



POLITECNICO DI MILANO
Computer Science and Engineering

Design Document

CodeKataBattle

Authors:

Manuela Marengi
Luca Cattani
Tommaso Fellegara

Professor:

Matteo Giovanni Rossi

Table of Contents

CodeKataBattle	1
Authors:	1
Professor:	1
Table of Contents	2
1. Introduction	4
1.1 Scope	4
1.1.1 Product domain	4
1.1.2 Main architectural choices	5
1.2 Definitions, acronyms, abbreviations	5
1.2.1 Definitions	5
1.2.2 Acronyms	6
1.2.3. Abbreviations	6
1.3 Overview	6
2. Architectural Design	7
2.1 Overview: high-level components and interactions	7
2.2 Component view	8
2.3 Deployment view	11
2.4 Component interfaces	12
2.4.1 Account Manager	12
Account Interface	12
2.4.2 Badges Manager	13
Badges Interface	13
2.4.3 Tournament Manager	13
Tournament Interface	13
2.4.4 Mail Manager	14
Mail Interface	14
2.4.5 Battle Manager	15
Battle Interface	15
2.4.6 Solution Evaluation Service	17
Solution Evaluation Interface	17
2.4.7 GitHub Manager	17
GitHub Interface	17
2.5 Runtime view	18
2.6 Selected architectural styles and patterns	32
2.7 Other design decisions	32
2.7.1 Database Structure	32
3. User Interface Design	34
4. Requirements Traceability	38
5. Implementation, Integration and test Plan	43
5.1 Services Integration plan	44
6. Effort Spent	45
Tommaso Fellegara	45

Manuela Marengi	45
Cattani Luca	45
7. References	46

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 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

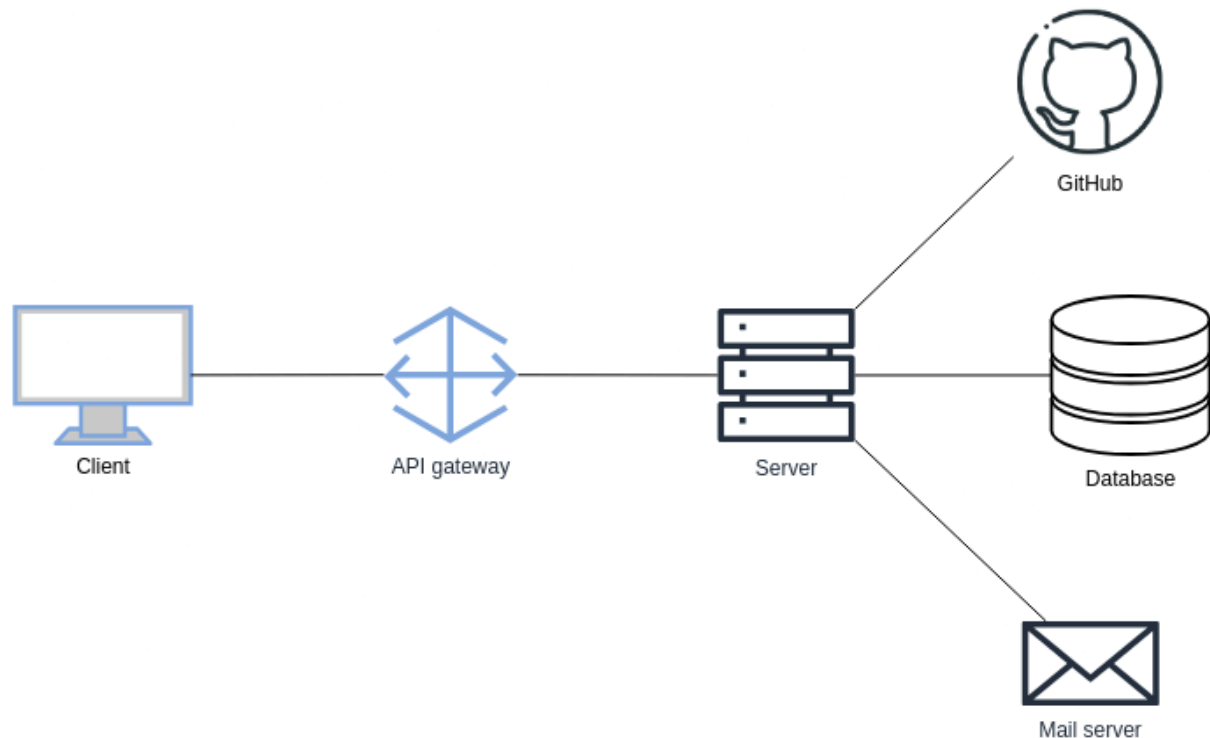
Another key part of the document is the requirements traceability chapter, here the relations between requirements and design elements is highlighted, providing a view of how the structure of the system is designed to satisfy requirements at best.

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).

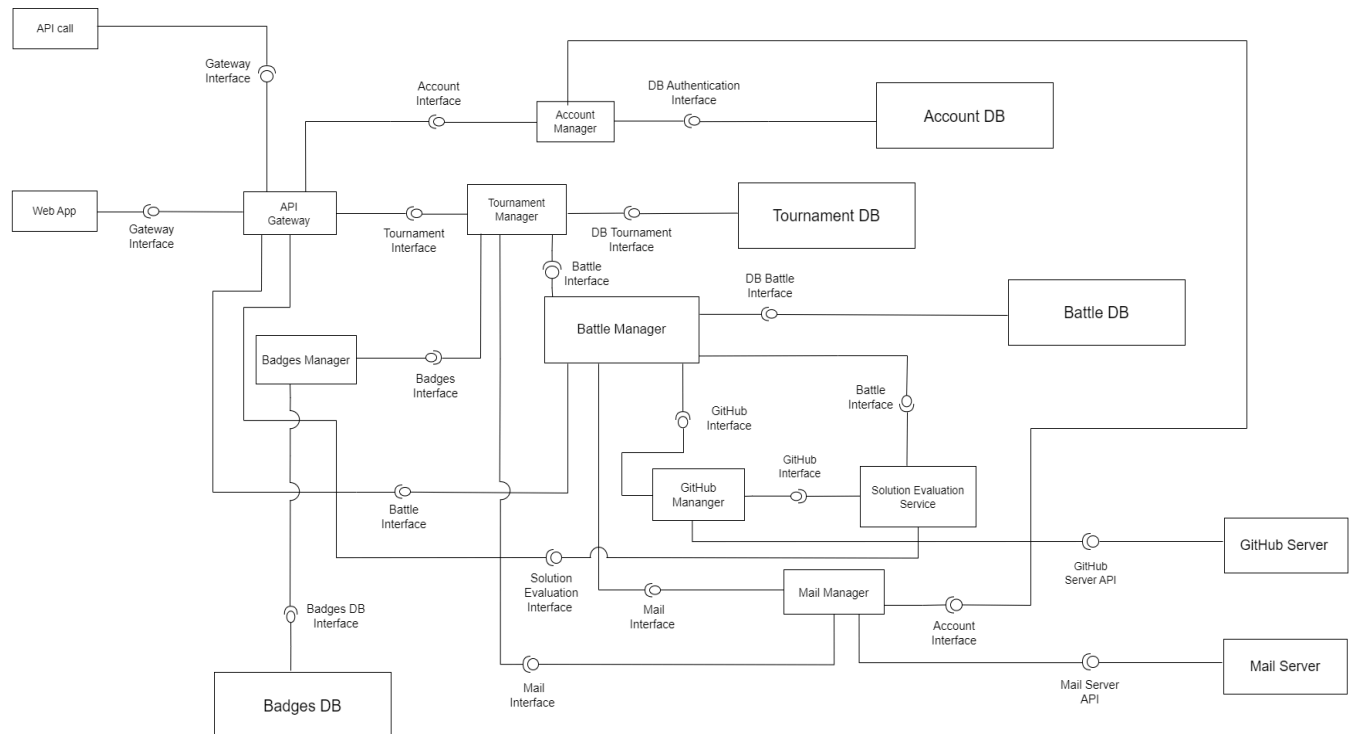


Whenever a client interacts with the system, its request is processed by an API gateway, which directs the request to the designed service, the server may then interact with other services depending on the interaction:

- 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. For simplicity we omitted to represent the discovery server.



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

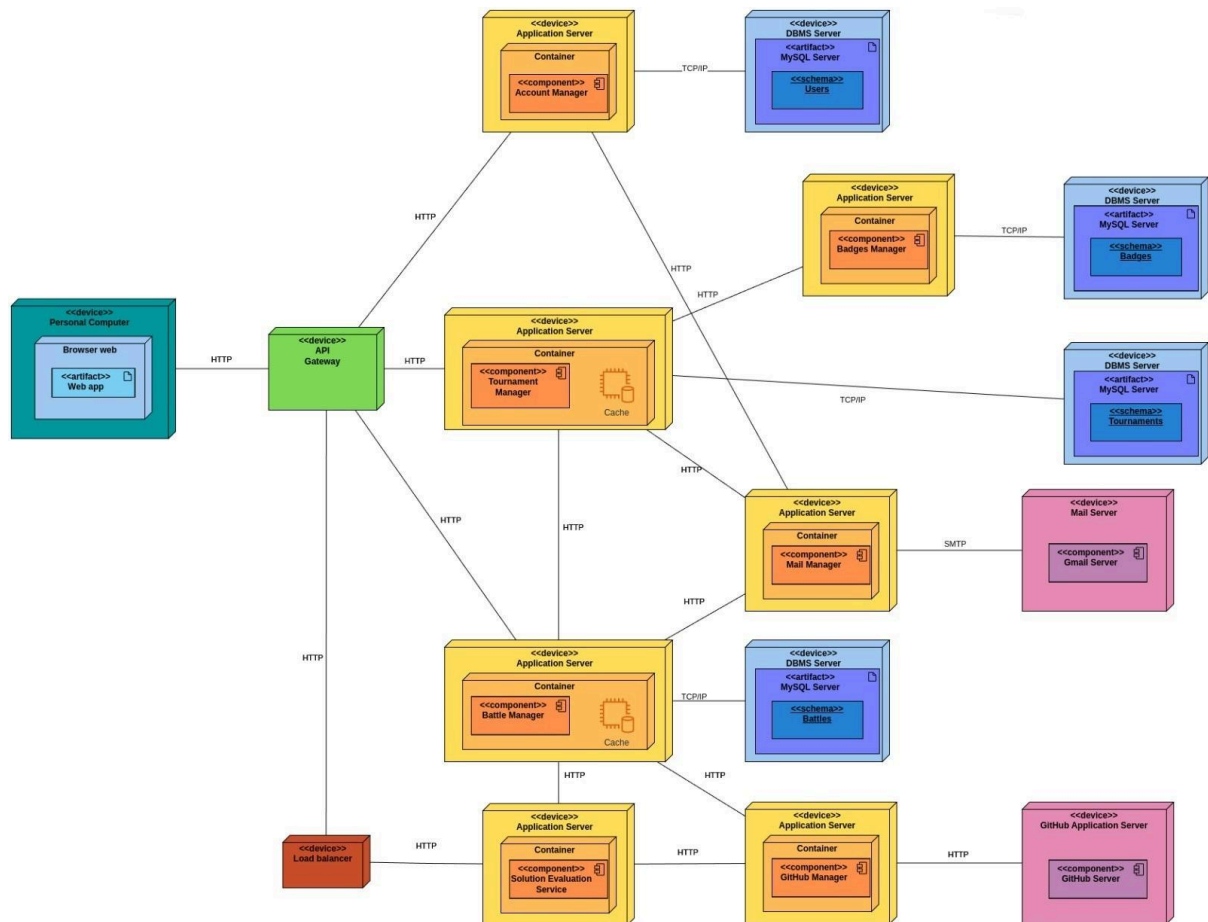
- **Web App** : represents the website used by each type of user of the system. By using the **Gateway Interface** it sends requests to the **API Gateway** that has the responsibility of redirecting each request to the right microservice that can handle it. Each user would have to use an arbitrary web 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.
- **Eureka Server** : this component lets all services communicate together. Each one of them subscribes to this server including information such as the service name, instance ID, and network location. In this way each service can ask this component and dynamically discover other instances of services. In this way it is also possible to load balanced microservices following the demand.

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 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.

Obviously all the microservices are connected to the Eureka server, but for the sake of simplicity the links and the Eureka server are not drawn.

