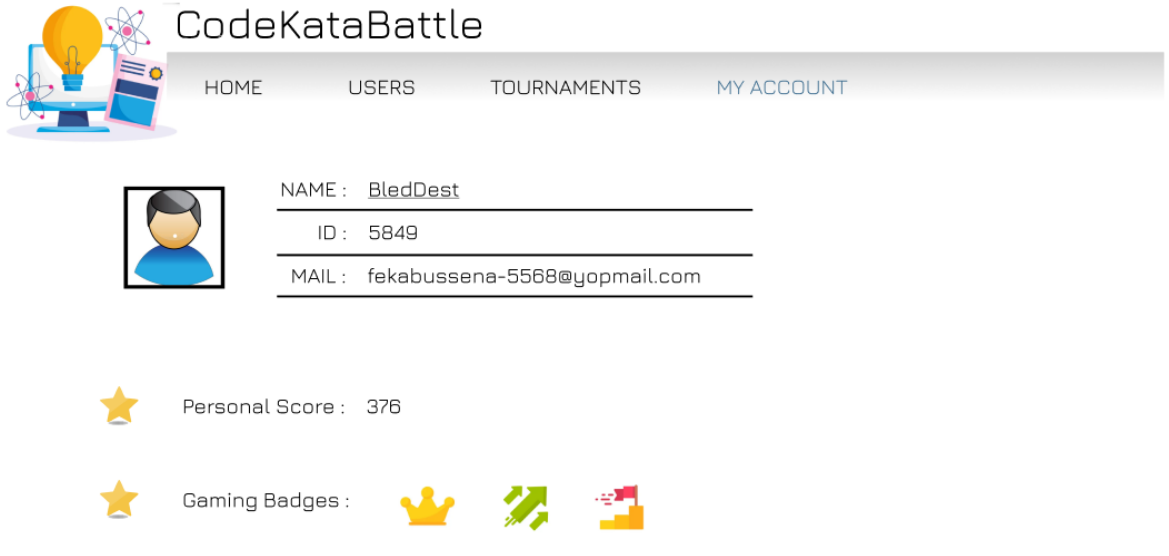


Fig.2 Personal page

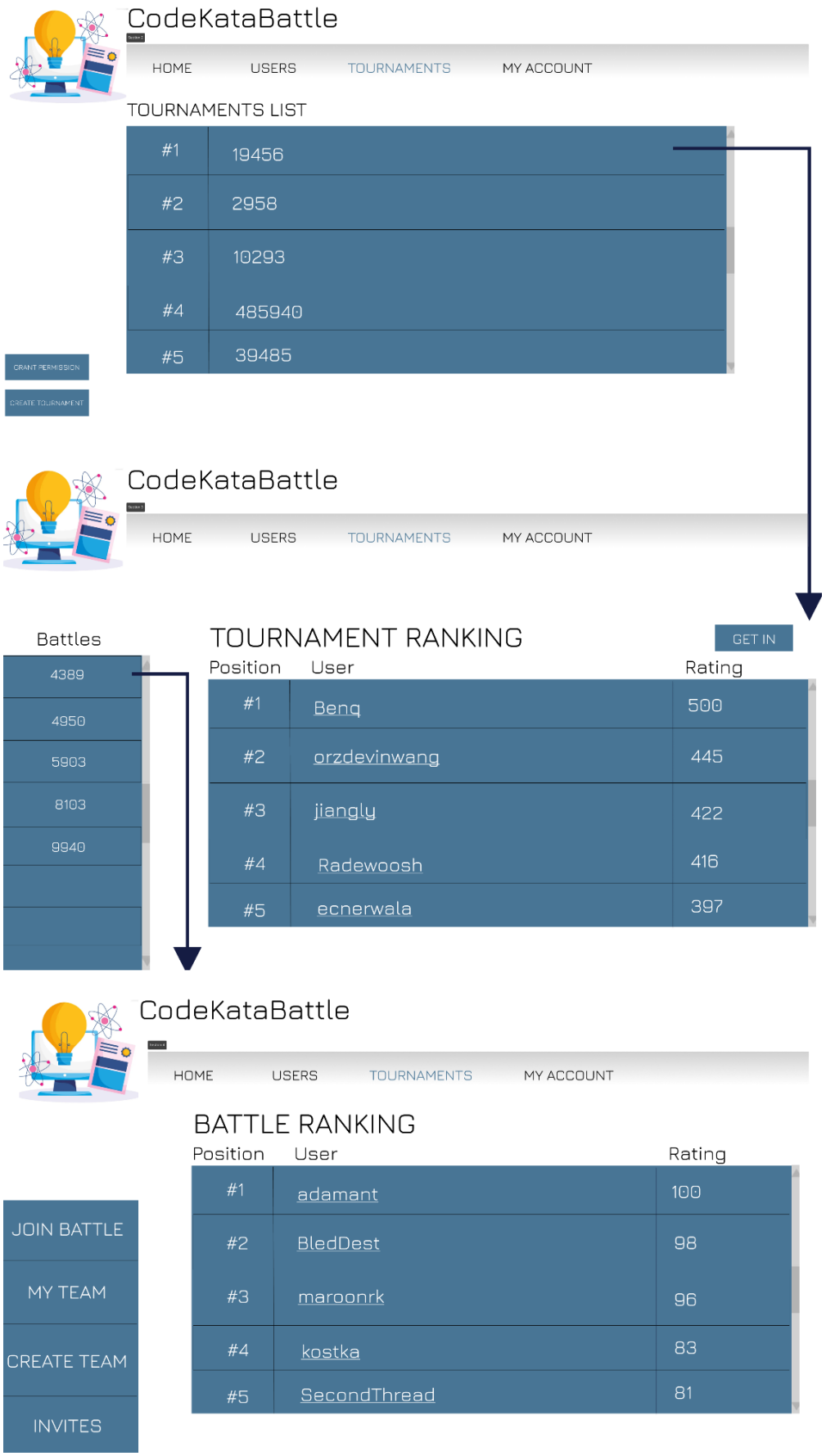


Fig.3 Tournament and Battle pages

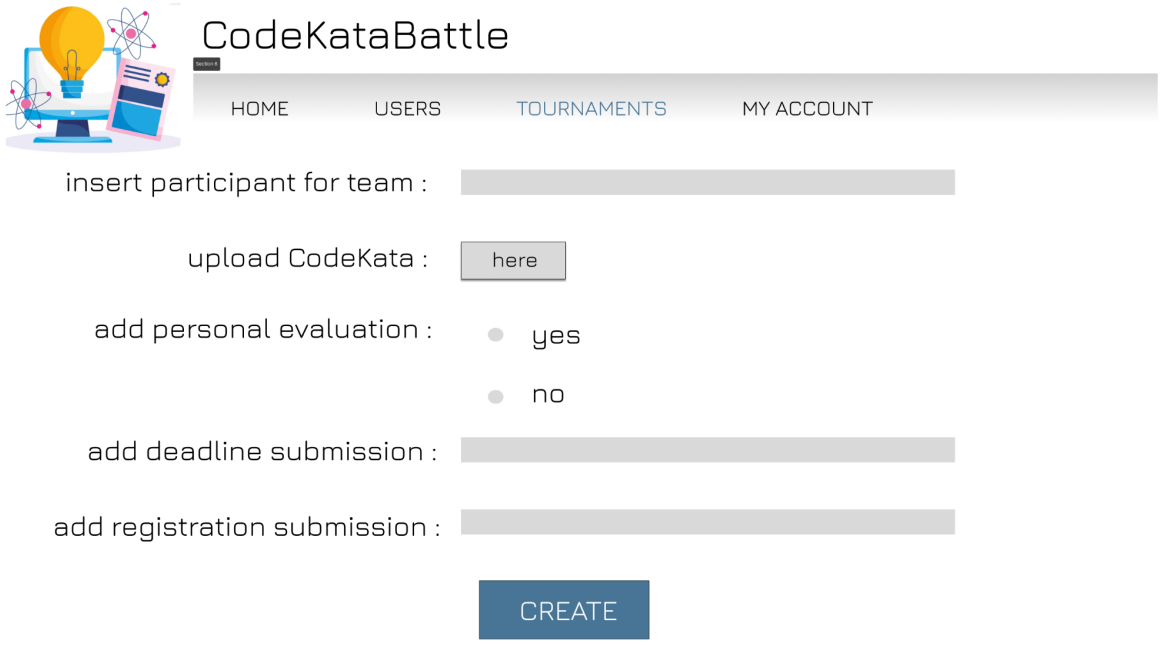


Fig.4 Battle creation page

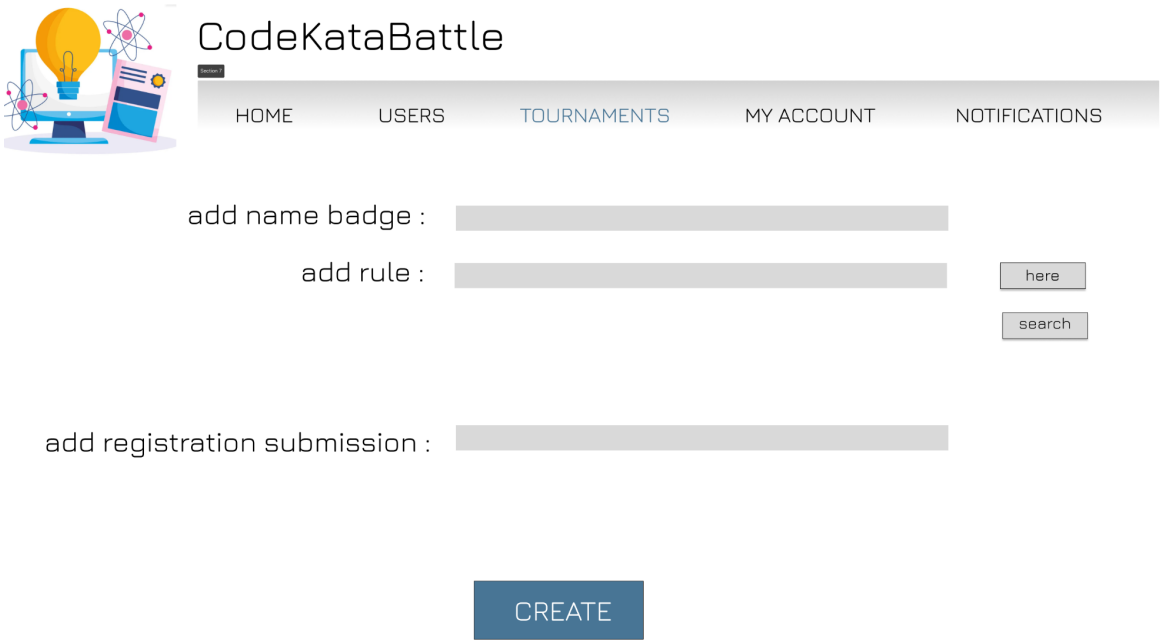


Fig.5 Tournament creation page

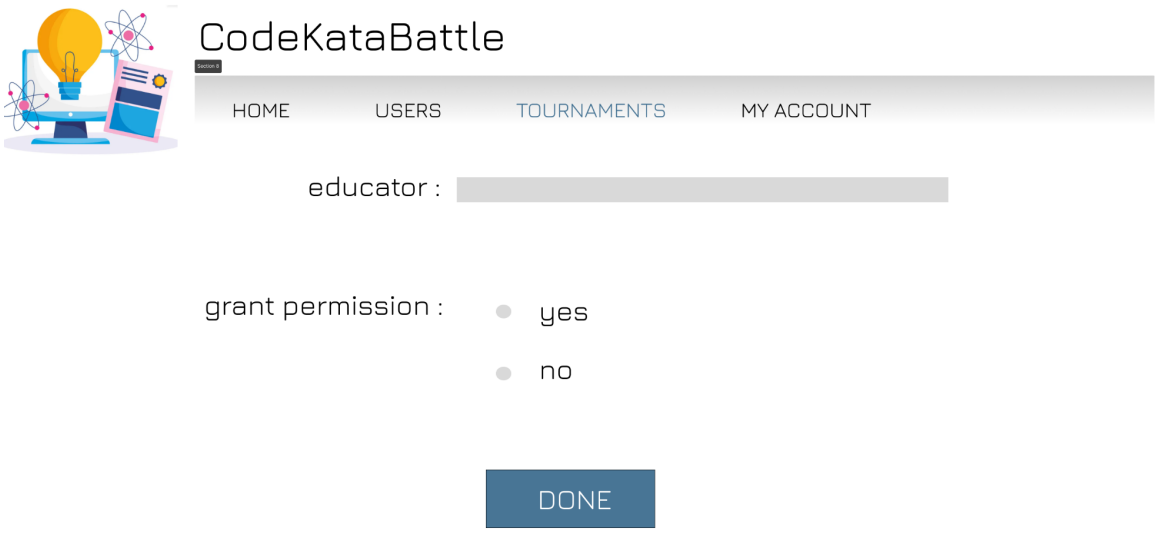


Fig.6 Gran permission page

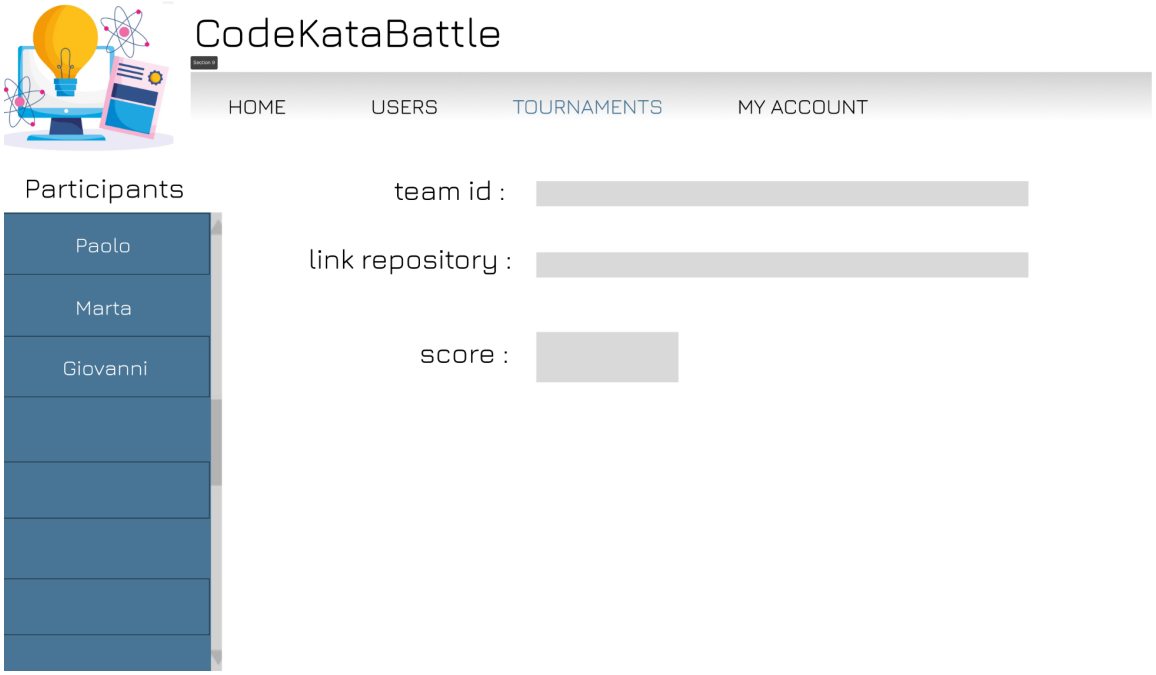


Fig.7 Team page

4. Requirements Traceability

This section shows which system components concur to the satisfaction of each requirement defined in the RASD document.

