POLITECNICO DI MILANO

SOFTWARE ENGINEERING II

CodeKataBattle

Design Document

Version 1.0

Authors

Biagio Marra
Raffaele Russo

Contents

1 Introduction

1.1 Purpose

The fundamental goal of the "CodeKataBattle" project is to facilitate the enhancement of students' programming skills through training with hands-on exercises known as CodeKata. This initiative not only aims to provide students with an interactive and engaging learning environment, but also offers teachers the opportunity to organize special tournaments with battles between teams of students within them. Through these tournaments, teams of students have the opportunity to test their programming skills in challenging competitions. In summary, the project aims to create a dynamic and educational environment where students can hone their technical skills, participate in engaging challenges, and grow as competent programmers and problem solvers.

1.2 Scope

CodeKataBattle is a system designed to meet the growing demand for software developers by offering a competition arena for programmers. The intuitive interface allows students to register for tournaments by the closing date for registration. Within these tournaments, students can participate in battles, creating teams with other participants before the registration deadline. At the end of the registration phase, the system sends each participant a link to the main repository on GitHub, giving them access to CodeKata to start their work.
The system automatically updates the team's score with each push made in the repository, evaluating aspects defined and customised by the educators through third-party tools. During the course of the battle, the system provides a real-time ranking to the participants and, at the end of the battle, communicates the final ranking to all participants.
The educators have a dedicated interface for managing tournaments, creating and ending battles, and inviting other educators to create new battles. This interface allows relevant information to be specified and the corresponding CodeKata to be uploaded.
CodeKataBattle is a system designed to meet the growing demand for skilled software developers, and we chose to implement it following a microservice architecture. This architectural choice offers many advantages, including modularity, scalability, and ease of maintenance. Thanks to the microservices approach, the system can be broken down into independent and autonomous components, each handling specific functionality. The architectural structure outlined provides a solid foundation for building a system that is highly scalable and easily adaptable to future needs. The ability to replicate microservice instances to handle increasing numbers of users is known as "horizontal scalability." This approach allows the workload to be distributed across multiple service instances, improving performance and the ability to manage the system linearly with respect to traffic growth. In addition, the modularity and independence of microservices allow the easy addition of new functionality to the system. Creating new microservices independently allows the capabilities of the system to be extended without affecting existing functionality. This flexibility in design facilitates the integration of new services efficiently and without disruption to the overall system.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Fork the GitHub repository**: This action means creating one's own independent copy of a project hosted on GitHub. This operation allows a user to work on a version of the project without directly affecting the original repository.
- **GitHub Actions**: GitHub Actions is an automation service built directly into GitHub that allows developers to automate software workflow. It allows processes to be defined, customized and automated through a series of actions performed in response to specific events within a GitHub repository.
- **Test first-approach**: The Test-First approach has been known as Test-First Development or TDD. It is an integrated quality method used in the Extreme Programming methodology in which developers write unit tests before writing production code.
- **Circuit Breaker**: It is a mechanism that protects a system from the consequences of prolonged or frequent failures of a microservice by preventing failures from propagating upstream of the system.

1.3.2 Acronyms

- **CKB** : CodeKataBattle
- **CK** : CodeKata
- **RASD** : Requirement Analysis and Specification Document.
- **API** - Application Programming Interface
- **DB** - Database

1.3.3 Abbreviations

- [Gn] - the n-th goal of the system
- [WPn] - the n-th world phenomena
- [SPn] - the n-th shared phenomena
- [UCn] - the n-th use case
- [Rn] - the n-th functional requirement

1.4 Revision history

- Version 1.0

1.5 Reference Documents

This document is based on:

- The specification of the RASD and DD assignment of the Software Engineering II course, held by professor Matteo Rossi, Elisabetta Di Nitto and Matteo Camilli at the Politecnico di Milano, A.Y 2022/2023;
- Slides of Software Engineering 2 course on WeBeep;

1.6 Document Structure

Mainly the current document is divided in 7 chapters, which are:

1. **Introduction:** The first chapter includes the introduction in which is explained the purpose of the document, then, a brief recall of the concepts introduced in the RASD is given. Finally, important information for the reader is given, i.e. definitions, acronyms, synonyms and the set of documents referenced.
2. **Architectural Design:** it includes a detailed description of the architecture of the system, including the high level view of the elements, the software components of CLup, a description through runtime diagrams of various functionalities of the system and, finally, an in-depth explanation of the architectural pattern used.
3. **User Interface Design:** are provided the mockups of the application user interfaces, with the links between them to help in understanding the flow between them.
4. **Requirements Traceability:** it describes the connections between the requirements defined in the RASD and the components described in the first chapter. This is used as proof that the design decisions have been taken with respect to the requirements, and therefore that the designed system can fulfill the goals.
5. **Implementation,Integration and Test Plan:** it describes the process of implementation, integration and testing to which developers have to stick in order to produce the correct system in a correct way.
6. **Effort spent:** it shows the time spent to realize this document, divided for each section;
7. **References:** it contains the references to any documents and to the Software used in this document.

2 Architectural design

2.1 Overview

2.1.1 High level view

The CodeKataBattle system is a distributed architecture that leverages the fundamental principles of microservice architecture. This choice was made in order to capitalize on the many advantages offered by this architecture, including scalability, facilitated maintainability, rapid development, and ease of integration. Figure 2.1 provides a high-level representation of the system.

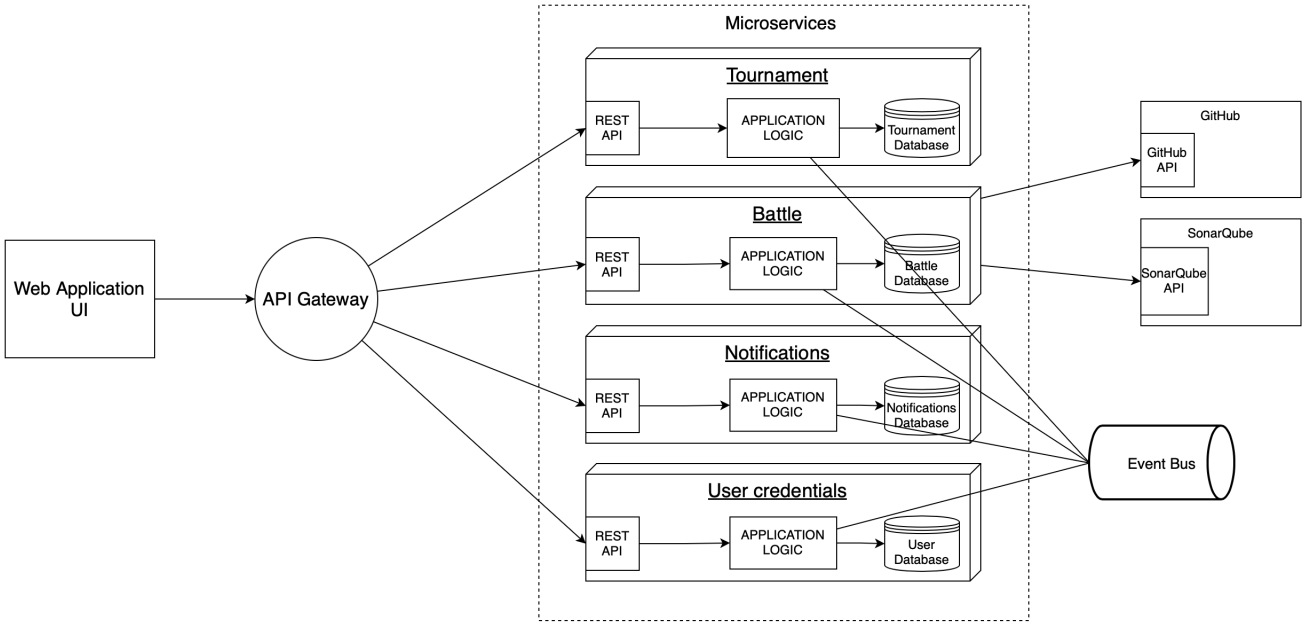

Figure 2.1: High Level View Architecture

Starting on the left side of the figure, the Web Application UI forms the interface through which users interact with the system. Each user interaction with the interface involves generating requests HTTPS to the API Gateway. The latter, acting as a central control and management point, simplifies the interaction between clients and the various backend services present in the microservice architecture. The Gateway API provides essential functionality to improve the efficiency, security, and overall management of the distributed system.
Next, we illustrate the microservices that form the core of the system, each microservice consists of the REST API with which it communicates with the outside world, the logic inside and its own database are store information, the chosen microservices used are Tournament, Battles, Notifications and User Credentials.
The microservice related to Tournament handles the complete management of tournaments, encompassing activities such as creation, registrations, and leaderboard management.

The Battles-related microservice handles every aspect of battles, from the creation of battles and repositories on GitHub to the final evaluations of the system and the SonarQube service.
The "Notifications" microservice takes responsibility for notifying the user through the application, as well as managing the storage of all notifications. This feature allows users to fully retrieve notifications when they use the application.
Finally, the User Credentials microservice maintains all information about registered users on the platform and manages the authentication process during login.
Finally we have the Event Bus, which acts as an intermediary for communication between microservices, allowing them to publish and receive messages without the need for direct, synchronous communication. Each microservice can interact with the bus-event to perform event-publishing operations, thereby notifying other services affected by certain state changes or actions. This asynchronous mode of communication has several advantages, including decoupling, scalability, robustness, and agility. This architectural structure provides a solid foundation for building a distributed system that fits the scalability, maintenance, rapid development, and integration needs of the CodeKataBattle system.

2.2 Component view

This section will report the component views for each microservice:

2.2.1 User Credentials Microservice

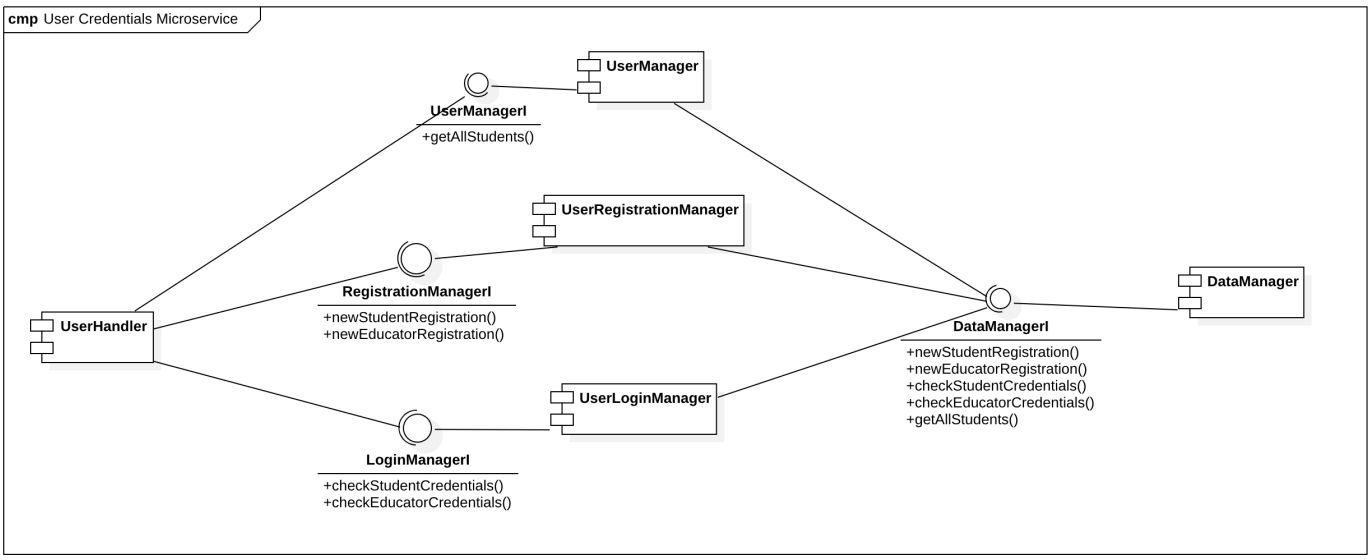

Figure 2.2: User Credentials Microservice Component View

The components present are as follows:

- **UserHandler**: is the key component within the microservice dedicated to receiving HTTP requests and external events. Its primary function is to use the interfaces of the other components to execute incoming requests, serving as an essential connector between the external interaction and the internal architecture of the microservice.
- **UserManager**: constitutes the component dedicated to managing users registered in the system. It provides an accessible interface for the Handler to request services and interacts with the "DataManager" to obtain the necessary information.
- **UserRegistrationManager**: represents the component in charge of the user registration operation in the system. Taking advantage of the Handler's interface, this module requests the necessary services from the "DataManager." Its main responsibility is to orchestrate the user registration process, acting as a bridge between the Handler interaction and the data persistence operations managed by the "DataManager."
- **UserLoginManager**: is the component specifically designed to manage the operation of user access to the system. Through the use of the interface provided by the Handler, this module requests the necessary services from the "DataManager." Its key function is to orchestrate the user authentication process, acting as an intermediary between the Handler interaction and the data management operations handled by the "DataManager."

- **DataManager**: is the core of the microservice, incumbent on providing methods for data persistence. Its primary responsibility is to facilitate the reliable storage of data, as well as to retrieve information from the database when requested by other components. Operating as the key interface for persistence operations.

2.2.2 Tournament Microservice

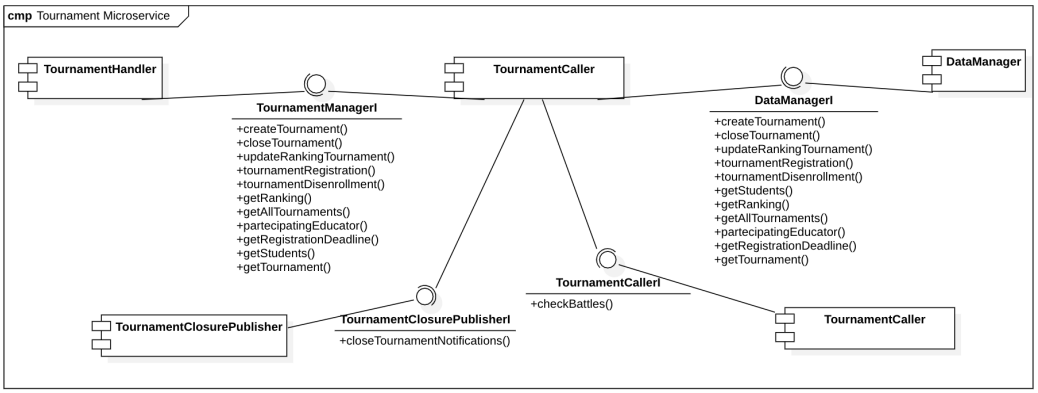

Figure 2.3: Tournament Microservice Component View

The components present are as follows:

- **TournamentHandler**: is the key component within the microservice dedicated to receiving HTTP requests and external events. Its primary function is to use the interfaces of the other components to execute incoming requests, serving as an essential connector between the external interaction and the internal architecture of the microservice. It is also responsible for receiving events from the bus dedicated to publishing the final scores of a battle.

- **TournamentManager**: is the component in charge of all tournament-related operations within the system. Interacting with the Handler through its interface, the "TournamentManager" requests the necessary services from the "DataManager." Its central function is to orchestrate the various tournament-related activities, acting as a bridge between the Handler's requests and the data persistence operations managed by the "DataManager."

- **DataManager**: is the core of the microservice, incumbent on providing methods for data persistence. Its primary responsibility is to facilitate the reliable storage of data, as well as to retrieve information from the database when requested by other components. Operating as the key interface for persistence operations.

- **TournamentClosurePublisher**: assumes the role of the person responsible for posting events related to the closing of a tournament within the bus-event.Which event will be forwarded to the notifications microservice.

- **TournamentCaller**: component responsible for asking the battles microservice if the battles within the tournament have ended.

2.2.3 Notifications Microservice

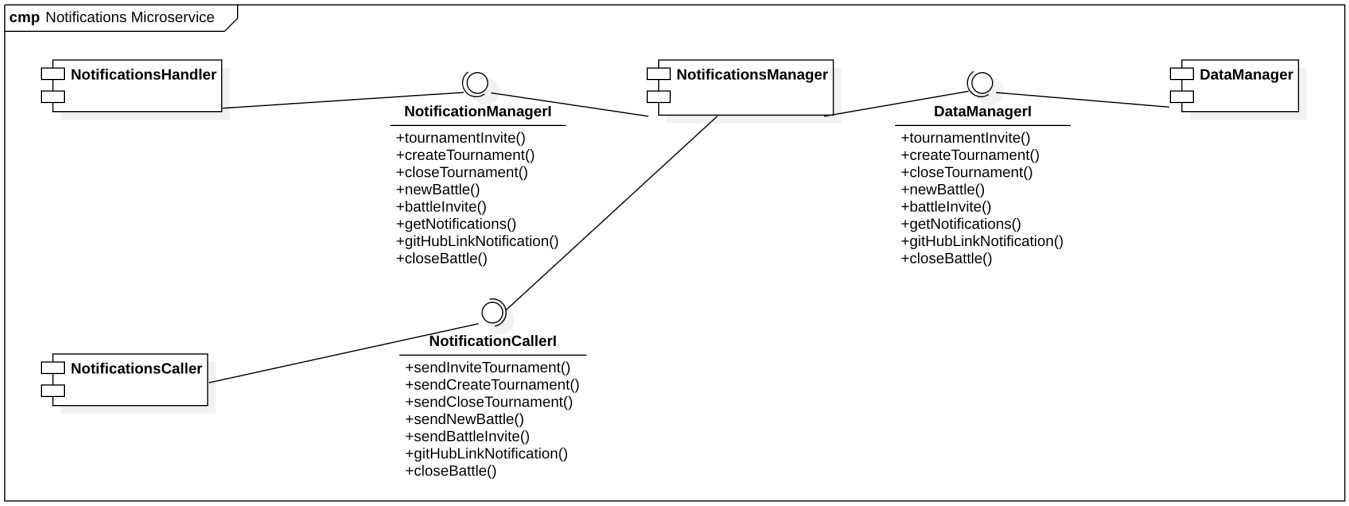

Figure 2.4: Notifications Microservice Component View

The components present are as follows:

- **NotificationsHandler**: represents the component designated to receive notification requests from other services or the application user. Its primary responsibility is to handle notification requests, serving as the initial interface between the requesting services and the microservice notification system. Once the requests are received, the "NotificationsHandler" forwards them to the relevant "NotificationsManager" for the actual handling and dissemination of notifications. It is also responsible for receiving events from the dedicated publication buses that need to be notified to the user.

- **NotificationsManager**: is the crucial component in the context of managing notifications within the system. Its primary functionality begins with the receipt of notification requests from the Handler. Once received, the "NotificationsManager" proceeds with the persistence of the information associated with the notifications in the database. And finally it entrusts another component with forwarding the notifications to the recipients.

- **DataManager**: is the core of the microservice, incumbent on providing methods for data persistence. Its primary responsibility is to facilitate the reliable storage of data, as well as to retrieve information from the database when requested by other components. Operating as the key interface for persistence operations.

- **NotificationsCaller**: provides an interface to assume the role of a dedicated component for forwarding notifications to all relevant recipients.

2.2.4 BattleMicroservice

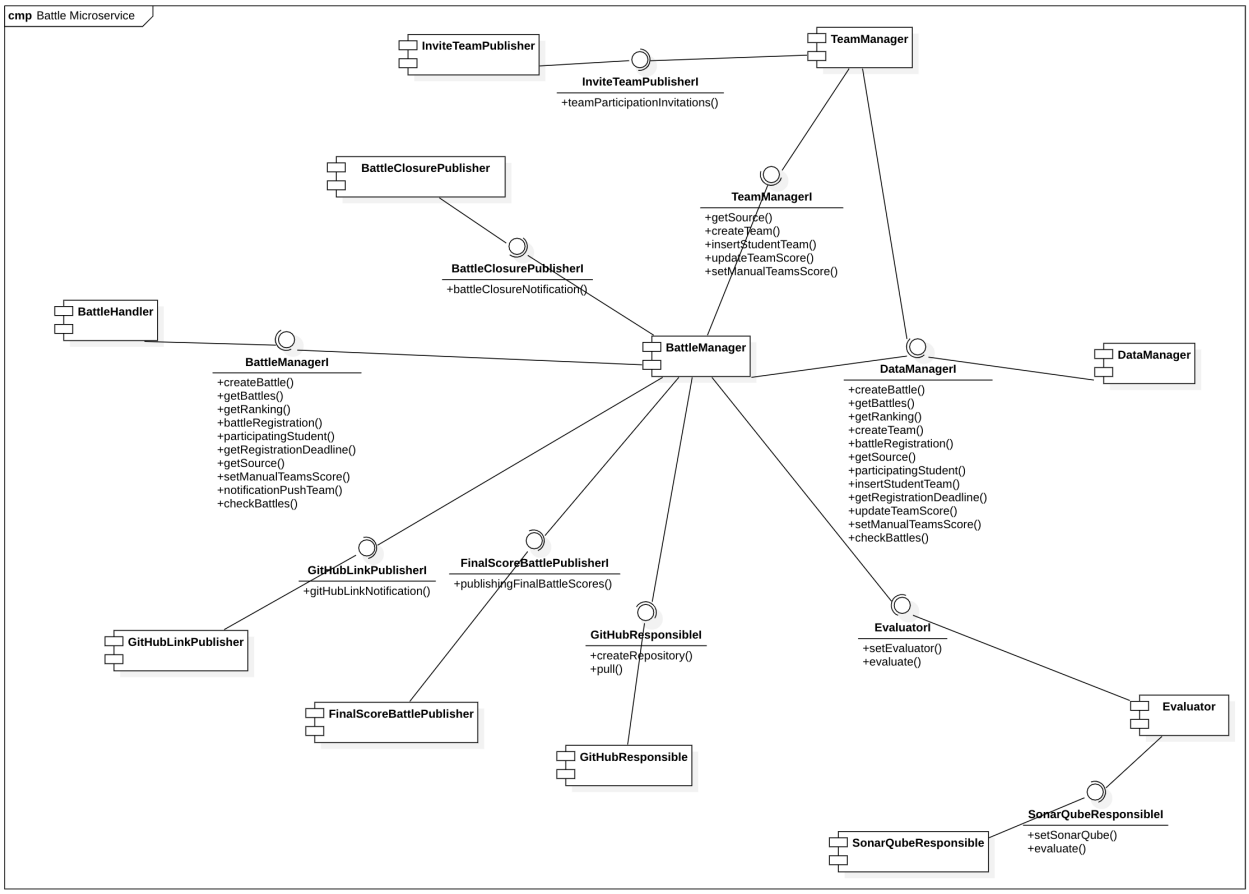

Figure 2.5: Battle Microservice Component View

The components present are as follows:

- **BattleHandler**: is the key component within the microservice dedicated to receiving HTTP requests and external events. Its primary function is to use the interfaces of the other components to execute incoming requests, serving as an essential connector between the external interaction and the internal architecture of the microservice.
- **BattleManager**: assumes the primary role in battle-related operations within the system. Its primary responsibility is to orchestrate and coordinate all activities related to battle management. Using the interface provided by other components.
- **TeamManager**: is the component that deals with the management of teams in the context of battle.
- **InviteTeamPublisher**: assumes responsibility for posting on bus events the request for a team invitation.
- **GitHubResponsible**: takes on the specific role of interfacing with the GitHub service and performing relevant actions on it. Its primary responsibility is to orchestrate formal interactions with the GitHub service.