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

POST: `"/api/account/mail"`

- purpose: retrieve a list of emails given the ids of the accounts
- response 200: the user has an email address
- response 404: the user has not got an email address
 - `List<String> getMail(List<Long> userIDs)`

GET: `"/api/account/mail-students"`

- purpose: retrieve a list of emails given the ids of the accounts
- response 200: the email list of the student is retrieved without errors
 - `List<String> getStudentMails()`

POST: `"/api/account/sign-in"`

- purpose: access the user to his/her account
- response 200: the user has sent the correct email and password
- response 404: the user has sent wrong email or password
 - `void signIn(String email, String password)`

POST: `"/api/account/sign-up"`

- purpose: subscribe a user to the CKB platform
- response 201: the account for the user is created
- response 400: the user has sent an email that is already used or wrong information
 - `void signUp(String email, String fullName, String password, Role role)`

POST: `"/api/account/update"`

- purpose: update the information of a user
- response 200: the information are successfully updated
- response 400: the user has sent wrong information
- response 500: there was an error updating the information
 - `void update(Long id, String email, String fullName, String password, Role role)`

POST: `"/api/account/user"`

- purpose: get the user information
- response 200: the information are retrieved successfully

- response 400: the user has not been found
 - User getUser(Long id)

2.4.2 Badges Manager

Badges Interface

A badge is an object that contains a title and a rule to achieve.

The rule is composed of the variables created by the educators.

POST: "/api/badge/create-badges"

- purpose: Create the badges for a tournament
- response 200: the badges are correctly created
- response 400: there is an error in the request
 - void createBadges(Long tournamentID, List<Badge> badges)

POST: "/api/badge/get-badges"

- purpose: Get all the badges achieved by an user
- response 200: the badges are correctly sent
- response 400: there is an error in the request
 - void createBadges(Long userID)

POST: "/api/badge/create-new-variable"

- purpose: create a new variable to create a rule
- response 200: the variable is correctly created
- response 400: there is an error in the request
 - void createNewVariable(Variable variable)

A variable can represent any piece of information available in CKB relevant for scoring

2.4.3 Tournament Manager

Tournament Interface

POST: "/api/tournament/check-permission"

- purpose: check the permission of an user
- response 200: the user has the permission to create a battle in the tournament
- response 404: the user has not the permission
 - void checkPermission(Long tournamentID, Long userID)

POST: "/api/tournament/close-tournament"

- purpose: close a tournament
- response 200: the tournament has been closed
- response 400: the tournament cannot be closed because some battles are not closed or the tournament does not exists or the user has not the privileges to close it
 - void closeTournament(Long tournamentID, Long userID)

POST: "/api/tournament/get-all-tournaments"

- purpose: get all the tournaments

- response 200: all the tournaments of the CKB platform are retrieved
 - `HashMap<Long, String> getTournaments()`

POST: `"/api/tournament/get-tournament-page"`

- purpose: get the page of the tournament
- response 200: the tournament page is retrieved
- response 400: the request is invalid or there is an error contacting the battle manager
 - `ResponseWrapper getTournamentPage(Long tournamentID)`
The `ResponseWrapper` is a class that consist in a list of longs that represent the ids of the battles, and the ranking of the tournament

POST: `"/api/tournament/inform-students"`

- purpose: inform all the students of a tournament about the creation of a battle
- response 200: successfully informed all the students of the tournament
- response 400: error while sending the emails to the students
 - `void informStudents(Long tournamentID, String battleName)`

POST: `"/api/tournament/new-tournament"`

- purpose: create a new tournament
- response 201: a new tournament is successfully created
- response 400: error while creating the new tournament
 - `void newTournament(Long creatorID, String name, Date registrationDeadline)`

POST: `"/api/tournament/permission"`

- purpose: give a permission to an educator to create battles in a tournament
- response 201: a permission is given
- response 400: error while giving the permission to another educator
 - `void createPermission(Long tournamentID, Long userID, Long creatorID)`

POST: `"/api/tournament/subscription"`

- purpose: subscribe a student to a tournament
- response 201: the subscription process has successfully finished
- response 400: the request has an error
 - `void subscription(Long tournamentID, Long userID)`

POST: `"/api/tournament/update-score"`

- purpose: update the score of a tournament
- response 200: the scores have been updated
- response 400: an error occurred during the update of the scores
 - `void updateScore(Long tournamentID, List<Pair<Long, Long>> userIDScore)`

2.4.4 Mail Manager

Mail Interface

POST: `"/api/mail/all-students"`

- purpose: send the mails to all the students subscribed in the CKB platform

- response 200: the mails are sent to the students
- response 400: an error occurred while retrieving or sending the emails
 - void mailAllStudents(String content)

POST: "/api/mail/direct"

- purpose: send the mails to the users specified in the input
- response 200: the mails are sent to the users
- response 400: the request is invalid or an error occurred while retrieving or sending the emails
 - void sendEmail(List<Long> userIDs, String content)

2.4.5 Battle Manager

Battle Interface

POST: "/api/battle/assign-score"

- purpose: assign the score of a team after having received a request by the solution evaluation manager
- response 200: successfully assigned the score to the team
- response 400: there is an error during the process of assigning the score
 - void assignScore(Long teamID, Integer score)

POST: "/api/battle/assign-personal-score"

- purpose: assign the personal score of the educator to a team
- response 200: successfully assigned the personal score of the educator to the team
- response 400: there is an error during the process of assigning the personal score
 - void assignPersonalScore(Long teamID, Integer score, Long educatorID)

POST: "/api/battle/battles-finished"

- purpose: request to the battle manager if all the battles are closed
- response 200: all the battles are finished
- response 400: there is at least one battle that is not closed yet
 - void canCloseTournament(Long tournamentID)

POST: "/api/battle/close-battle"

- purpose: request to the battle manager if a battle can be closed
- response 200: the battle can be closed
- response 400: the battle cannot be closed
 - void closeBattle(Long battleID, Long creatorID)

POST: "/api/battle/create-battle"

- purpose: create a battle
- response 200: the battle is created successfully
- response 400: there is an error in the request
- response 500: there is an error during the creation of the battle
 - void createBattle (MultipartFile zipFile, Long tournamentId, Long authorId, Integer minStudents, Integer maxStudents, LocalDateTime regDeadline, LocalDateTime subDeadline, String name)

For testing purposes another endpoint has been created with the following path
“api/battle/create-battle” with the same arguments except for the zipFile that is represented
by a Pair<String, String> where the left object is the path of the file and the second is the
content

POST: “/api/battle/get-battles-tournament”

- purpose: get all the battles that are in a tournament
- response 200: the ID of the battles of the tournament specified by the ID
 - List<Long> getBattlesOfTournament(Long tournamentID)

POST: “/api/battle/evaluation-params”

- purpose: get the repository url of the battle
- response 200: the repository url of the battle is retrieved
- response 400: the team doesn't exists
 - EvaluationParamsResponse evaluationParams(Long teamID)
EvaluationParamsResponse is an object composed by the official repository
url and if the solution evaluation service has to evaluate the security, the
reliability and the maintainability of the program

POST: “/api/battle/get-team”

- purpose: get the participants of a team
- response 200: get all the participants of a team
- response 404: the team has not been found
 - List<TeamInfoMessage> getTeam(Long battleID, Long studentID)
TeamInfoMessage: class used to wrap the participants name and the id of
the team

POST: “/api/battle/get-teams-battle”

- purpose: get the list of team IDs that participate to the battle and their score
- response 200: all the teams and their score are retrieved
- response 400: the battle does not exist
 - List<Pair<Long, Integer> getTeamsOfBattle(Long battleID)

POST: “/api/battle/invite-student-to-team”

- purpose: invite a student to the personal team
- response 200: successfully invited the student to the team through an email
- response 400: an error occurred during the process
 - void inviteStudentToTeam(Long studentID, Long teamID)

POST: “/api/battle/join-battle”

- purpose: join the battle
- response 200: successfully joined the battle
- response 400: an error occurred during the process
 - void joinBattle(Long studentID, Long BattleID)

POST: “/api/battle/leave-battle”

- purpose: leave the battle
- response 200: successfully leaved the battle
- response 400: an error occurred during the process

- void leaveBattle(Long studentID, Long battleID)

POST: “/api/battle/register-student-team”

- purpose: a student has accepted the invite and wants to join the team
- response 200: successfully joined the team
- response 400: an error occurred during the process
 - void registerStudentToTeam(Long studentID, Long teamID)

2.4.6 Solution Evaluation Service

Solution Evaluation Interface

POST: “/api/solution-evaluation/c”

- purpose: evaluate the submission of a team
- response 200: the submission is evaluated correctly
- response 500: an internal error occurred during the evaluation or during the update of the score of the team
 - void evaluate(String repositoryUrl, Long teamID)

2.4.7 GitHub Manager

GitHub Interface

POST: “/api/github/create-repo”

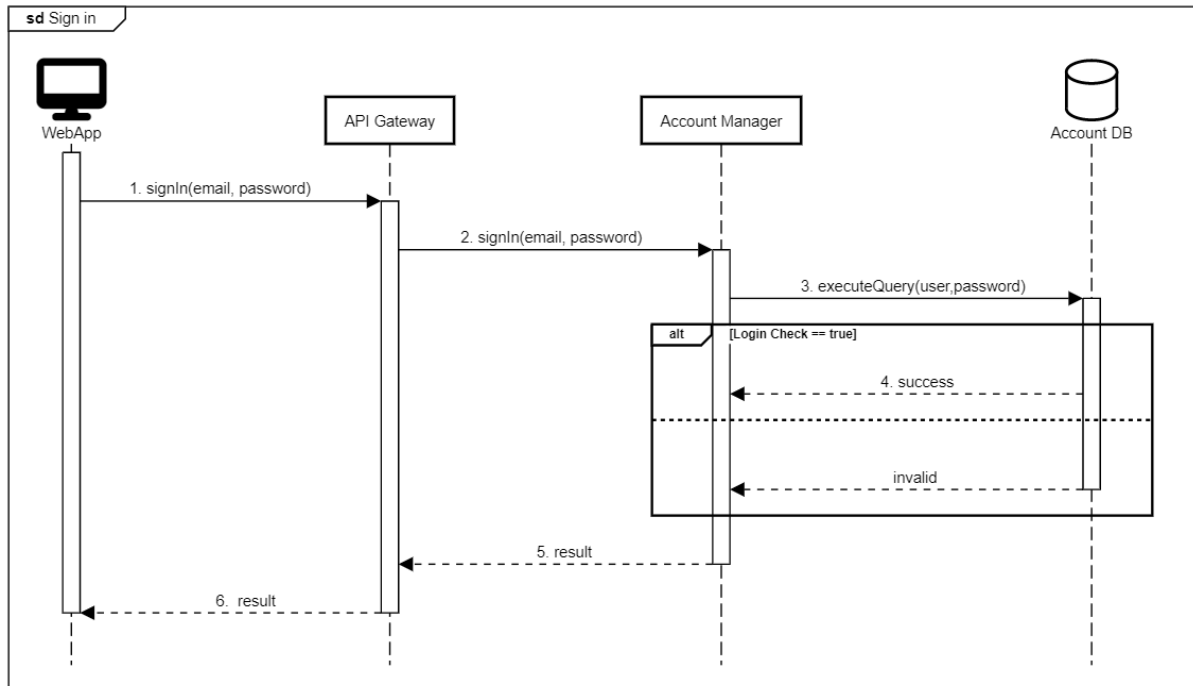
- purpose: create a private repository on Github and return the link to that repository
- response 201: the repository is successfully created
- response 500: an error occurred while creating the repository or during the commit and push of the files
 - String createBattleRepository(CreateRepositoryRequest request)
CreateRepositoryRequest is a wrapper that contains the name of the repository and a List of Pairs<String, String> in which there are the path and the content of the files that will be committed and pushed

POST: “/api/github/make-public”

- purpose: make public a repository on Github
- response 200: the repository has been made public
- response 500: an error occurred while changing the visibility
 - void makePublic(String repositoryName)