Requirements	**[R1]** The system allows users to sign up **[R2]** The system allows users to sign in **[R3]** The system allows users to browse other users profiles
Components	• Web App • Account Manager • DBMS

Requirements	**[R4]** The system allows users to browse the list of tournaments **[R5]** The system allows users to browse tournament rankings **[R7]** The system allows educators to create tournaments **[R8]** The system allows educators to specify a tournament registration deadline **[R9]** The system allows students to join tournaments **[R21]** The system allows educators to grant permission to create new coding battles for a tournament they have created to other educators
Components	• Web App • Tournament Manager • DBMS

Requirements	**[R6]** The system allows users to browse battle rankings **[R22]** The system allows students to create a group for each battle **[R24]** The system allows students to join a group for a battle **[R26]** The system allows educators to specify battle deadlines when creating a new battle **[R27]** The system allows educators to specify boundaries for the number of students in each group when creating a new battle **[R29]** The system allows educators to upload code katas when creating a new battle by providing a textual description, a set of test cases and build automation scripts
Components	• Web App • Battle Manager • DBMS

Requirements	**[R10]** The system allows educators to define gamification badges, consisting in a title and one or more rules that must be fulfilled for a student to obtain the badge
Components	• Web App

	• Badges Service • DBMS

Requirements	**[R11]** The system requires educators to define a way to assign scores to students submission in an automated way **[R12]** The system allows educators to decide whether they have to assign personal scores to students solutions during the consolidation phase
Components	• Web App • Battle Manager • Solution Evaluation Service