- **Evaluator**: constitutes the component in charge of evaluating each source provided by teams during battles.
- **SonarQubeResponsible**: Has the specific role of interfacing with SonarQube service and requesting evaluation from it on the source.
- **BattleClosurePublisher**: takes responsibility for publishing on the bus events that a battle has been closed.
- **GitHubLinkPublisher**: performs the responsibility of publishing on the bus events the information needed by the microservice notifications to send the GitHub repository link.
- **FinalScoreBattlePublisher**: plays the role of publishing the final scores of a battle on the bus-event platform.

2.3 Deployment view

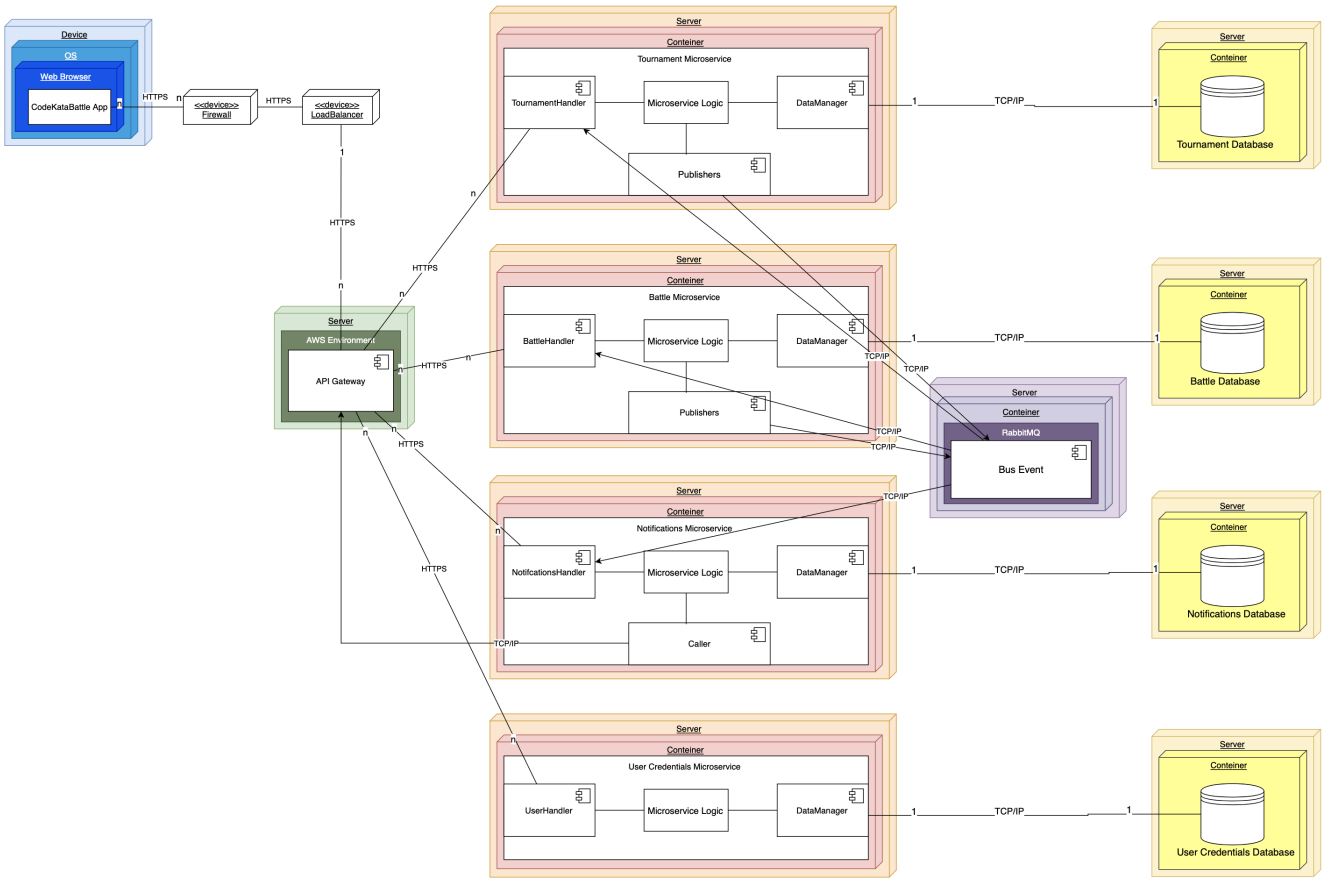

Figure 2.6: Deployment View

Further details about the elements in the graph are provided in the following, specifically on the links are the communication protocol and cardinality:

- **Device**: The device in question has the ability to access a web browser in order to use the CodeKataBattle web application.
- **Firewall**: The firewall is placed between the client devices and the API Gateway, acting as a protective barrier. Implemented security policies include filtering traffic according to specific rules and monitoring suspicious activity.
- **Load Balancer**: A load balancer is a device responsible for performing the load balancing operation. In the context of our system, its use is aimed at equitable load distribution among the different available API Gateway instances.
- **API Gateway**: API Gateway is a fully managed service that simplifies API management by providing a centralized entry point for API design, publishing, maintenance and protection. This service is hosted by Amazon's AWS servers and acts as an interface between users and backend services. In our case, multiple instances of the API Gateway were introduced to avoid the single point of failure.

- **Microservices server**: Each microservice is hosted within a container on dedicated servers. Scalability is managed by replicating microservice instances as needed. Microservices communicate with each other via bus event and receive requests from the Gateway API. Each microservice has its own database.
- **Bus Event**: The event bus is implemented on a container developed by RabbitMQ, receiving events from different microservices and delivering events to microservices subscribed to those buses.
- **Database server**: The databases are implemented within containers on dedicated servers. Each database is closely associated with a specific microservice and establishes communication with the latter through TCP/IP-based channels. This structure enables a distributed architecture in which each microservice can efficiently interact with its related database.

2.4 Runtime views

In this chapter are presented all the runtime views associated with the use cases described in the RASD relative to CodeKataBattle. Runtime views show the interactions between the various components to carry out the functionalities offered by CodeKataBattle.

[UC1] - Registration

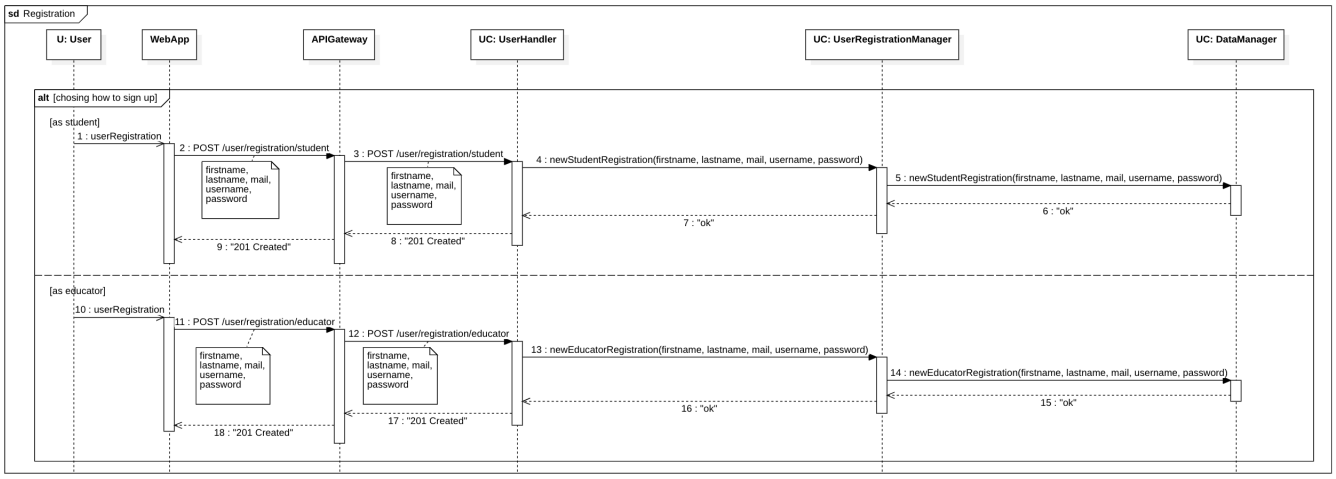

Figure 2.7: Registration Runtime View

The sequential representation illustrates the registration flow within the system. The user, via the WebApp, fills out the registration form with the required data, such as firstname, lastname, mail, username and password. The choice to register as educator or student is made by selecting the respective buttons.
After making the selection, the WebApp sends the registration request to the APIGateway via an HTTP POST request, which acts as an intermediary. The API gateway forwards this request to the 'User Credential' microservice, more precisely to the "UserHandler" component, using the REST API provided by the microservice. The 'UserHandler' receives and handles the request. Next, the "UserHandler" routes this request to the "UserRegistrationManager", which is responsible for managing the registration process. Once the data have been processed, they are sent to the "DataManager", which is responsible for storing them in the database. If the execution was successful it is responded with a confirmation.