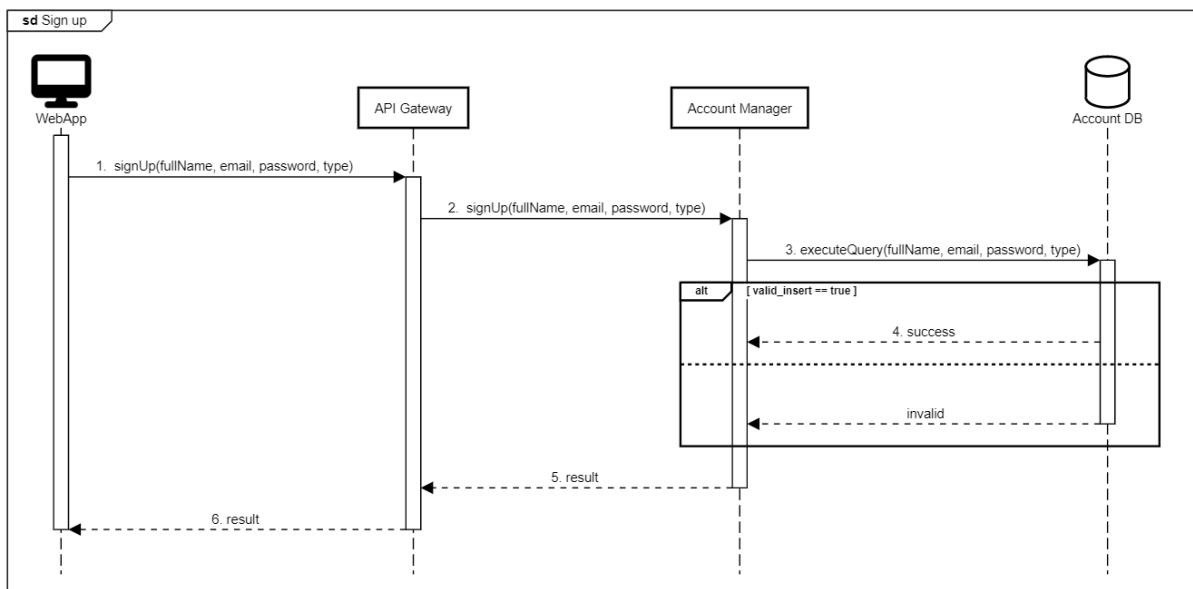
2.5 Runtime view

[Sign in]



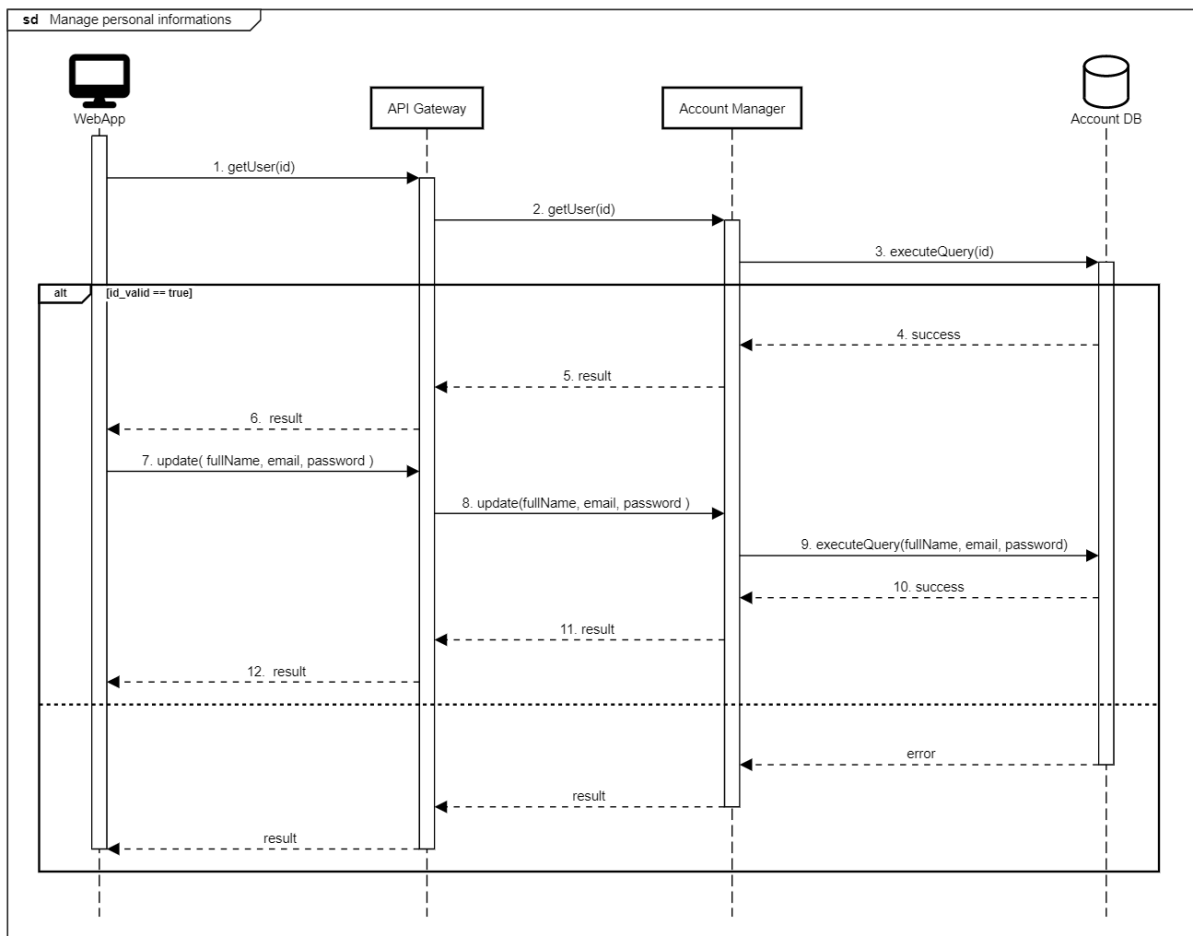
This sequence diagram represents the interaction that happens when a user wants to sign in to the CKB application assuming that the student is not.

[Sign up]



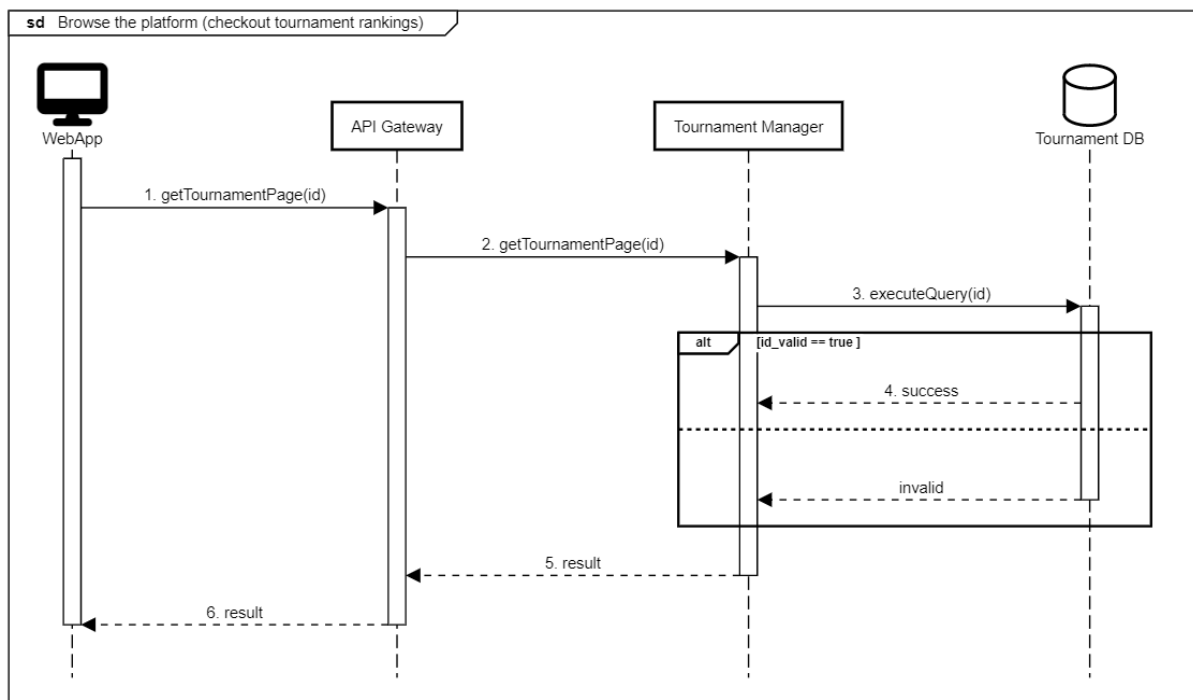
This sequence diagram represents the interaction that happens when a user wants to sign up to the CKB application assuming that the student is not.

[Manage personal informations]



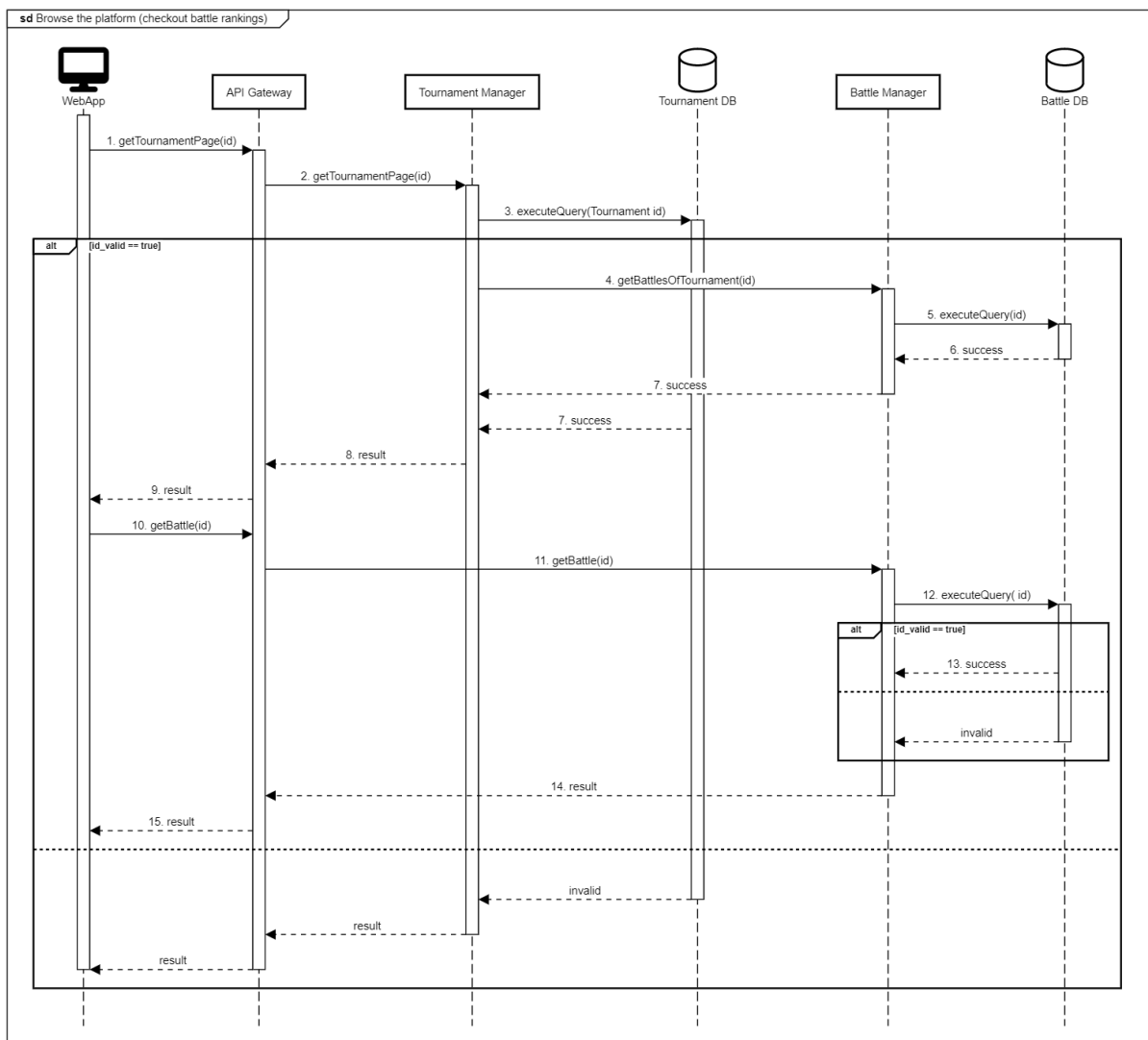
This sequence diagram represents the interaction that happens when a user wants to manage its personal information in the CKB application.