Requirements	**[R13]** The system allows educators to assign a personal score during the consolidation phase if they decided to allow it when creating the battle **[R28]** The system allows students and educators to see evolving rankings before a code battle has reached its submission deadline
Components	• Web App • Solution Evaluation Service • DBMS

Requirements	**[R14]** The system allows the creator of a battle to terminate the consolidation phase after having evaluated all of the groups sources (if they decided to do so when creating the battle), effectively terminating the battle
Components	• Web App • Battle Manager • Solution Evaluation Service • DBMS

Requirements	**[R15]** The system notifies all students subscribed to the platform whenever a new tournament is created **[R18]** The system notifies students when the final tournament ranking become available
Components	• Tournament Manager • Mail Service • DBMS

Requirements	**[R16]** The system notifies students when a battle is created if they are registered to that battle's tournament
Components	• Battle Manager • Tournament Manager

	• Mail Service • DBMS

Requirements	**[R17]** The system notifies students when the battle's final rankings become available
Components	• Solution Evaluation Service • Battle Manager • Mail Service • DBMS

Requirements	**[R19]** The system provides all students subscribed to a battle with that battles's code kata by notifying them with a link to that code kata's GitHub repository when a battle's registration deadline expires if they are subscribed
Components	• Battle Manager • GitHub Manager • Mail Service • DBMS

Requirements	**[R20]** The system allows educators to create coding battles for a specific tournament if they either have been given permission from the tournament creator to do so or they created that tournament
Components	• Web App • Battle Manager • Tournament Manager • DBMS

Requirements	**[R23]** The system allows students to invite other students to their group for a battle
Components	• Web App • Battle Manager • Mail Service • DBMS

Requirements	**[R25]** The system allows students to compete in a coding battle if, when the registration deadline expires, they are part of a team composed by a number of students within the boundaries defined by that battle's creator
Components	• Battle Manager • Mail Service • DBMS

Requirements	**[R30]** The system provides an API to allow users to submit their solution to a code battle, triggering the system to run automated tests to analyze the students code
Components	● CKB API ● GitHub Manager ● Solution Evaluation Service ● DBMS

Requirements	**[R31]** The system allow educators to create new variables to use during the definition of the rules for gamification badges, using a pseudo-language
Components	● Web App ● Tournament Manager ● Badges Service ● DBMS

For the sake of simplicity and conciseness, the figure below represents the same relations but in a more compact form, using a traceability matrix.