[UC2] - Login

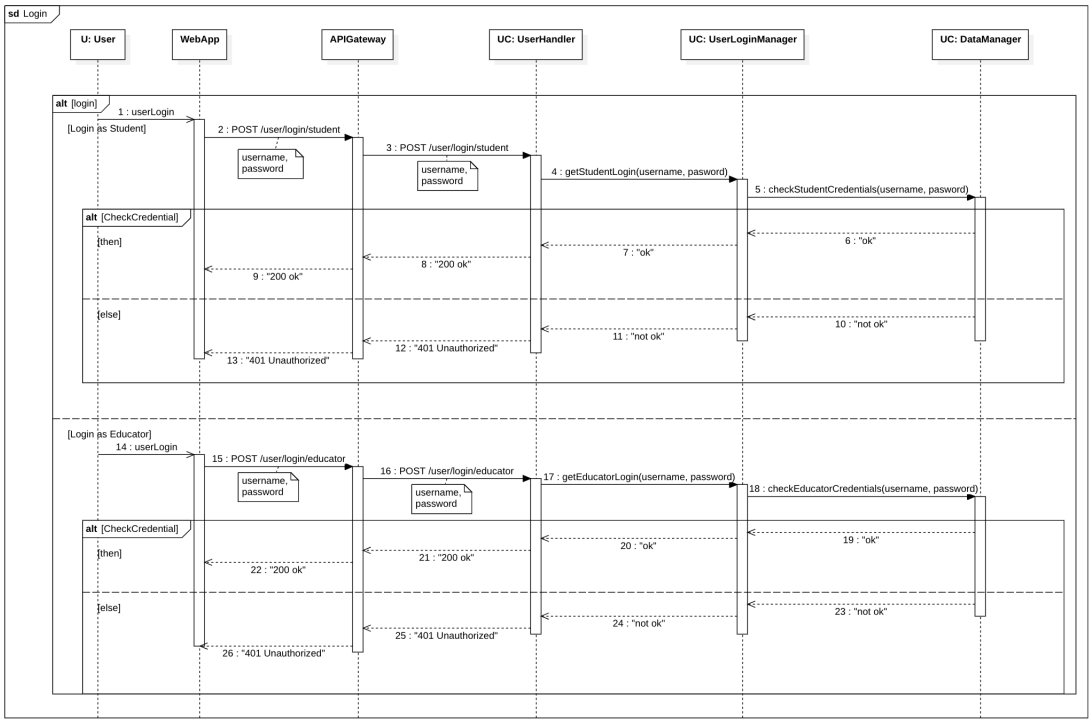

Figure 2.8: Login Runtime View

The diagram describes the access sequence to the platform. The user, via the WebApp, provides the login credentials, entering the username and password, and selects the desired type of access, as educator or student.
Next, the WebApp sends a POST request containing the username and password to the APIGateway. The latter, via the available REST API, routes the request to the "User Credential" microservice, involving the "UserHandleer" component, which forwards the request to the "UserLoginManager". The latter passes the request to the "DataManager", which verifies the correctness of the credentials provided. If the credentials are correct, the user receives a confirmation message; if not, an error message is returned.

[UC3] - Tournament Creation

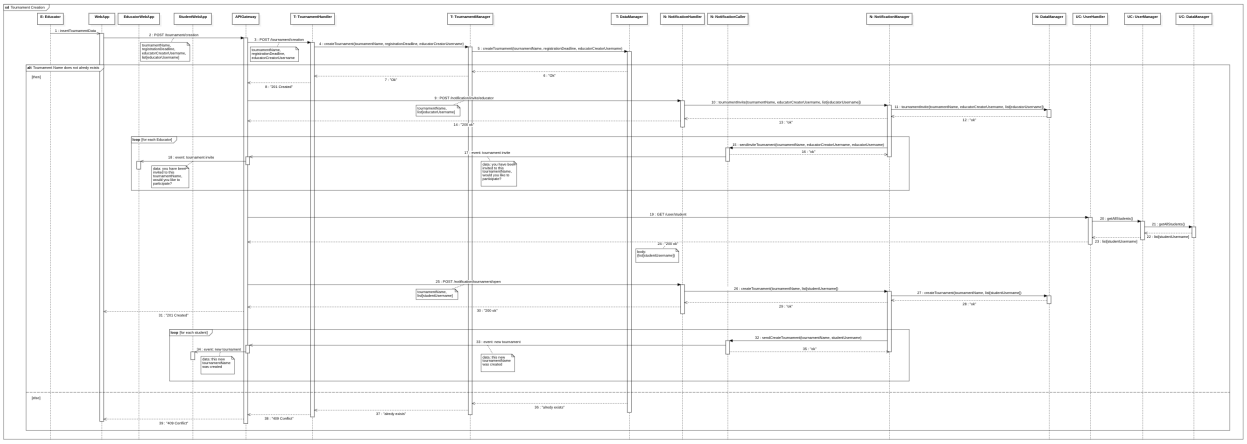

Figure 2.9: Tournament Creation Runtime View

The sequence diagram presented shows the process of creating a tournament within the system. The flow begins with the Educator entering the tournament data via the WebApp which sends a POST request to the APIGateway to create the tournament. The request includes the name of the Tournament, the Registration Deadline and the list of Educators invited to participate.
The request is forwarded to the "Tournament" microservice via the APIGateway, providing only the name of the tournament and the Registration Deadline. Within the 'Tournament' microservice, the tournament creation process takes place via its internal components. During this phase, a check is carried out on the tournament name entered. If the name already exists, an error message "409 Conflict" is returned to the educator. Otherwise, a "200 OK" message is returned confirming the correct creation of the tournament.
After the tournament has been created, notification to the educators comes via the "Notificacion" microservice. The APIGateway communicates with the "Notification" microservice by passing it the name of the tournament and the list of educators via the "Notification Handler" component. Subsequently, the data is forwarded to the "Notification Manager", which notifies each educator via the "Notification Caller", specifying the event and including the details in the body of the message. In addition, the data is stored in the database via the "Data Manager".
Finally, the APIGateway requests and retrieves the list of students enrolled in the platform via the "User Credentials" microservice. The GET request is handled by the "UserHandler" component, which forwards it to the "UserLoginManager" and then to the "DataManager". The "UserHandler" finally returns the list of students enrolled in the platform.
Once the list of enrolled students has been obtained, the APIGateway transmits the name of the tournament, the username of the educator who created it and the list of students to the "Notification" micro-service. The latter takes care of notifying all the students on the list of the creation of the tournament.

[UC4] - Battle Creation

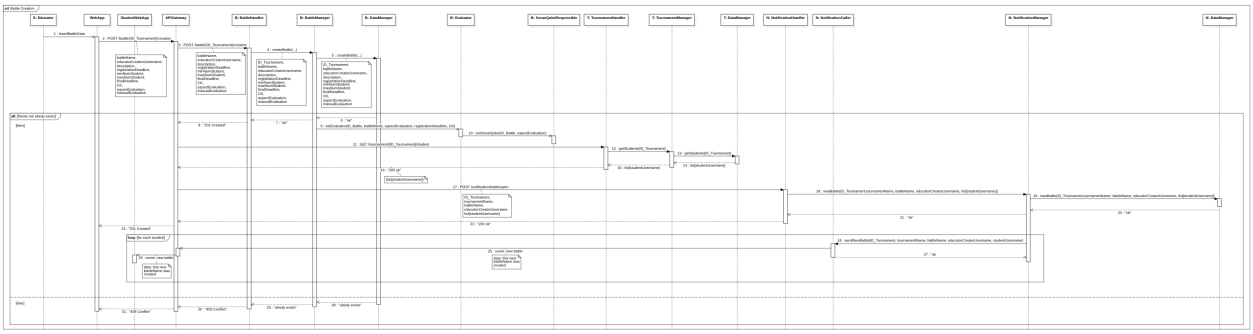

Figure 2.10: Battle Creation Runtime View

The sequence diagram highlights the process of creating a Battle. Initially, the Educator enters data via the WebApp. This data is transmitted to the APIGateway via a POST request generated by the WebApp. The request includes essential information such as the name of the Battle, a short description, the deadline for registration, the minimum and maximum number of students in the teams, together with the CodeKata file and the automatic and manual evaluation modes.
The APIGateway routes this request to the 'Battle' microservice, the "BattleHandler", which then communicates with the "BattleManager" and he with the "DataManager". During this procedure, the system verifies the uniqueness of the Battle name within the tournament. If it already exists, an error message '409 Conflict' is returned to the Educator; otherwise, the details of the Battle, including the ID, registration deadline and evaluation aspects, are communicated to the 'Evaluator'. Subsequently, the "Evaluator" forwards the evaluation aspects to the "SonarQubeResponsible", which in turn forwards them to SonarQube.
Next, the APIGateway makes a GET request to the "Tournament" microservice to retrieve the list of students registered for the tournament. It then communicates the details of the Battle and the list of students to the "Notification" microservice. The latter is responsible for notifying all students of the creation of the new Battle within the tournament.

[UC5] - Tournament closing

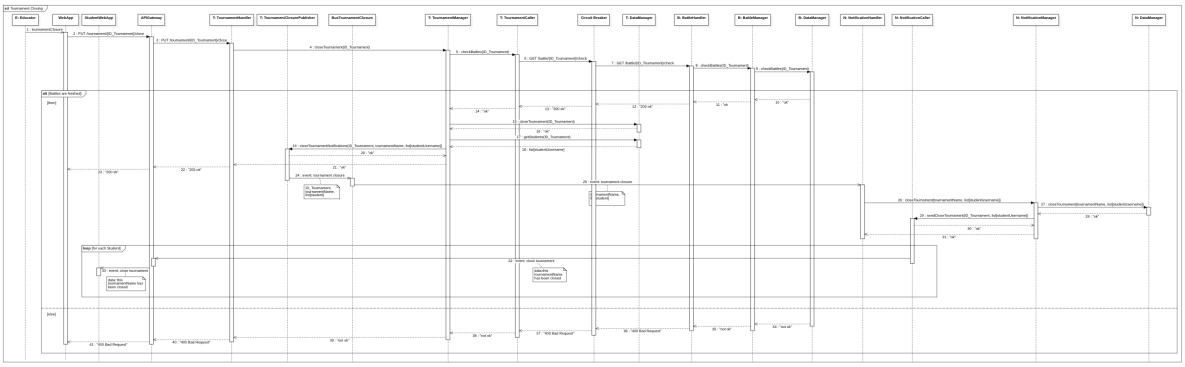

Figure 2.11: Tournament Closing Runtime View

The diagram illustrates the process of closing a tournament. The process starts with the intervention of the educator, responsible for management, who decides to close the tournament using the WebApp.
The WebApp generates and sends a PUT request to the APIGateway. The APIGateway routes this request to the microservice called "Tournament". The "TournamentHandler" within this microservice receives the request from the APIGateway and passes it on to the "TournamentManager". The latter handles the data flow, directing the request to the "DataManager", the component responsible for updating the tournament status within the database.
The WebApp generates and sends a PUT request to the APIGateway. The APIGateway routes this request to the microservice called "Tournament". The "TournamentHandler" within this microservice receives the request from the APIGateway and passes it to the "TournamentManager." The "TournamentManager" has to check if all battles are finished by calling the "TournamentCaller" component to make the request to the "Battles" microservice via a GET request using the Circuit Breaker. The "BattleHandler" receives the request and routes it to the "BattleManager" and "DataManager" components where the latter will check whether the battles are finished or not, forwarding the response to the previously listed components.

- If the "TournamentManager" receives that the battles have ended, it routes the request to close the tournament to the "DataManager," which is responsible for updating the tournament status in the database.
  Next, the "TournamentManager" retrieves the complete list of students registered for the tournament and sends it to the "TournamentClosurePublisher," which is responsible for sending the notification microservice to all students. The event, containing the name of the tournament and the list of students to be notified, is transmitted to the "BusTournamentClosure" through an asynchronous communication channel between the microservices. Subsequently, this message is routed to the "Notification" microservice.
  Within this microservice, the "NotificationHandler" receives the event and forwards it to the "NotificationManager", which passes it on to the

"DataManager" to save the notification to the database. Subsequently, the "NotificationManager" uses the "NotificationCaller", which forwards the message of the end of the tournament to the APIGateway, allowing notification to all registered students, informing them of the end of the tournament.