[Browse the platform (checkout tournament rankings)]



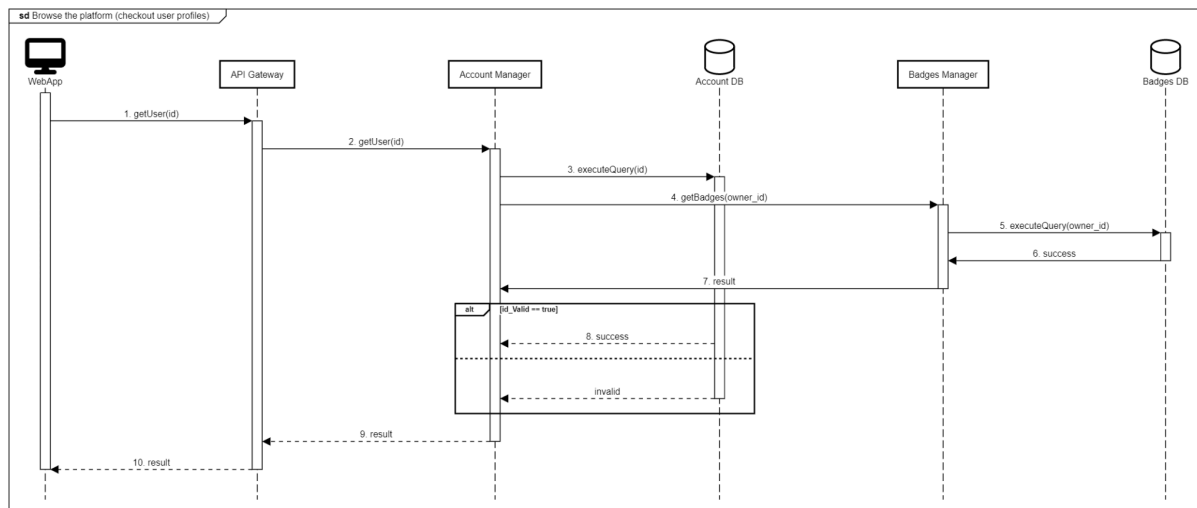
This sequence diagram represents the interaction that happens when a user wants to checkout tournament rankings that equals reaching the specific tournament page.

[Browse the platform (checkout battle rankings)]



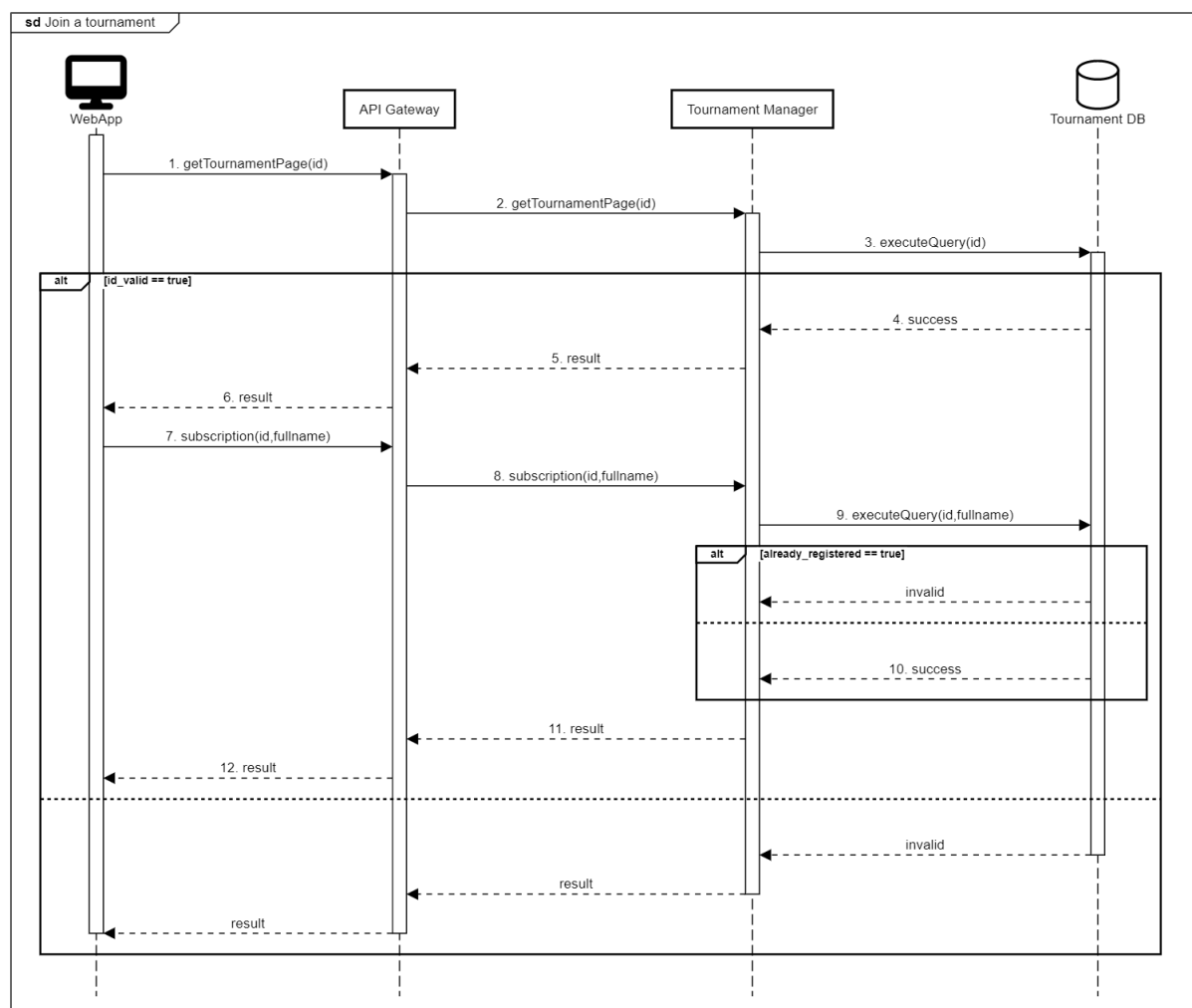
This sequence diagram represents the interaction that happens when a user wants to checkout battle rankings that equals reaching the specific battle page.

[Browse the platform (checkout user profiles)]



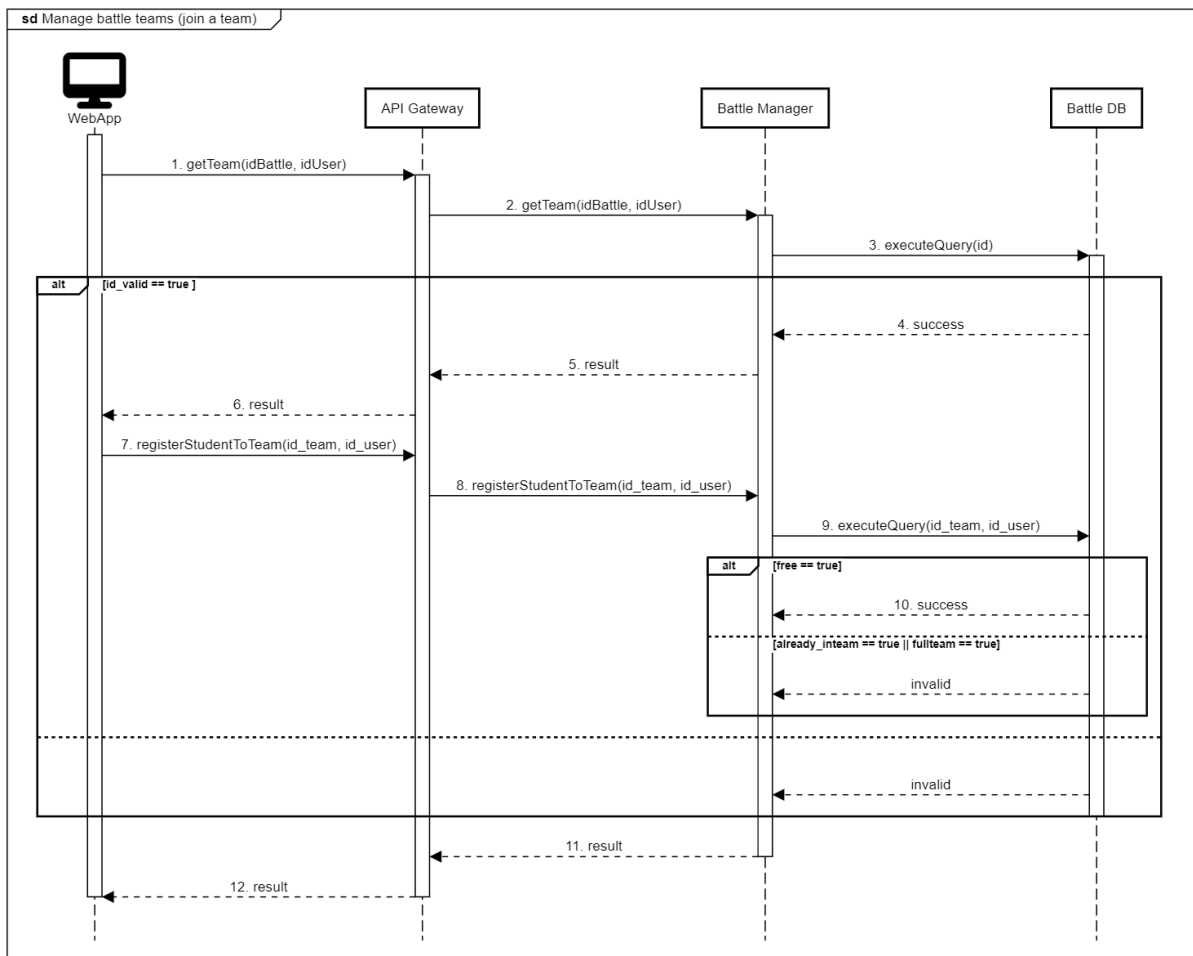
This sequence diagram represents the interaction that happens when a user wants to search for another user.

[Join a tournament]



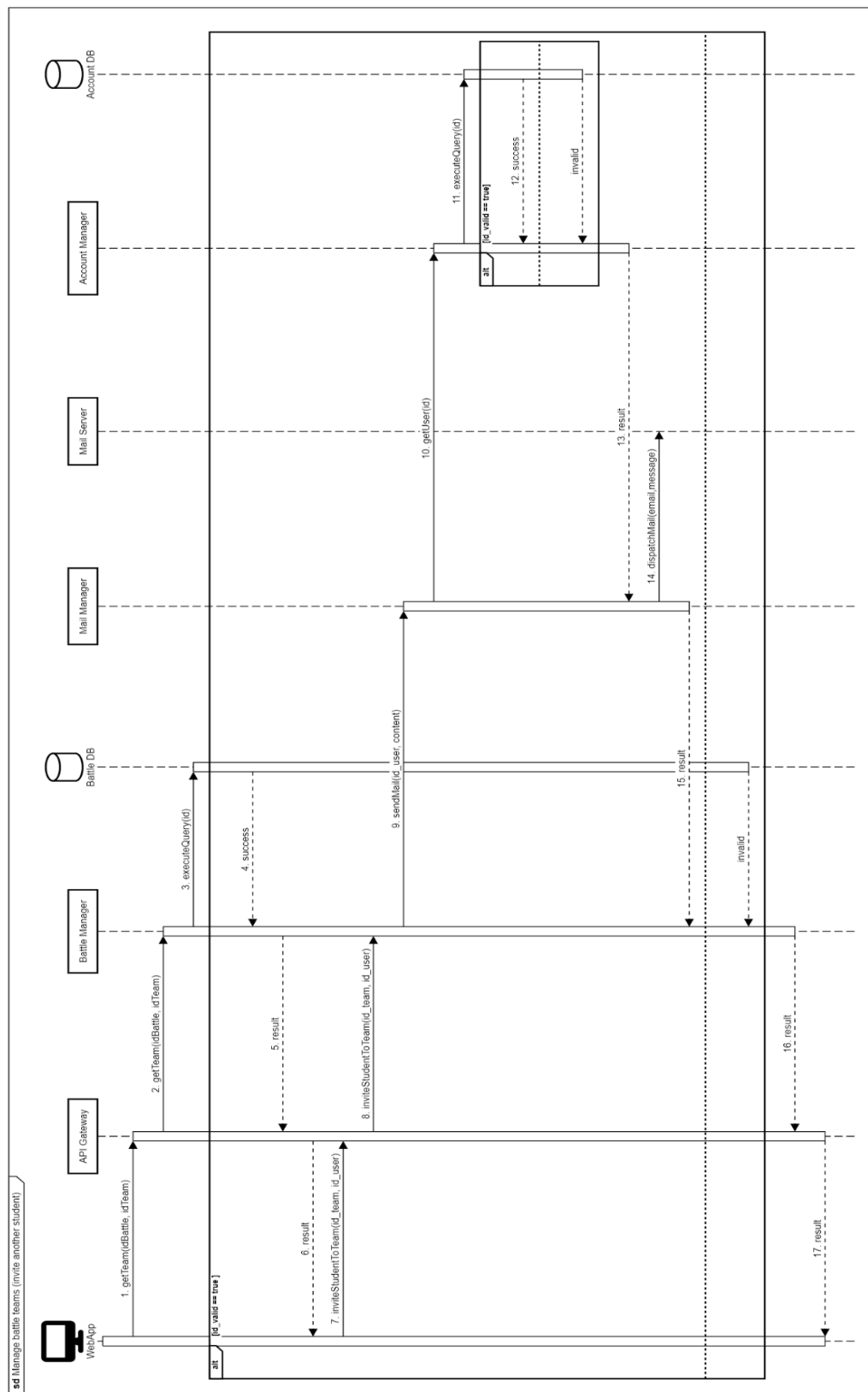
This sequence diagram represents the interaction that happens when a student wants to register to a specific tournament assuming that it's already logged.