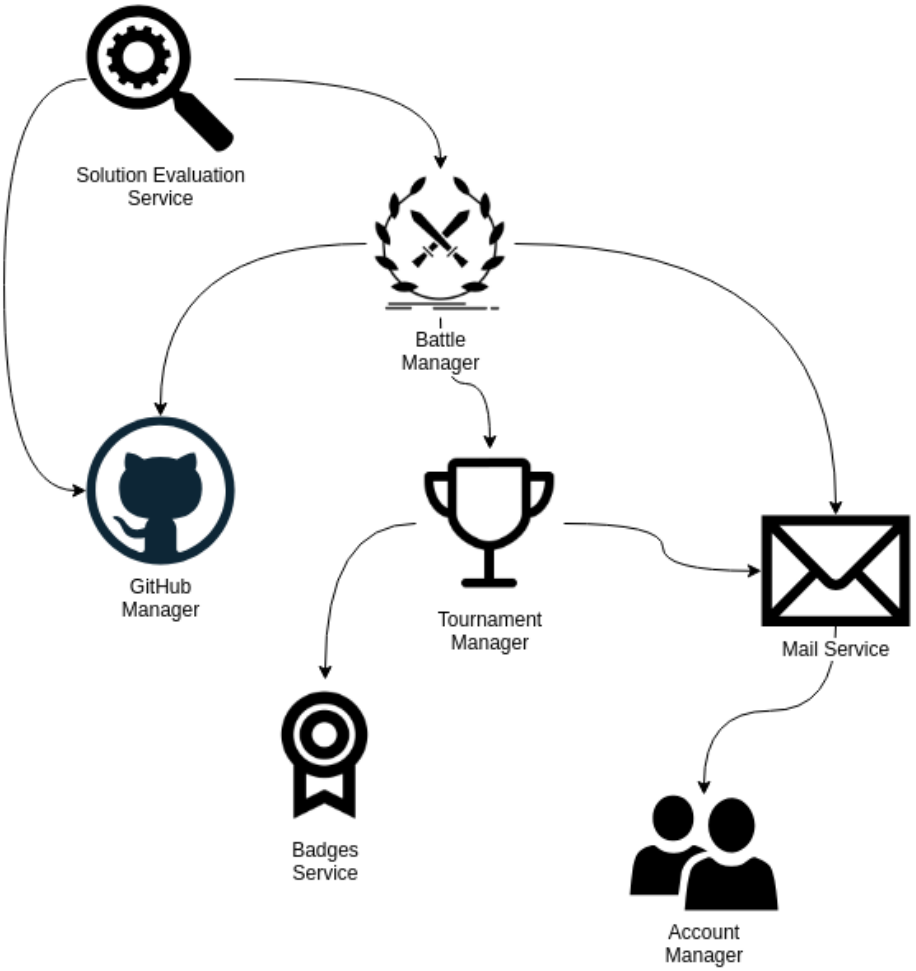
	Battle Manager	Tournament Manager	Mail Service	GitHub Manager	Solution Evaluation Service	Badges Service	DBMS	Account Manager	Web App	CKB API
R1							x	x	x	
R2							x	x	x	
R3							x	x	x	
R4		x					x		x	
R5		x					x		x	
R6	x						x		x	
R7		x					x		x	
R8		x					x		x	
R9		x					x		x	
R10						x	x		x	
R11	x				x				x	
R12	x				x				x	
R13					x		x		x	
R14	x				x		x		x	
R15		x	x				x			
R16	x	x	x				x			
R17	x		x		x		x			
R18		x	x				x			
R19	x		x	x			x			
R20	x	x					x		x	
R21		x	x				x		x	
R22	x						x		x	
R23	x		x				x		x	
R24	x						x		x	
R25	x		x				x			
R26	x						x		x	
R27	x						x		x	
R28					x		x		x	
R29	x						x		x	
R30				x	x		x			x
R31		x				x	x		x	

5. Implementation, Integration and test Plan

Since the system is to be developed in a microservices architecture, all the different services can be implemented and tested in parallel by different teams, the interactions between services can be simulated during implementation and testing.
The single services can essentially be implemented and tested as stand-alone units and testing for individual services should be done by first performing unit tests as soon as a component is developed to ensure it’s working correctly, after multiple components are developed and unit tested, they can be integrated together and integration testing can be performed.
At a larger scale, testing for services can be carried out following the same logic, after a service is developed, it should be tested as a stand-alone unit, assuring that its functions are carried out correctly by simulating any eventual interactions with other services. After a service has been thoroughly tested, it can be tested together with other components, to ensure interactions among services work correctly, without simulation.

5.1 Services Integration plan

Since some services rely on the correct interaction of others, it's important to define an integration plan to decide in which order the single services have to be integrated together. To do that we identify which services are used by which other, so we can first integrate the services that are to be used by other services.
From that, we work out the following graph:



Since the graph is not cyclic, we can define a clear order in which to integrate the services with one another, starting from the nodes not having any outgoing arcs. then deleting them and repeating the process.
Note that we decided to take this approach to enable the system to be built (i.e. integrated) with a step-by-step approach, making it possible to see the system working as soon as possible and evolving as features are added to it.

The order in which services are integrated is decided using the graph as follows:

- Integrate nodes without outgoing edges
- When a node is fully integrated and tested (using the services it needs without simulating interactions), remove it from the graph
- When a node is deleted delete it's ingoing edges

Note that as integration tests are carried out (integration among multiple services) on a branch of the graph, if bugs with services that have already been fully integrated and tested arise, no other nodes using that branch should be integrated until the bugs are fixed.

6. Effort Spent

Tommaso Fellegara

Introduction	
Architectural Design	12
User Interface Design	
Requirements Traceability	1
Implementation, Integration and test Plan	
miscellaneous	1

Manuela Marenghi

Introduction	
Architectural Design	7
User Interface Design	5
Requirements Traceability	1
Implementation, Integration and test Plan	
miscellaneous	1

Cattani Luca

Introduction	
Architectural Design	5
User Interface Design	
Requirements Traceability	4
Implementation, Integration and test Plan	2
miscellaneous	1

7. References

- Sequence diagrams made with: https://sequencediagram.org/
- User Interface mockups made with: https://www.figma.com/
- Component view made with: https://app.diagrams.net/