- If the "TournamentManager" receives that the battles are not finished, it forwards an error message to the APIGateway, the latter forwards an error message of type "400 Bad Request" to the WebApp.

[UC6] - Evaluation of final project

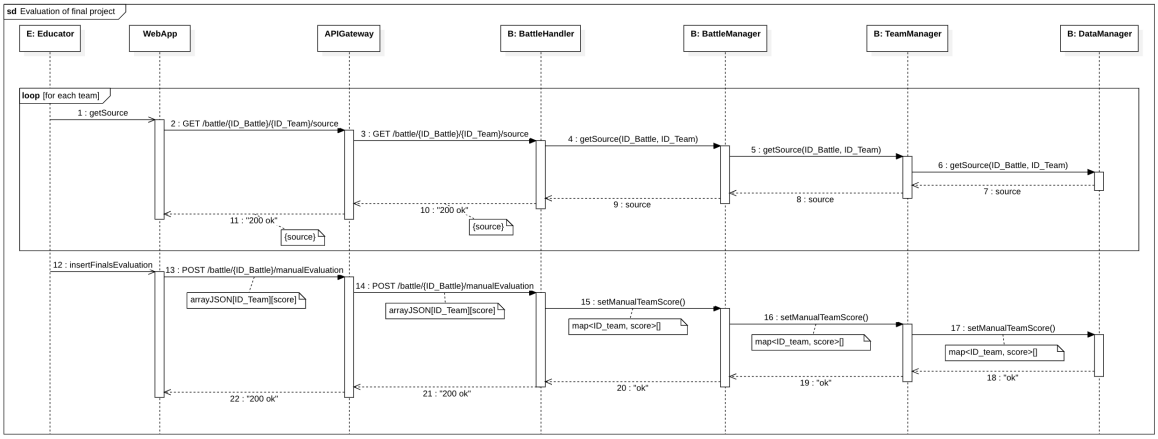

Figure 2.12: Evaluation of final project Runtime View

The sequence diagram depicts the process of entering the final assessment by the Educator. Through the WebApp, the Educator accesses the list of teams and their source code. To evaluate each team, the Educator requests the source code from the platform by making a GET request to the APIGateway. The APIGateway transmits this request to the "Battle" microservice which, through its components such as "BattleHandler", "BattleManager", "TeamManager" and "DataManager", retrieves the source file from the database.
Next, the Educator manually assigns the final score via the WebApp. The WebApp sends a POST request to the APIGateway containing an array containing the evaluated team and its score. The APIGateway forwards this request to the 'Battle' microservice, passing through the components described above. The "DataManager" then takes care of storing this data within the database."

[UC7] - Battle Ranking

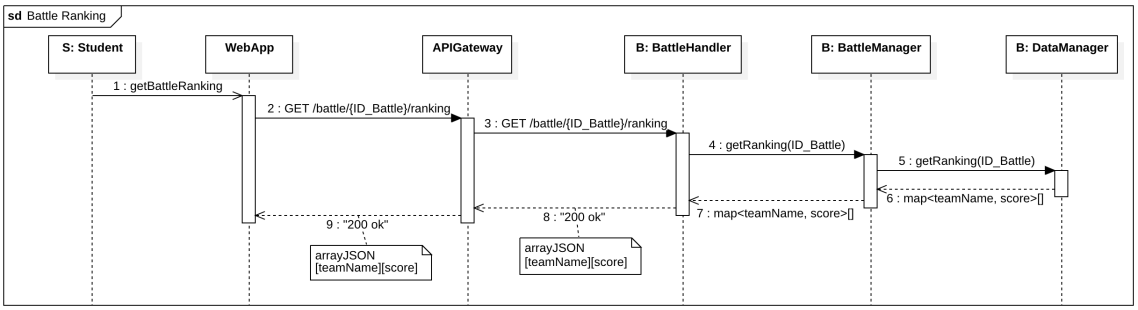

Figure 2.13: Battle Ranking Runtime View

The sequence diagram illustrates the process for displaying the battle ranking, both by a student enrolled in the battle and by the Educator who is its creator.
The student, via the WebApp, requests to view the battle ranking. The WebApp, via a GET request, establishes communication with the "Battle" microservice. The request passes through the various components of the microservice such as the "BattleHandler", "BattleManager", and the "DataManager". The latter retrieves the battle ranking, containing the names of the teams and their respective scores.

[UC8] - Tournament Ranking

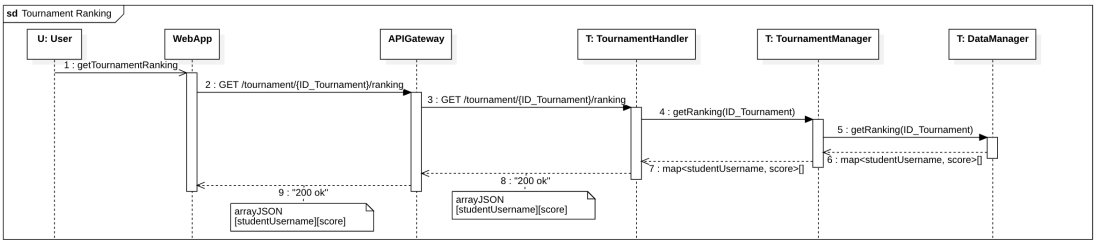

Figure 2.14: Tournament Ranking Runtime View

The sequence diagram presented illustrates the process of displaying the tournament ranking by a user registered on the platform.
Through the WebApp, the user requests the display of the tournament ranking. The WebApp, via a GET request, establishes communication with the "Tournament" microservice. The request is routed through the various components of the microservice such as the "TournamentHandler", "TournamentManager", and the "DataManager". Next, the tournament ranking containing the student's username and score is returned.

[UC9] - Signing up for a tournament

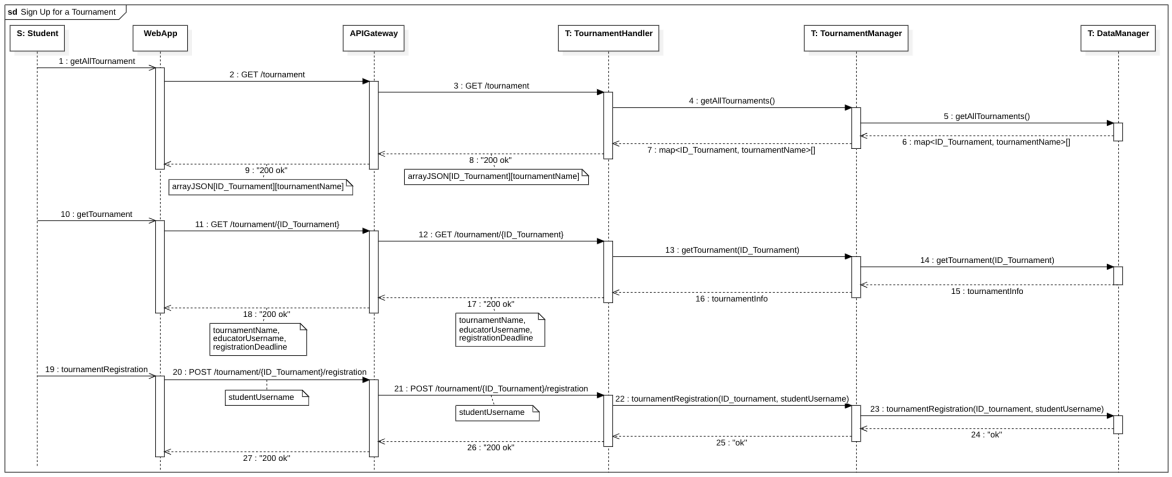

Figure 2.15: Tournament Registration Runtime View

The following sequence diagram shows the process of registering for a tournament. The Student, using the WebApp, requests the list of available tournaments via a GET request. This request is addressed to the APIGateway, which in turn handles communication with the "Tournament" microservice. The latter uses internal components such as the "TournamentHandler", the "TournamentManager" and the "DataManager" to obtain the list of currently offered tournaments and returns it to the WebApp via the APIGateway.
After viewing the list of tournaments, the student selects a tournament via the WebApp and sends a GET request specifying the ID of the chosen tournament. This request goes through the APIGateway, which communicates again with the "Tournament" microservice via the same chain of internal components in order to obtain the specific details of the selected tournament. The requested information is then returned from the WebApp via the APIGateway.
Next, the student proceeds with the tournament registration via the WebApp, sending a POST request containing his username. This request is handled by the APIGateway, which forwards the request to the "Tournament" microservice. The flow continues through the various internal components, until it reaches the "DataManager" which is responsible for entering the student's information into the database of tournament participants. Once the operation is complete, a confirmation message is generated and returned through the chain of components to the WebApp, confirming that the student has been registered for the tournament.

[UC10] - Unsubscription from the Tournament

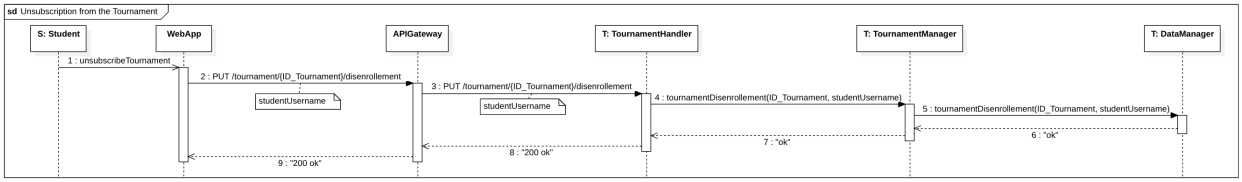

Figure 2.16: Unsubscription from the tournament Runtime View

The sequence diagram illustrates the process of unsubscribing from a tournament. Through the WebApp, the student unsubscribes from a tournament. The WebApp initiates a PUT request to the APIGateway, including the student's username. The latter APIGateway communicates with the "Tournament" microservice, forwarding the request through the various components, such as the "TournamentHandler", "TournamentManager", and the "DataManager". The latter component handles the operation of updating the student's status within the database.

[UC11] - View the list of battles

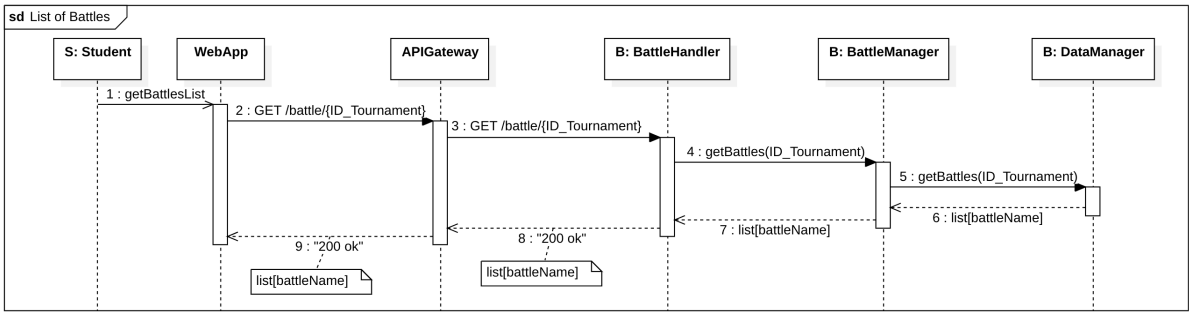

Figure 2.17: View the list of battles Runtime View

The sequence diagram represents the process of obtaining the list of battles within a tournament by a registered student.
Through the WebApp, the student requests the battle list and sends a GET request to the APIGateway. The latter routes the request to the "Battle" microservice, via the various components such as "BattleHandler", "BattleManager" and "DataManager". It is the DataManager that takes care of retrieving the information on the battles in the specific tournament context, from the database.