[Manage battle teams (join a team)]



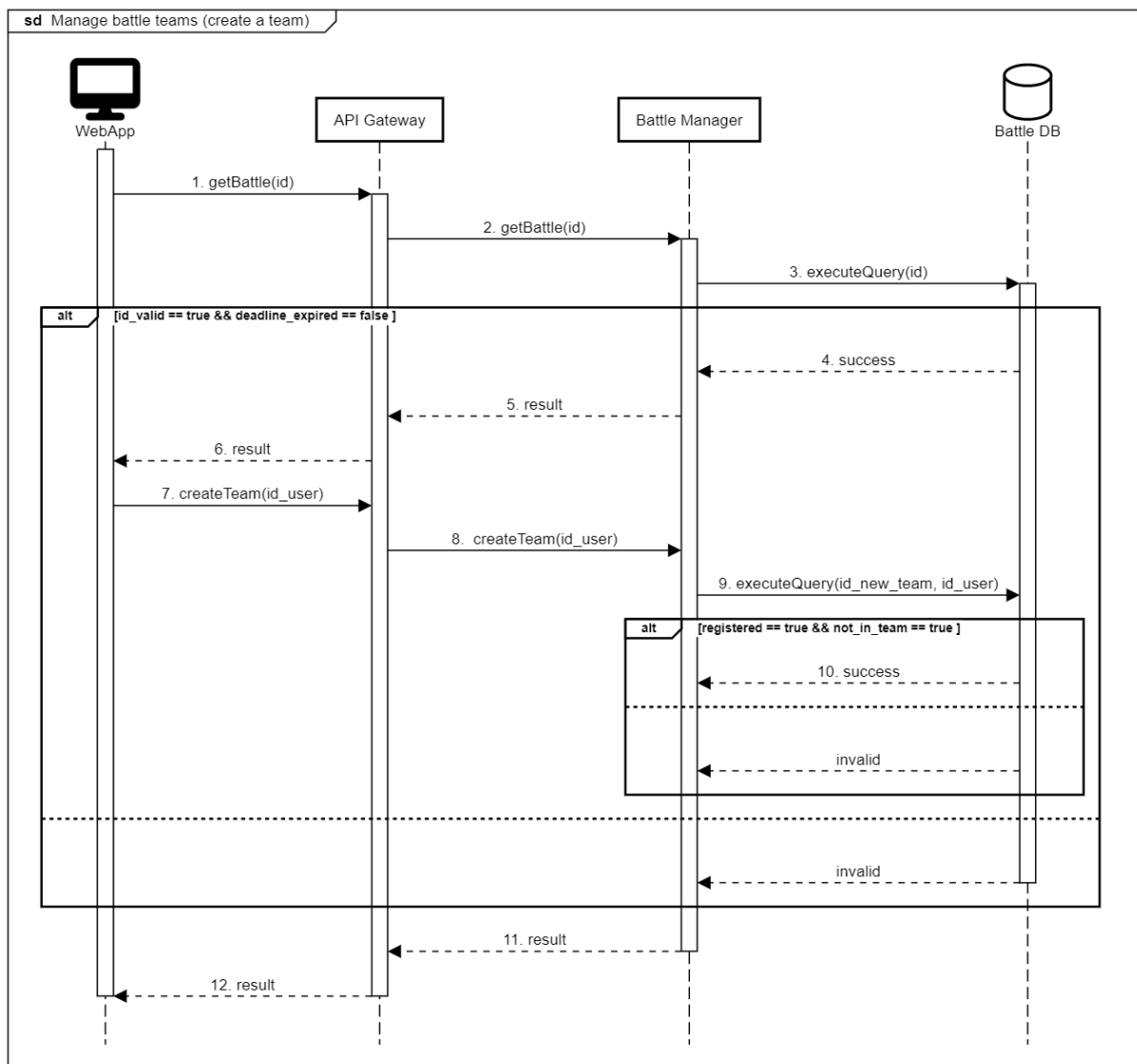
This sequence diagram represents the interaction that happens when a student wants to join a team.

[Manage battle teams (invite another student)]



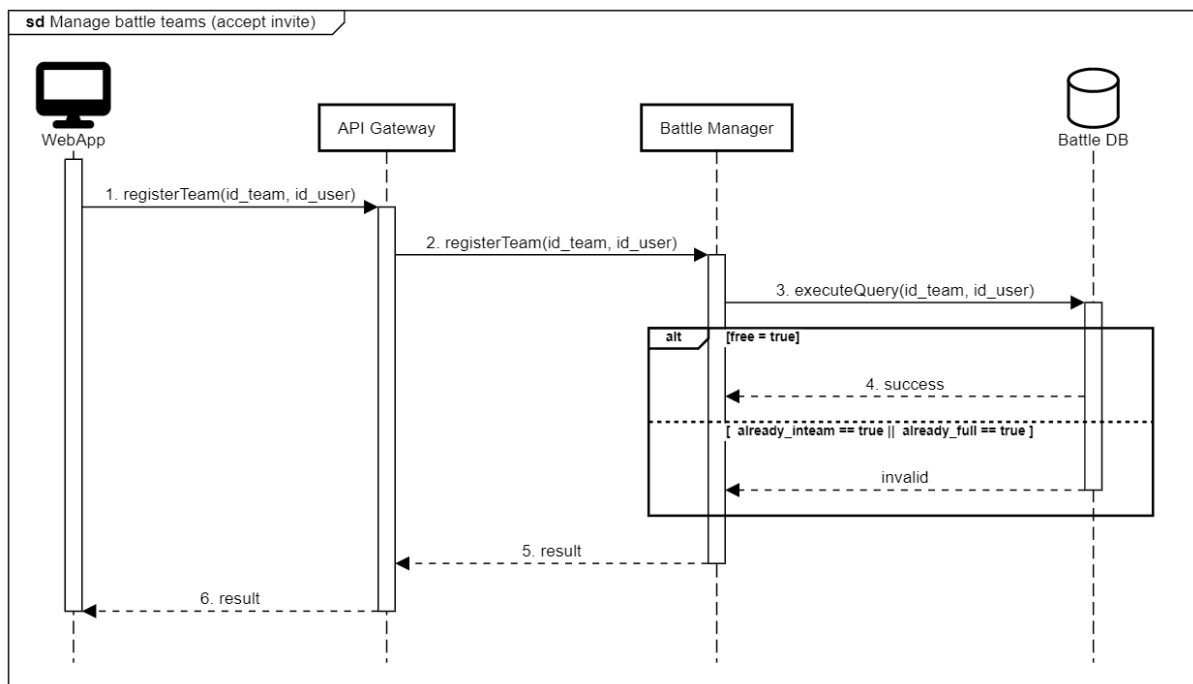
This sequence diagram represents the interaction that happens when a student wants to invite another student to its team.

[Manage battle teams (create a team)]



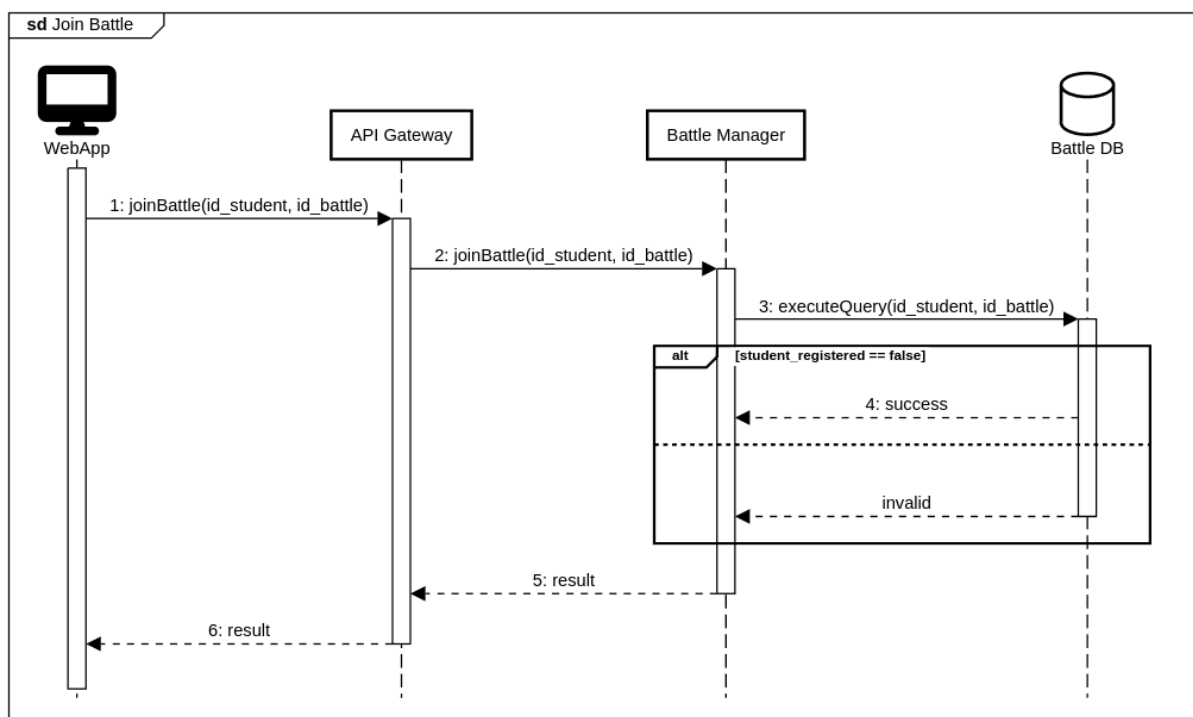
This sequence diagram represents the interaction that happens when a student wants to create a team.

[Manage battle teams (accept invite)]



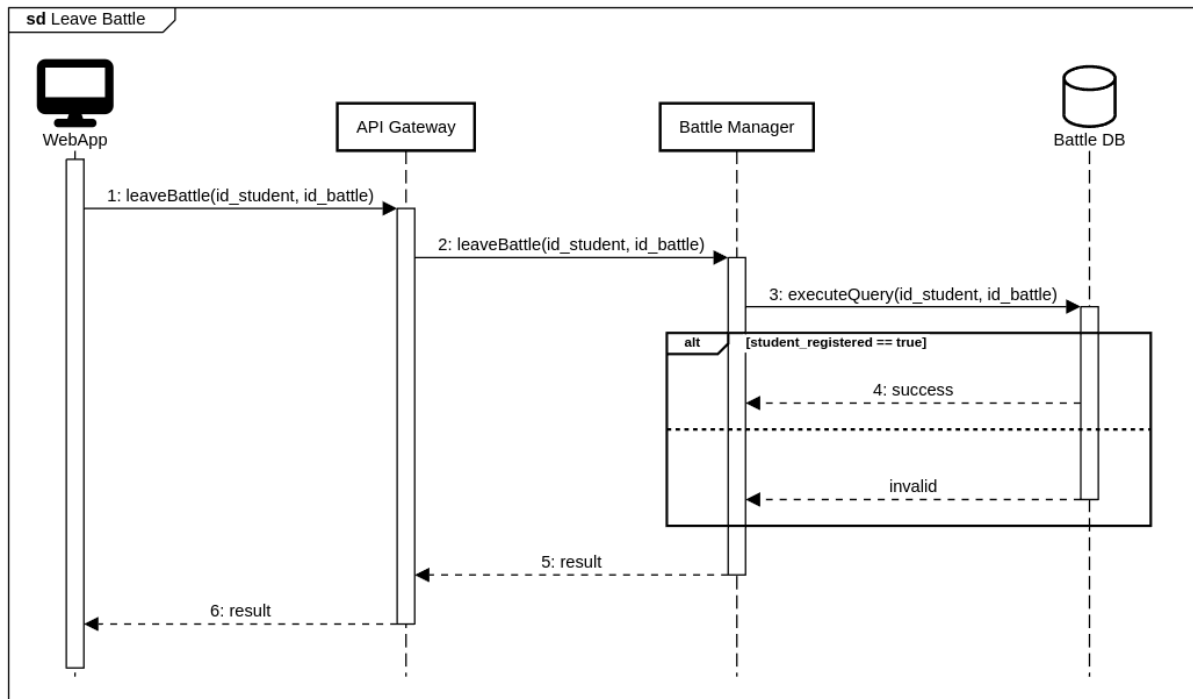
This sequence diagram represents the interaction that happens when a student wants to accept an invite received from another student.