[UC12] - Signing up for a battle

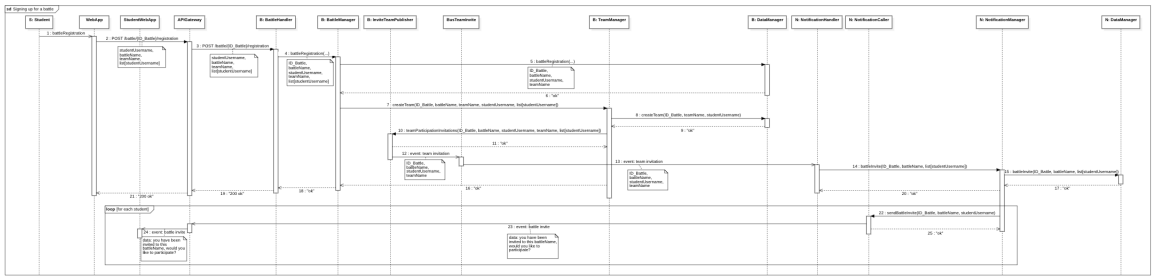

Figure 2.18: Signing up for a battle Runtime View

The sequence diagram shows the process of a student registering for a battle.
The student uses the WebApp's dedicated interface to enter the necessary details for the battle entry, such as the team name and the list of selected students. Once the data is filled in, the WebApp sends a POST request to the APIGateway, containing the battle information and the student's username.
The APIGateway forwards the request to the "Battle" microservice, including the specific battle ID. Within the microservice, the request passes through several internal components such as the "BattleHandler", the "BattleManager", and the "DataManager". The latter takes care of storing the details of the battle registration in the database, confirming the successful registration.
Next, the "BattleManager" communicates the creation of the team to the "TeamManager". The latter, in turn, coordinates with the "DataManager" to save the information on the team formed by the student at the time of battle registration in the database.
In parallel, information about the battle, the student who created the team and the list of students is sent to the "InviteTeamPublisher". The latter, asynchronously via the "BusTeamInvite", connects to the "Notification" microservice to communicate with the "NotificationHandler".
The "NotificationHandler" routes the received data to the "NotificationManager", which in turn passes it on to the "DataManager" to save the invitation notification in the database. Next, the "NotificationManager" passes the data to the "NotificationCaller", which sends an invitation message to each student in the list. Finally, the "NotificationCaller" communicates with the APIGateway to forward the invitation messages to the relevant students.

[UC13] - Push effects on GitHub

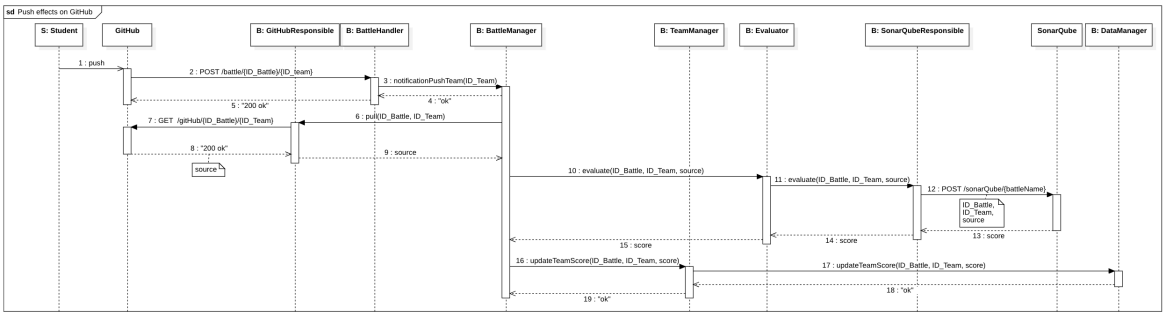

Figure 2.19: Push effects on GitHub Runtime View

The following sequence diagram shows what happens when a student makes a push on GitHub.
The process starts when the student performs a project push on GitHub. GitHub, notifies the microservice "Battle" about the push made by a team on a specific battle, transmitting the ID of the team involved.
The "BattleHandler" receives the notification and forwards it to the "BattleManager". Subsequently, the "BattleManager" requests the pull of the team via its ID to the "GitHubResponsible", which then executes a GET request to GitHub to retrieve the corresponding source file.
Once the source file has been obtained, the "BattleManager" transmits it to the "Evaluator" component, which is responsible for calculating functional aspects and timeliness. In addition, the "Evaluator" forwards the source file and the team ID to the "SonarQubeResponsible" component.
The "SonarQubeResponsible" communicates with the SonarQube platform, which calculates the score for the source file and returns it to the "BattleManager" via a path that again involves the "SonarQubeResponsible" and the "Evaluator".
Once the "BattleManager" has received the score, it updates the team's score. This update takes place by sending the new score to the "TeamManager", which then forwards this information to the "DataManager" which is responsible for updating the score within the database for the team involved in the battle.

[UC14] - Participation in a Battle by invitation

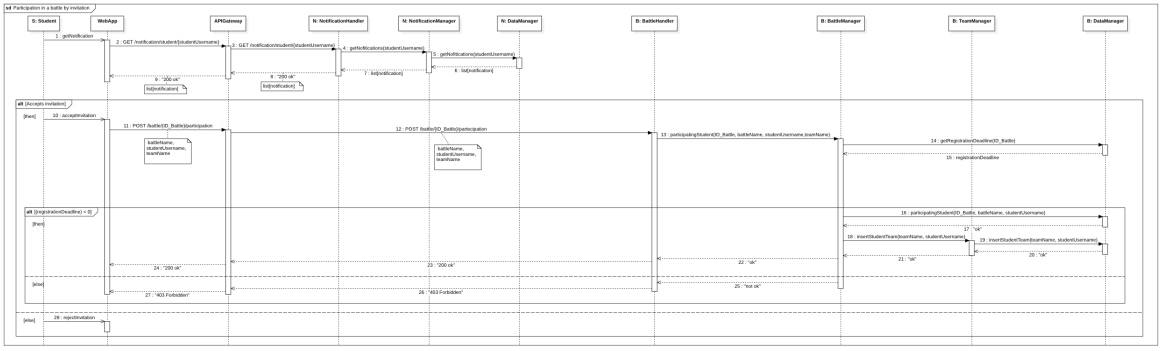

Figure 2.20: Participation in a Battle by invitation Runtime View

The sequence diagram represents the process of participating in a battle by invitation by the student who created the team during the battle entry.
The student accesses the notification area via the WebApp interface. The WebApp initiates a GET request to the APIGateway, which routes this request to the "Notification" microservice. Through internal components such as the "NotificationHandler", the "NotificationManager", and the "DataManager", student-specific notifications are retrieved and returned in response to the request.
The student has the option of accepting or rejecting the invitation. In the case of a refusal, no further action is taken.
If the student accepts the invitation, the WebApp sends a POST request to the APIGateway, including the Battle information obtained through the notification, together with the student's username. Subsequently, the APIGateway forwards this request to the "Battle" microservice.
The "BattleHandler" of the "Battle" microservice receives the participation request and forwards it to the "BattleManager". The latter performs a check to determine whether the registration period for the battle is still open. To do this, it asks the 'DataManager' for the expiry date for registration. If the specified date has expired, the "BattleManager" reports an error to the "BattleHandler". The latter transmits the error to the APIGateway, which communicates it to the WebApp. If the registration period has not yet expired, the "BattleManager" instructs the "DataManager" to register both the battle entry and the student's team entry. Subsequently, a confirmation message is returned to the "BattleHandler", which forwards it to the APIGateway and then to the WebApp.

[UC15] - Participation in a Tournament by invitation

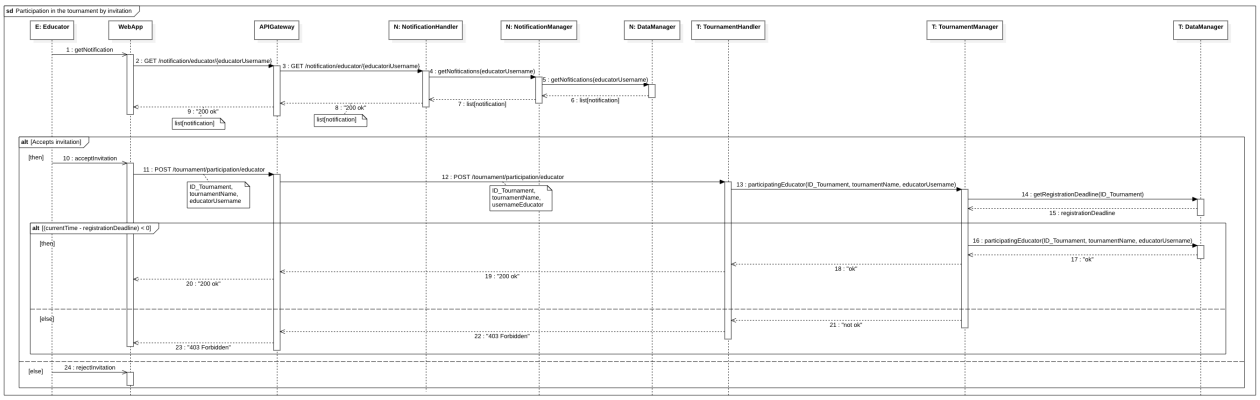

Figure 2.21: Participation in a Tournament by invitation Runtime View

The sequence diagram represents the process of participating in a tournament by invitation by the educator who created it.
The educator, via the WebApp, accesses the notification area. The WebApp initiates a GET request to the APIGateway, which routes the request to the 'Notification' microservice. Via the internal components: the "NotificationHandler", the "NotificationManager" and the "DataManager", the specific educator notifications are retrieved and returned.
At this point, the educator may decide to accept or reject the invitation. If he opts for rejection, no further action is required by the system.
If the educator accepts the invitation, a POST request is sent via the WebApp to the APIGateway, including the tournament information obtained through the notification, together with the educator's username. Subsequently, the APIGateway forwards this request to the 'Tournament' microservice.
The "TournamentHandler" component of the "Tournament" microservice receives the participation request and forwards it to the "TournamentManager". The latter performs a check to determine whether or not the registration period for the tournament has expired. To do this, it asks the "DataManager" for the expiry date for registration and, if the period has expired, sends an error message to the "TournamentHandler". The latter routes the error to the APIGateway, which forwards it to the WebApp. If, on the other hand, the registration period has not yet expired, the "TournamentManager" instructs the "DataManager" to register the educator's tournament entry and returns a confirmation message to the WebApp via the path already followed.

2.5 Component interfaces

2.5.1 User Credentials Microservice

2.5.1.1 REST API

Student registration to the platform

EndPoint:	POST /user/registration/student
Description:	This endpoint allows the registration of a new student on the platform.
Parameters of the Body of the Request:	• 'Firstname': Student's name • 'Lastname': Student's last name • 'Mail': Student Mail • 'Username': Username chosen by the student • 'Password': Password chosen by the student
Response states:	• '201 Created': The resource was successfully created. The body of the response is empty. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Educator registration to the platform

EndPoint:	POST /user/registration/educator
Description:	This endpoint allows the registration of a new educator on the platform.
Parameters of the Body of the Request:	• 'Firstname': Educator's name • 'Lastname': Educator's last name • 'Mail': Educator Mail • 'Username': Username chosen by the educator • 'Password': Password chosen by the educator
Response states:	• '201 Created': The resource was successfully created. The body of the response is empty. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Student Login

EndPoint:	POST /user/login/student
Description:	This endpoint allows a student to login to the platform.
Parameters of the Body of the Request:	• 'Username': Student username • 'Password': Student password
Response states:	• '200 OK': The request was successfully executed. The body of the response is empty. • '401 Unauthorized': The user is not authorized to perform the requested action.

Educator Login

EndPoint:	POST /user/login/educator
Description:	This endpoint allows a educator to login to the platform.
Parameters of the Body of the Request:	• 'Username': Educator username • 'Password': Educator password
Response states:	• '200 OK': The request was successfully executed. The body of the response is empty. • '401 Unauthorized': The user is not authorized to perform the requested action.

List of students on the platform

EndPoint:	GET /user/student
Description:	This endpoint allows you to get all students enrolled in the platform.
Response states:	• '200 OK': The request was executed successfully. The body of the response contains the list of students enrolled in the platform. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

2.5.1.2 Interfaces

- **UserManagerI**
  - List<String studentUsername>: getAllStudent()
- **RegistrationManagerI**
  - Boolean: newStudentRegistration(String firstname, String lastname, String mail, String username, String password)
  - Boolean: newEducatorRegistration(String firstname, String lastname, String mail, String username, String password)
- **LoginManagerI**
  - Boolean: checkStudentCredentials(String username, String password)

- Boolean: checkEducatorCredentials(String username, String password)

- **DataManagerI**
  - Boolean: newStudentRegistration(String firstname, String lastname, String mail, String username, String password)
  - Boolean: newEducatorRegistration(String firstname, String lastname, String mail, String username, String password)
  - Boolean: checkStudentCredentials(String username, String password)
  - Boolean: checkEducatorCredentials(String username, String password)
  - List<String studentUsername>: getAllStudent()

2.5.2 Tournament Microservice

2.5.2.1 REST API

Tournament creation

EndPoint:	POST /tournament/creation
Description:	This endpoint allows an educator to create a tournament.
Parameters of the Body of the Request:	• 'Tournament Name': Tournament name • 'Registration Deadline ': Registration deadline for the tournament • 'EducatorCreatorUsername': Tournament creator username
Response states:	• '201 Created': The resource was successfully created. The body of the response is empty. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

List of students registered for a tournament

EndPoint:	GET /tournament/{ID_Tournament}/students
Description:	This endpoint returns the list of all students participating in the tournament.
Response states:	• '200 OK': The request was executed successfully. The body of the response contains the list of students participating in the tournament. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Closing of the tournament

EndPoint:	PUT /tournament/{ID_Tournament}/close
Description:	This endpoint allows you to close a tournament
Response states:	• '200 OK': The request was executed successfully. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

List of students registered for a tournament

EndPoint:	GET /tournament/{ID_Tournament}/ranking
Description:	This endpoint allows the tournament ranking to be obtained.
Response states:	• '200 OK': The request was executed successfully. The body of the response contains the list of students registered for the tournament with their scores. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Tournament list

EndPoint:	GET /tournament
Description:	This endpoint allows you to get the list of all tournaments.
Response states:	• '200 OK': The request was executed successfully. The body of the response contains the list of all tournaments. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Tournament info

EndPoint:	GET /tournament/{ID_Tournament}
Description:	This endpoint allows you to get the tournament information.
Response states:	• '200 OK': The request was executed successfully. The body of the response contains the name of the tournament, the username of the tournament creator, and the registration deadline. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Tournament Registration

EndPoint:	POST /tournament/{ID_Tournament}/registration
Description:	This endpoint allows a student to register
Parameters of the Body of the Request:	• 'Student username': the student's username.
Response states:	• '200 OK': The request was executed successfully. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Tournament disenrollment

EndPoint:	PUT /tournament/{ID_Tournament}/disenrollment
Description:	This endpoint allows a student to unsubscribe from a tournament
Parameters of the Body of the Request:	• 'Student username': the student's username.
Response states:	• '200 OK': The request was executed successfully. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Educator registration for a tournament

EndPoint:	POST /tournament/participation/educator
Description:	This endpoint allows an educator to participate in a tournament.
Parameters of the Body of the Request:	• 'ID_Tournamet': Tournament ID • 'Name Tournament': Name Tournament • 'Username educator': Username educator
Response states:	• '200 OK': The request was executed successfully. • '403 Forbidden':The user does not have the necessary permissions to access the resource.

2.5.2.2 Interfaces

- **TournamentManagerI**
  - Boolean: createTournament(String tournamentName, ZonedDateTime registrationDeadline, String educatorCreatorUsername)
  - Boolean: closeTournament(int ID_Tournament)
  - Boolean: updateRankingTournament(int ID_Tournament, Map<String, int> mapStudentUsernameScore)
  - Boolean: tournamentRegistration(int ID_Tournament, String studentUsername)
  - Boolean: tournamentDisenrollment(int ID_Tournament, String studentUsername)
  - Map<String studentUsername, int score>: getRanking(int ID_Tournament)
  - Map<int ID_Tournament, String tournamentName>: getAllTournaments()
  - Boolean: partecipatingEducator(int ID_Tournament, String tournamentName, String educatorUsername)
  - ZonedDateTime: getRegistrationDeadline(int ID_Tournament)
  - List<String studentUsername>: getStudents(int ID_Tournaments)
  - TournamentInfo: getTournament(int ID_Tournament)
- **TournamentCallerI**
  - Boolean: checkBattles(int ID_Tournament)
- **TournamentClosurePublisherI**
  - Boolean: closeTournamentNotifications(int, ID_Tournament, String tournamentName, List<String> studentUsernameList)
- **DataManagerI**
  - Boolean: createTournament(String tournamentName, ZonedDateTime registrationDeadline, String educatorCreatorUsername)
  - Boolean: closeTournament(int ID_Tournament)
  - Boolean: updateRankingTournament(int ID_Tournament, Map<String, int> mapStudentUsernameScore)
  - Boolean: tournamentRegistration(int ID_Tournament, String studentUsername)
  - Boolean: tournamentDisenrollment(int ID_Tournament, String studentUsername)
  - Map<String studentUsername, int score>: getRanking(int ID_Tournament)
  - Map<int ID_Tournament, String tournamentName>: getAllTournaments()

- Boolean: partecipatingEducator(int ID_Tournament, String tournamentName, String educatorUsername)
- ZonedDateTime: getRegistrationDeadline(int ID_Tournament)
- List<String studentUsername>: getStudents(int ID_Tournament)
- TournamentInfo: getTournament(int ID_Tournament)

In addition, we have implemented a structure called **TournamentInfo**, which includes the following parameters:

- String tournamentName
- String educatorUsername
- ZonedDateTime registrationDeadline

2.5.3 Notification Microservice

2.5.3.1 REST API

Tournament opening notification

EndPoint:	POST /notification/tournament/open
Description:	This endpoint allows all students registered on the platform to be notified of the opening of the new tournament.
Parameters of the Body of the Request:	• 'Tournament name': Tournament name • 'List[studentUsername]': List of students to be notified
Response states:	• '200 OK': The request was executed successfully. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Notification of educator invitation to a tournament

EndPoint:	POST /notification/invite/educator
Description:	This endpoint allows the educator to invite other educators to participate in their tournament.
Parameters of the Body of the Request:	• 'Tournament name': Tournament name • 'List[educatorUsername]': List of educator to be notified
Response states:	• '200 OK': The request was executed successfully. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Battle opening notification

EndPoint:	POST /notification/battle/open
Description:	This endpoint allows notification of all students enrolled in that tournament of the opening of the new battle.
Parameters of the Body of the Request:	• 'ID_Tournamet': Tournament ID • 'educatorCreatorUsername': Educator creator username • 'Tournament name': Tournament name • 'Battle name': Battle name • 'List[studentUsername]': List of students to be notified
Response states:	• '200 OK': The request was executed successfully. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

List of student notifications

EndPoint:	GET /notification/student/{studentUsername}
Description:	This endpoint allows you to get all notifications from a student.
Response states:	• '200 OK': The request was executed successfully. The body of the response contains the list of all notifications of the relevant student. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

List of educator notifications

EndPoint:	GET /notification/educator/{educatorUsername}
Description:	This endpoint allows you to get all notifications from a educator.
Response states:	• '200 OK': The request was executed successfully. The body of the response contains the list of all notifications of the relevant educator. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

2.5.3.2 Interfaces

- **NotificationManagerI**
  - Boolean: tournamentInvite(String tournamentName, String educatorCreatorUsername, List<String> educatorUsername)
  - Boolean: createTournament(String tournamentName, String, List<String> studentUsername)
  - Boolean: closeTournament(String tournamentName, List<String> studentUsername)
  - Boolean: newBattle(int ID_Tournament, String tournamentName, String battleName, String educatorCreatorUsername, List<String> studentUsername)
  - Boolean: battleInvite(int ID_Battle, String battleName, List<String> studentUsername)
  - List<String notification>: getNotifications(String username)
  - Boolean: gitHubLinkNotification(String tournamentName, String battleName, List<String> studentUsername, String linkGitHub)
  - Boolean: closeBattle(String tournamentName, String battleName, List<String> studentUsername)
- **NotificationCallerI**
  - Boolean: sendInviteTournament(String tournamentName, String educatorCreatorUsername, String educatorUsername)
  - Boolean: sendCreateTournament(String tournamentName, List<String> studentUsername)
  - Boolean: sendCloseTournament(int ID_Tournament, List<String> studentUsername)
  - Boolean: sendNewBattle(int ID_Tournament, String tournamentName, String battleName, String educatorCreatorUsername, String studentUsername)

- Boolean: sendBattleInvite(int ID_Battle, String battleName, String studentUsername)
- Boolean: gitHubLinkNotification(String tournamentName, String battleName, List<String> studentUsername, String linkGitHub)
- Boolean: closeBattle(String tournamentName, String battleName, List<String> studentUsername)

- **DataManagerI**
  - Boolean: tournamentInvite(String tournamentName, String educatorCreatorUsername, List<String> educatorUsername)
  - Boolean: createTournament(String tournamentName, List<String> studentUsername)
  - Boolean: closeTournament(String tournamentName, List<String> studentUsername)
  - Boolean: newBattle(int ID_Tournament, String tournamentName, String battleName, String educatorCreatorUsername, List<String> studentUsername)
  - Boolean: battleInvite(int ID_Battle, String battleName, List<String> studentUsername)
  - List<String notification>: getNotifications(String username)
  - Boolean: gitHubLinkNotification(String tournamentName, String battleName, List<String> studentUsername, String linkGitHub)
  - Boolean: closeBattle(String tournamentName, String battleName, List<String> studentUsername)