[Join Battle]



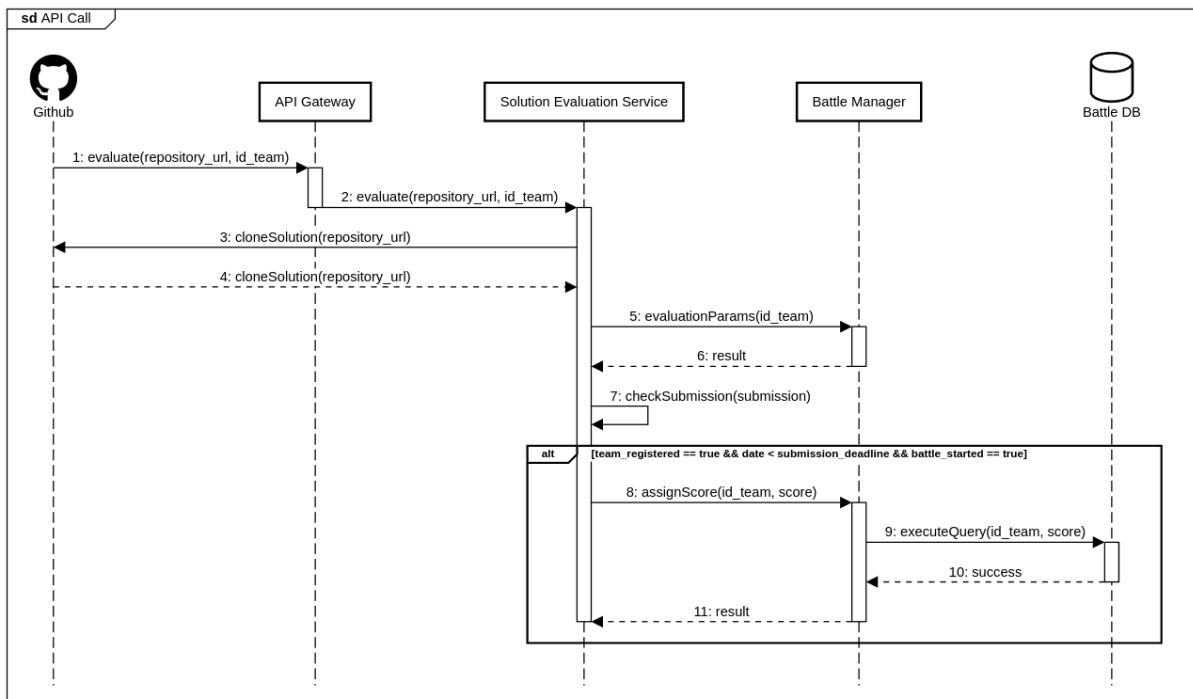
This sequence diagram represents the interaction that happens when a student wants to join a battle in the CKB application.

[Leave Battle]



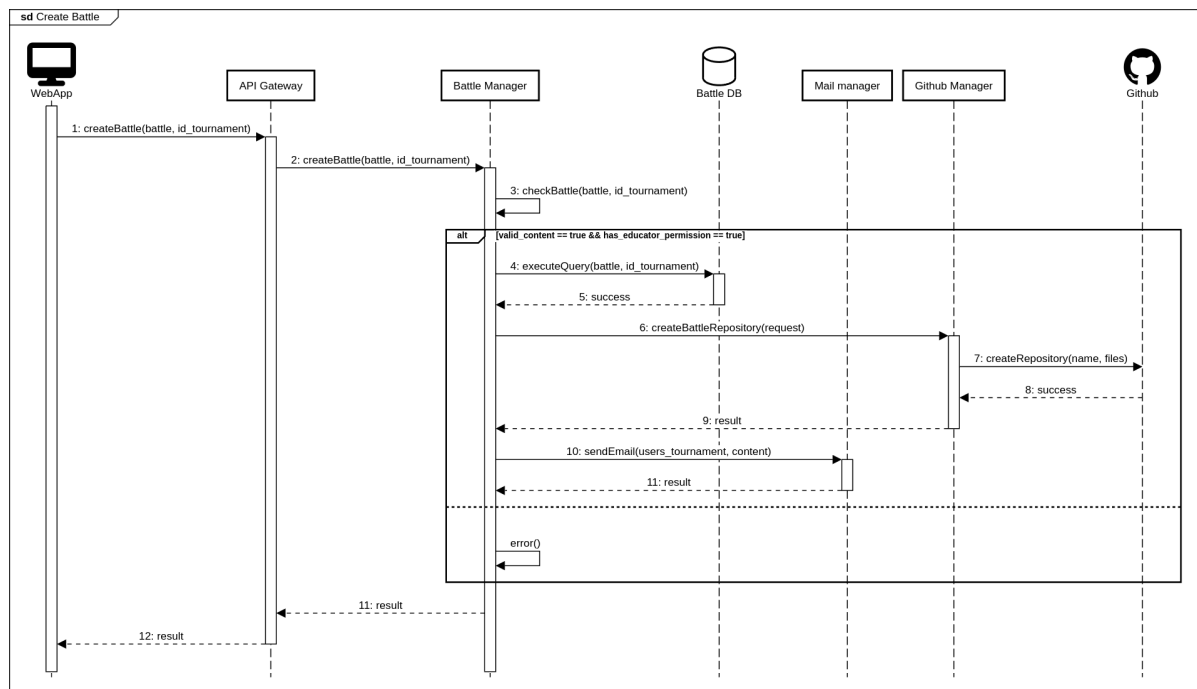
This sequence diagram represents the interaction that happens when a student wants to leave a battle in the CKB application.

[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]

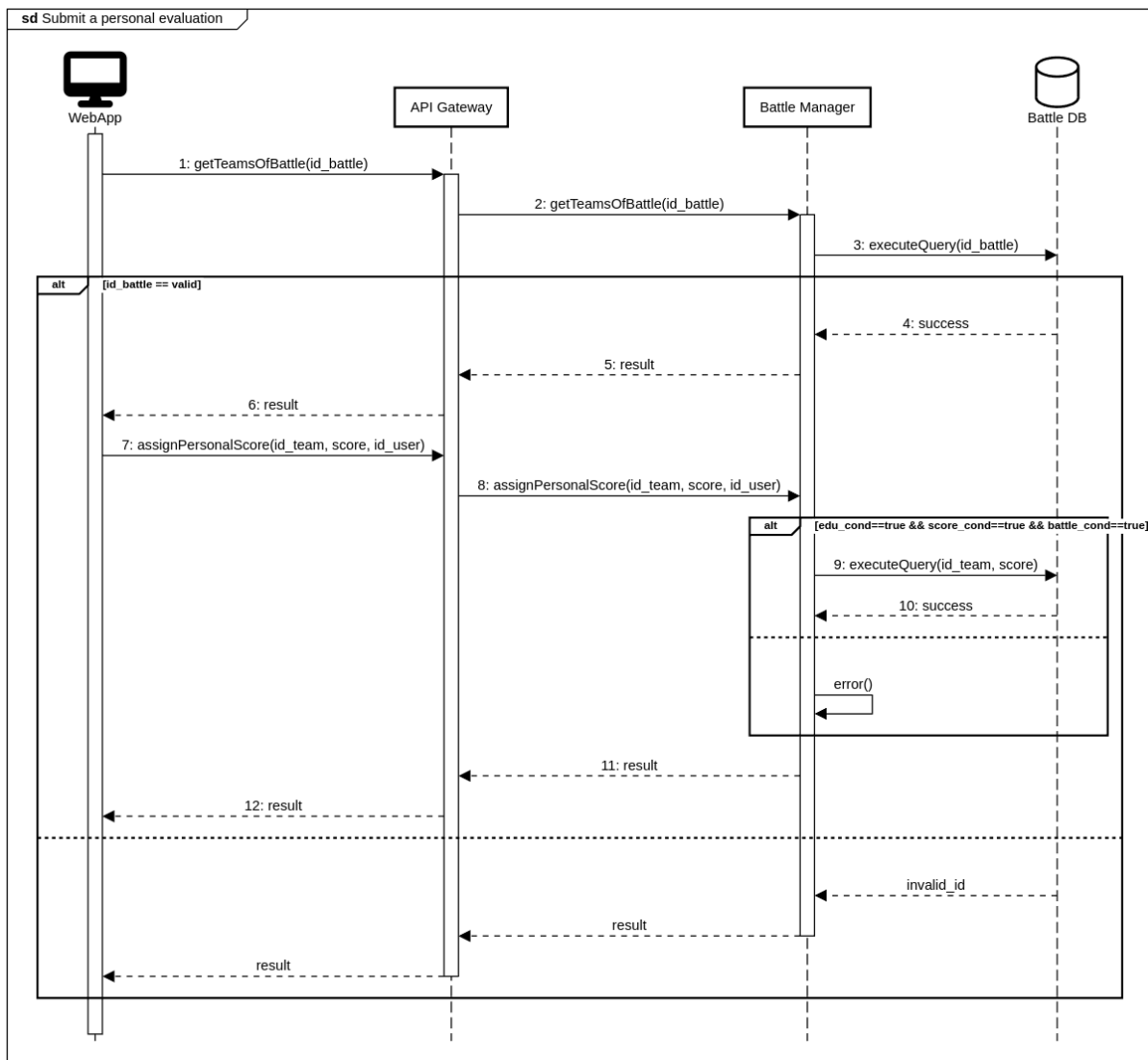


This sequence diagram represents the interaction that happens when an educator wants to create a battle in the CKB application.

There is an omission for simplicity: during the checks of the battle there is the control if the author is effectively an educator and if he/she has the permission to create a battle in the tournament.

The argument battle in createBattle is an object that contains a zipFile, an authorId, a minStudents, a maxStudents, a registrationDeadline, a submissionDeadline and a name

[Submit a personal evaluation]



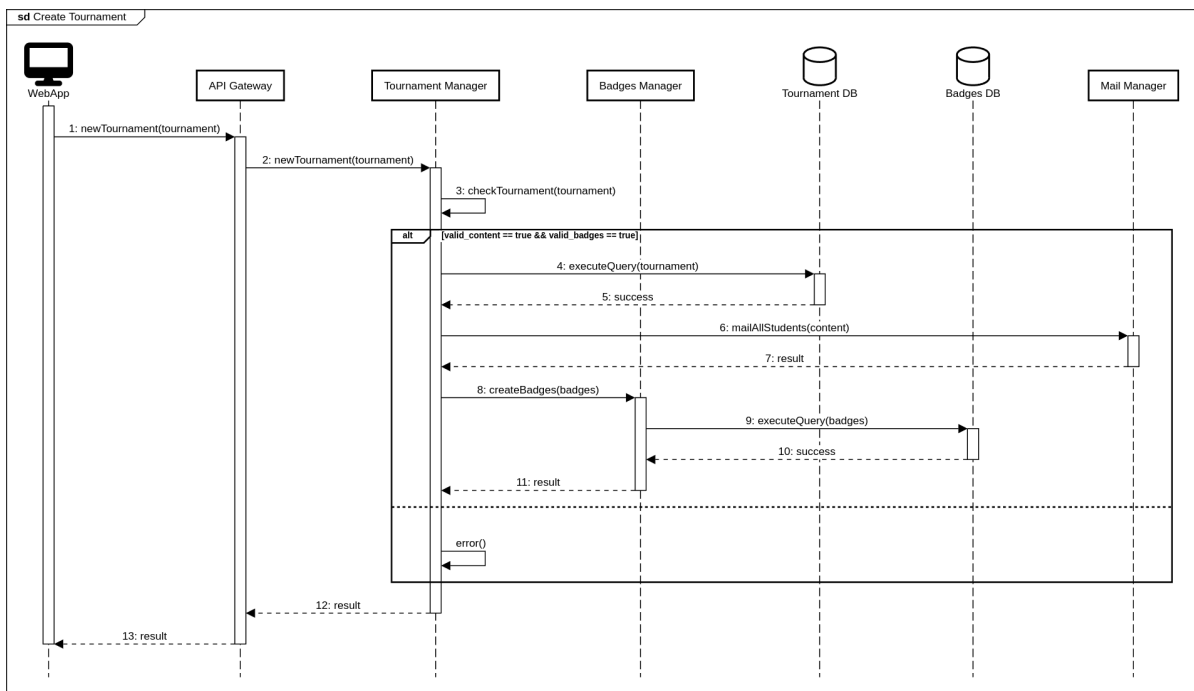
This sequence diagram represents the interaction that happens when an educator wants to assign a personal evaluation to a team in the CKB application.

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

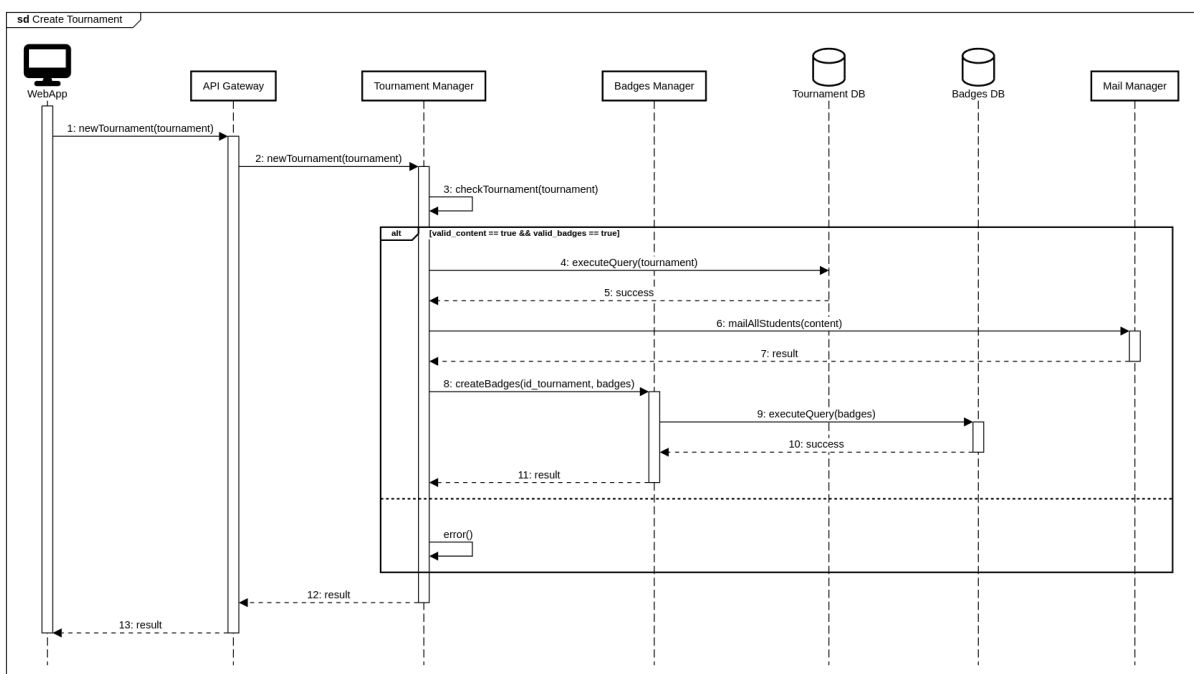
[Create Tournament]



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

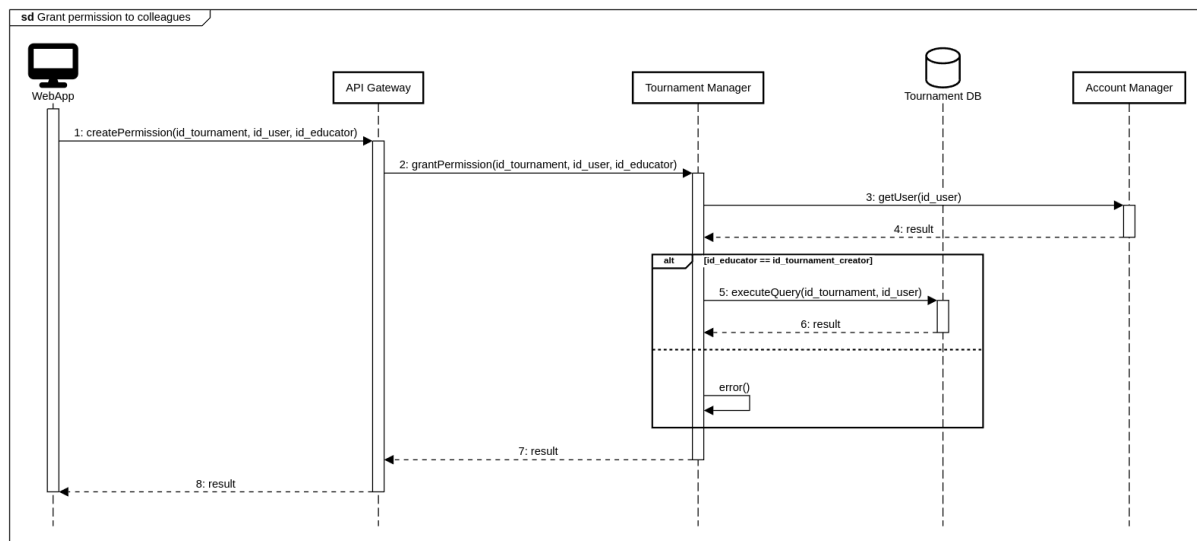
A tournament is an object that contains a creatorID, a name and a registrationDeadline.

[Close Tournament]



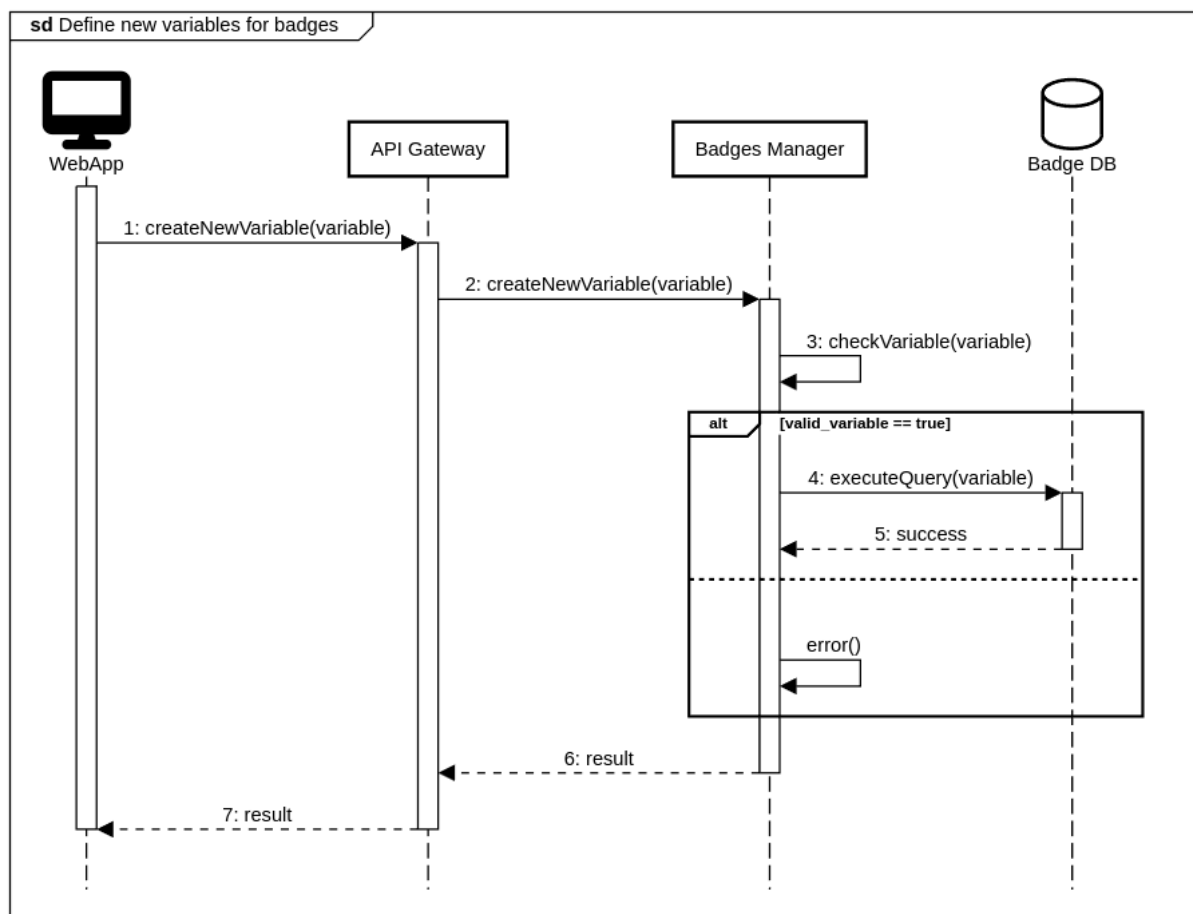
This sequence diagram represents the interaction that happens when an educator wants to close a tournament in the CKB application.

[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

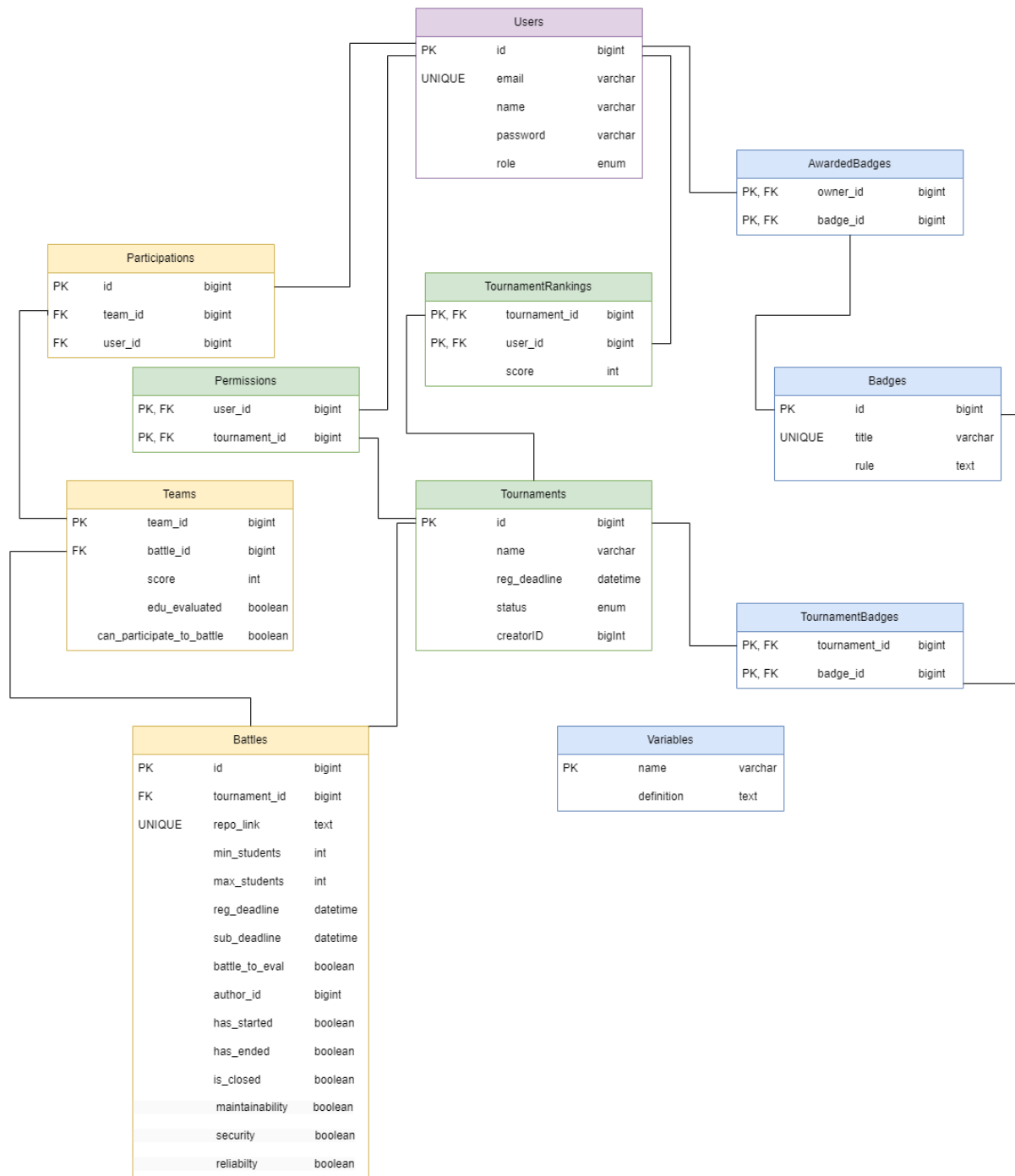
2.7.1 Database Structure

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



3. User Interface Design

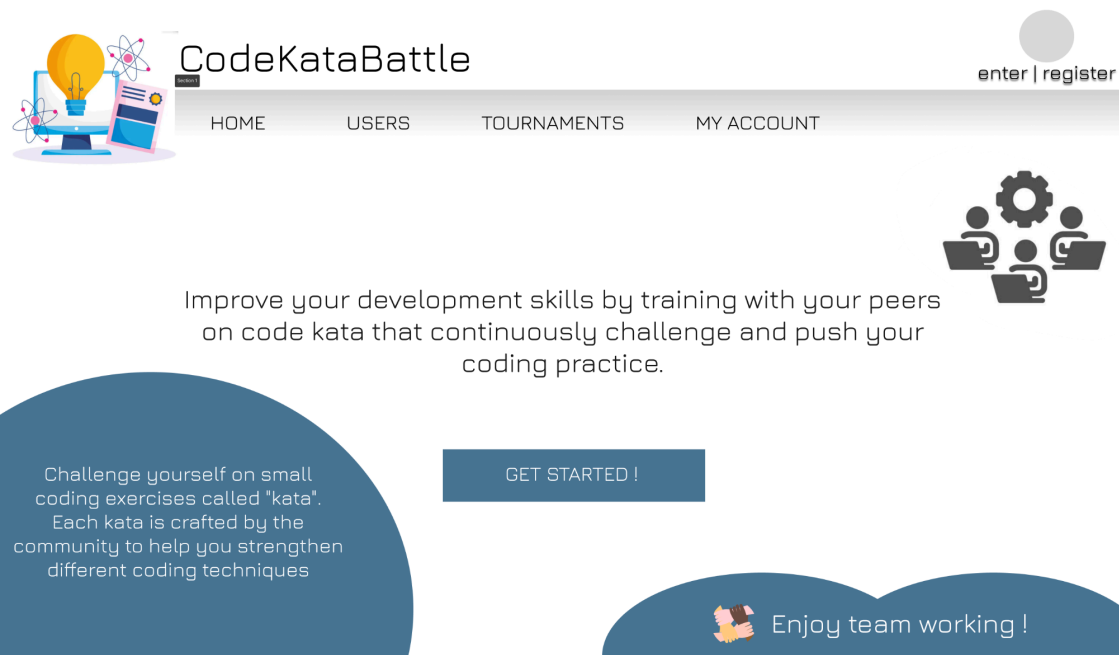


Fig.1 Home page

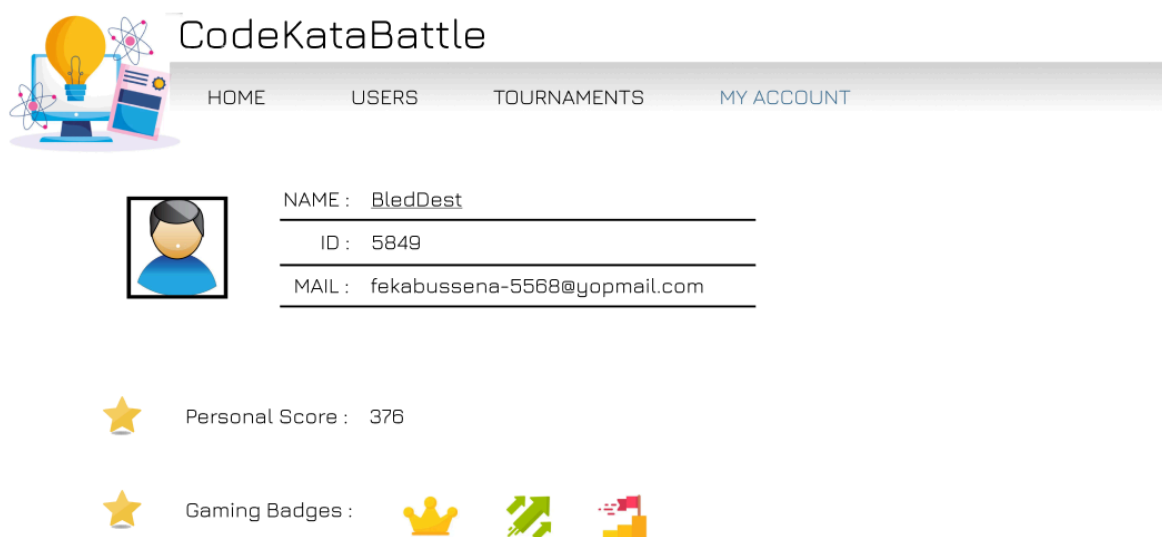


Fig.2 Personal page

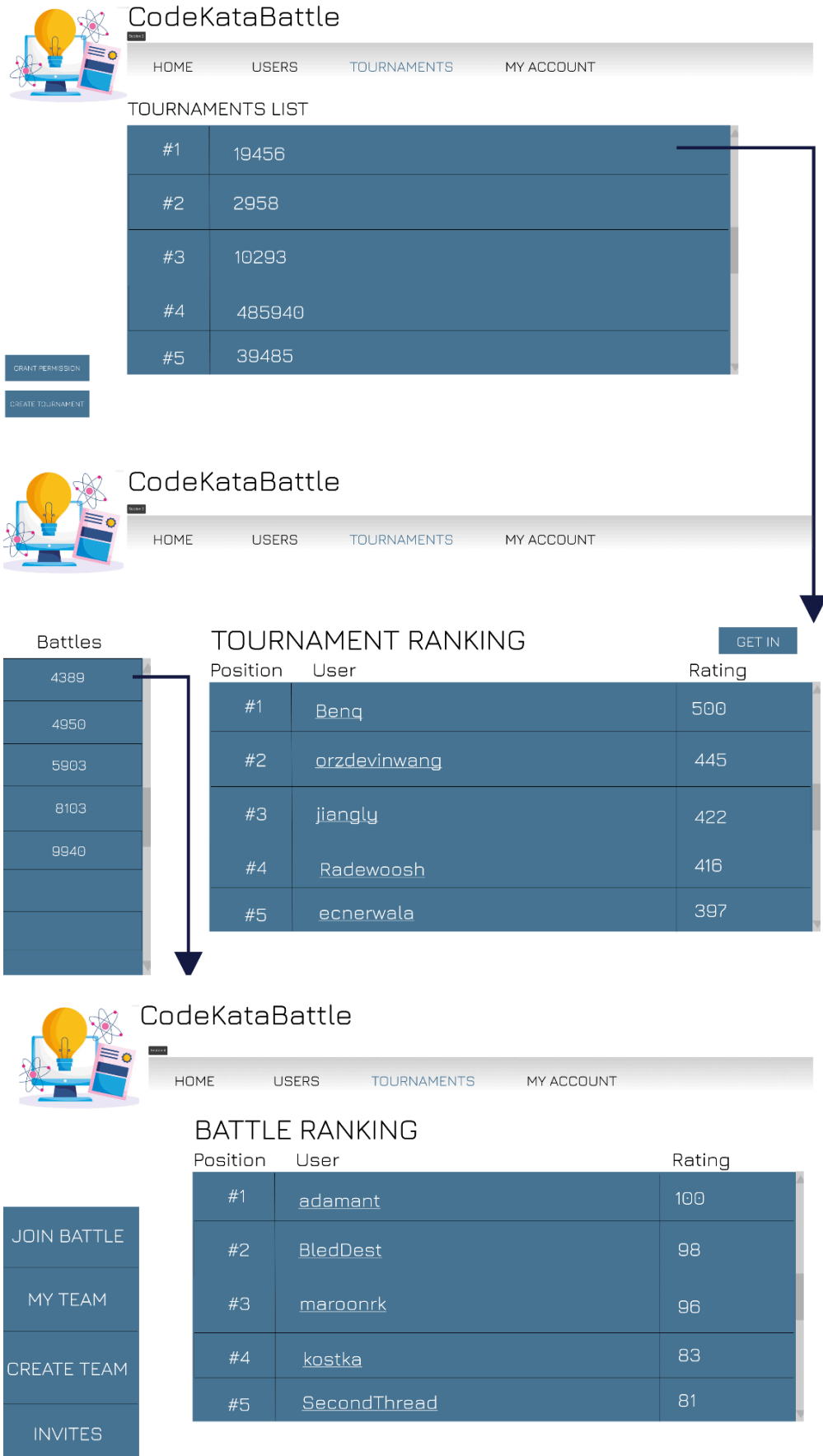



Fig.3 Tournament and Battle pages



CodeKataBattle

HOME USERS TOURNAMENTS MY ACCOUNT

insert participant for team :

upload CodeKata :


add personal evaluation : ☐ yes
☐ no

add deadline submission :

add registration submission :

CREATE

Fig.4 Battle creation page



CodeKataBattle

HOME USERS TOURNAMENTS MY ACCOUNT NOTIFICATIONS


add name badge :

add rule :

add registration submission :

CREATE

Fig.5 Tournament creation page




CodeKataBattle

HOME USERS TOURNAMENTS MY ACCOUNT

educator :

grant permission : ☐ yes ☐ no

Fig.6 Gran permission page



CodeKataBattle

HOME USERS TOURNAMENTS MY ACCOUNT

Participants

Paolo

Marta

Giovanni

team id :

link repository :

score :

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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• 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	<ul style="list-style-type: none">• Web App

	<ul style="list-style-type: none"> • Badges Service • DBMS
--	--

Requirements	<p>[R11] The system requires educators to define a way to assign scores to students submission in an automated way</p> <p>[R12] The system allows educators to decide whether they have to assign personal scores to students solutions during the consolidation phase</p>
Components	<ul style="list-style-type: none"> • Web App • Battle Manager • Solution Evaluation Service

Requirements	<p>[R13] The system allows educators to assign a personal score during the consolidation phase if they decided to allow it when creating the battle</p> <p>[R28] The system allows students and educators to see evolving rankings before a code battle has reached its submission deadline</p>
Components	<ul style="list-style-type: none"> • Web App • Solution Evaluation Service • DBMS

Requirements	<p>[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</p>
Components	<ul style="list-style-type: none"> • Web App • Battle Manager • Solution Evaluation Service • DBMS

Requirements	<p>[R15] The system notifies all students subscribed to the platform whenever a new tournament is created</p> <p>[R18] The system notifies students when the final tournament ranking become available</p>
Components	<ul style="list-style-type: none"> • Tournament Manager • Mail Service • DBMS

Requirements	<p>[R16] The system notifies students when a battle is created if they are registered to that battle's tournament</p>
Components	<ul style="list-style-type: none"> • Battle Manager • Tournament Manager

	<ul style="list-style-type: none"> • Mail Service • DBMS
--	--

Requirements	[R17] The system notifies students when the battle's final rankings become available
Components	<ul style="list-style-type: none"> • Solution Evaluation Service • Battle Manager • Mail Service • DBMS

Requirements	[R19] The system provides all students subscribed to a battle with that battle'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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • Web App • Battle Manager • Tournament Manager • DBMS

Requirements	[R23] The system allows students to invite other students to their group for a battle
Components	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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	<ul style="list-style-type: none"> • 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 (by mocking the implementation of the other services with the use of mock servers). After a service has been thoroughly tested, it can be tested together with other components, to ensure interactions among services work correctly, without simulation.

It's also important to implement a CI/CD pipeline to automate building and testing, have a fast feedback loop of the effect of the application developments and to ensure that everything is always tested and nothing breaks between the implementation and integration of different parts of the system, using a CI/CD pipeline also provides a reliable and consistent environment for testing, removing any platform dependencies and ensures that the code released into production is always up to standards with the required tests.

5.1 Services Integration plan

Since some services rely on the correct interaction with others, it's important to define what is going to need testing when integrating the different components. To do that we identify which services use by which other, working out the graph:



Following what is represented by this graph we understand that when a service has to be integrated in the code base, all the edges going in and out of its node, i.e., its interactions with other components, have to be tested. As an example, when integrating the tournament manager component, we'll need to test for the correct behavior of the interfaces it provides that are used by the battle manager component, moreover we'll need to test the APIs provided by the mail service and battle manager component which are used by the tournament manager.

If any of the services have not been implemented yet or are in any way not ready for integration, the test on that component can be held back until the interfaces that need testing are actually implemented.

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	1
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/>