2.5.4 Battle Microservice

2.5.4.1 REST API

Battle creation

EndPoint:	POST /battle/{ID_Tournament}/creation
Description:	This endpoint allows an educator to create a new battle in the context of the tournament present in the URL.
Parameters of the Body of the Request:	• 'battleName ': Battle name • 'educatorCreatorUsername': Battle creator username • 'description': Battle description • 'registrationDeadline': Deadline for registration from the battle • 'minNumStudent': Minimum number of students per team • 'maxNumStudent': Maximum number of students per team • 'finalDeadline': Deadline for delivery of the final project • 'CK': The CodeKata project, which teams will have to carry out in the battle • 'aspectEvaluation': Aspects that will be evaluated by the source analysis tool • 'manualEvaluation': If the educator wishes to perform manual analysis of the source
Response states:	• '200 OK': The request was executed successfully. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Source of a team

EndPoint:	GET /battle/{ID_Battle}/{ID_Team}/source
Description:	This endpoint allows you to get the source of a team.
Response states:	• '200 OK': The request was executed successfully. The body of the response contains the source of the relevant team. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Manual evaluation

EndPoint:	POST /battle/{ID_Battle}/manualEvaluation
Description:	This endpoint allows a score to be assigned to each source of each team in a battle
Parameters of the Body of the Request:	• 'ArrayJSON[ID_Team][Score]': An array with a pair inside representing the team ID and its score.
Response states:	• '200 OK': The request was executed successfully. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Ranking battle

EndPoint:	GET /battle/{ID_Battle}/ranking
Description:	This endpoint allows you to get the ranking of the battle.
Response states:	• '200 OK': The request was executed successfully. The body of the response contains the list of teams with their scores. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Enrollment in a battle

EndPoint:	POST /battle/{ID_Battle}/registration
Description:	This endpoint allows a student to sign up for a battle, form a team and invite other students.
Parameters of the Body of the Request:	• 'studentUsername': Name of student who wants to enroll. • 'battleName': Battle name. • 'teamName': Team name. • 'list[studentUsername]': List of students that the creators decided to invite
Response states:	• '200 OK': The request was executed successfully. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Participation in the battle by invitation

EndPoint:	POST /battle/{ID_Battle}/participation
Description:	This endpoint allows a student to join the battle by invitation.
Parameters of the Body of the Request:	• 'battleName': Name of student who wants to enroll. • 'studentUsername': Battle name. • 'teamName': Team name.
Response states:	• '200 OK': The request was executed successfully. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Team push notification

EndPoint:	POST /battle/{ID_Battle}/{ID_Team}
Description:	This endpoint allows the microservice to know when a team has made a push to its main repository
Response states:	• '200 OK': The request was executed successfully. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Battle list

EndPoint:	GET /battle/{ID_Tournament}
Description:	This endpoint allows the list of battles in a tournament.
Response states:	• '200 OK': The request was executed successfully. The body of the response contains the list of all battle in a tournament. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

Check Battles

EndPoint:	GET /battle/{ID_Tournament}/check
Description:	This endpoint allows to know if all the battles in the tournament are completed.
Response states:	• '200 OK': The request was executed successfully. In the body of the response includes true if all the battles are finished or else false. • '400 Bad Request': The request is syntactically incorrect or cannot be processed. A detailed error message is included in the body of the response.

2.5.4.2 Interfaces

- **BattleManagerI**
  - Boolean: createBattle(int ID_Tournament, String battleName, String educatorCreatorUsername, String description, ZonedDateTime

registrationDeadline, int min, int max, ZonedDateTime finalDeadline, CodeKata CK, List<String> aspectEvaluation, Boolean manualEvaluation)

- List<String battleName> : getBattles(int ID_Tournament)
- Map<int ID_Team, int score>: getRanking(int ID_Battle)
- Boolean: battleRegistration(int ID_Battle, String battleName, String studentUsername, String teamName, List<String> studentUsername)
- Boolean: participatingStudent(int ID_Battle, String battleName, String studentUsername, String teamName)
- ZonedDateTime: getRegistrationDeadline(int ID_Battle)
- File: getSource(int ID_Battle, int ID_Team)
- Boolean: setManualTeamScore(Map<int, int> mapTeamScore)
- Boolean: notificationPushTeam(int ID_Team)
- Boolean: checkBattles(int ID_Tournament)

- **InviteTeamPublisherI**
  - Boolean: teamParticipationInvitations(int ID_Battle, String battleName, String studentUsername, String teamName, List<String> studentUsername)
- **GitHubResponsibleI**
  - String link: createRepository(int ID_Battle, String educatorUsername, CodeKata CK)
  - File: pull(int ID_Battle, int ID_Team)
- **TeamManagerI**
  - File: getSource(int ID_Battle, int ID_Team)
  - Boolean: createTeam(int ID_Battle, String battleName, String teamName, String studentUsername, List<String> studentUsername)
  - Boolean: insertStudentTeam(String teamName, String studentUsername)
  - Boolean: updateTeamScore(int ID_Battle, int ID_Team, int score)
  - Boolean: setManualTeamScore(Map<int, int> mapTeamScore)
- **DataManagerI**
  - Boolean: createBattle(int ID_Tournament, String battleName, String educatorCreatorUsername, String description, ZonedDateTime registrationDeadline, int min, int max, ZonedDateTime finalDeadline, List<String> aspectEvaluation, Boolean manualEvaluation)
  - List<String battleName>: getBattles(int ID_Tournament)
  - Map<int ID_Team, int score>: getRanking(int ID_Battle)
  - Boolean: createTeam(int ID_Battle, String battleName, String teamName, String studentUsername, List<String> studentUsername)

- Boolean: battleRegistration(int ID_Battle, String battleName, String studentUsername, String teamName, List<String> studentUsername)
- File: getSource(int ID_Battle, int ID_Team)
- Boolean: participatingStudent(int ID_Battle, String battleName, String studentUsername, String teamName)
- Boolean: insertStudentTeam(String teamName, String studentUsername)
- ZonedDateTime: getRegistrationDeadline(int ID_Battle)
- Boolean: updateTeamScore(int ID_Battle, int ID_Team, int score)
- Boolean: setManualTeamScore(Map<int, int> mapTeamScore)
- Boolean: checkBattles(int ID_Tournament)

- **EvaluatorI**
  - void: setEvaluator(int ID_Battle, String battleName, List<String> aspectEvaluation, ZonedDateTime registrationDeadline, CodeKata CK)
  - int: evaluate(int ID_Battle,int ID_Team, File source)

- **SonarQubeResponsibleI**
  - void: setSonarQube(int ID_Battle, List<String> aspectEvaluation)
  - int: evaluate(int ID_Battle, int ID_Team, File source)

- **BattleClosurePublisherI**
  - Boolean: battleClosureNotification(String tournamentName, String battleName, List<String> studentUsername)

- **FinalScoreBattlePublisherI**
  - Boolean: publishingFinalBattleScores(String tournamentName, String battleName, Map<String, int> mapStudentUsernameScore)
- **GitHubLinkPublisherI**
  - Boolean: gitHubLinkNotification(String tournamentName, String battleName, List<String> studentUsername, String link)

In addition, we have implemented a structure called **CodeKata**, which includes the following parameters:

- String language
- String description
- ZipFile codeKataZip

2.6 Selected architectural Style and Patterns

2.6.1 Microservices architecture

The microservices architectural style is a design approach in which an application is divided into autonomous components, known as microservices, each of which handles a specific functionality or service. These microservices operate independently and communicate with each other through lightweight mechanisms such as HTTP APIs or asynchronous messaging.
The microservice architecture promotes modularity, scalability, and development flexibility. Each microservice is self-sufficient and can be developed, tested, and deployed independently of the others. This separation enables rapid development and release of new functionality.
In addition, the microservice approach simplifies management and maintenance of the system as a whole. Developers can focus on individual microservices without the need to understand the entire system, facilitating collaboration and work distribution.

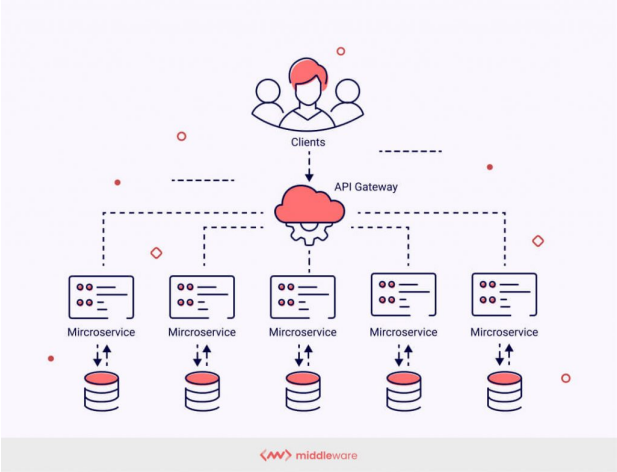

Figure 2.22: Microservices architecture

2.6.2 API Gateway Pattern

The introduction of the API Gateway pattern into a microservices architecture offers several strategic advantages. The API Gateway serves as a centralized entry point, simplifying traffic management and providing a unified interface for external requests. This component manages the routing of requests based on specific criteria, hiding the complexity of the underlying architecture.
In addition, the Gateway API can aggregate responses from different microservices, reducing the number of network calls required to fulfill complex requests. It manages critical aspects of security, including user authentication and resource authorization, providing a central control point for enforcing security policies.
To improve performance, the Gateway API implements a response cache, speeding up responses to repeated requests. It also provides broad visibility into API traffic, simplifying performance monitoring and troubleshooting. The API Gateway supports API versioning, facilitating the gradual evolution of interfaces over time.

Overall, this pattern optimizes API management in a microservice context, helping to improve the security, scalability, and maintainability of the entire distributed system.

2.6.3 Client-Side Rendering (CSR)

In the CSR approach, the loading and construction of HTML pages takes place client-side. After receiving data from the microservices, client-side JavaScript code takes care of DOM manipulation and dynamic page creation. This approach offers high interactivity, but requires the browser to download and execute JavaScript code to display the page.

2.7 Other design decisions

2.7.1 Microservices communication and circuit breaker pattern

The microservice architecture uses an event bus as an asynchronous communication channel between the various microservices. This bus allows microservices to exchange information, represented in the form of events, in a flexible and decoupled manner. Events can be published by producer microservices and subscribed to by interested microservices, contributing to the scalability, decoupling and flexibility of the architecture.
To handle synchronous calls between microservices and ensure system resilience, the implementation of a circuit breaker pattern was adopted. The circuit breaker is a proactive strategy that involves handling errors in microservice calls, providing protection against prolonged or repeated failures.
The circuit breaker operates by monitoring service calls and, when it detects a certain number of consecutive failures or latency above a predetermined threshold, temporarily interrupts the call, thus preventing further attempts.

3 User Interface Design

The two pages available for access and registration to the platform are illustrated respectively in Figures 3.1 e 5.2.

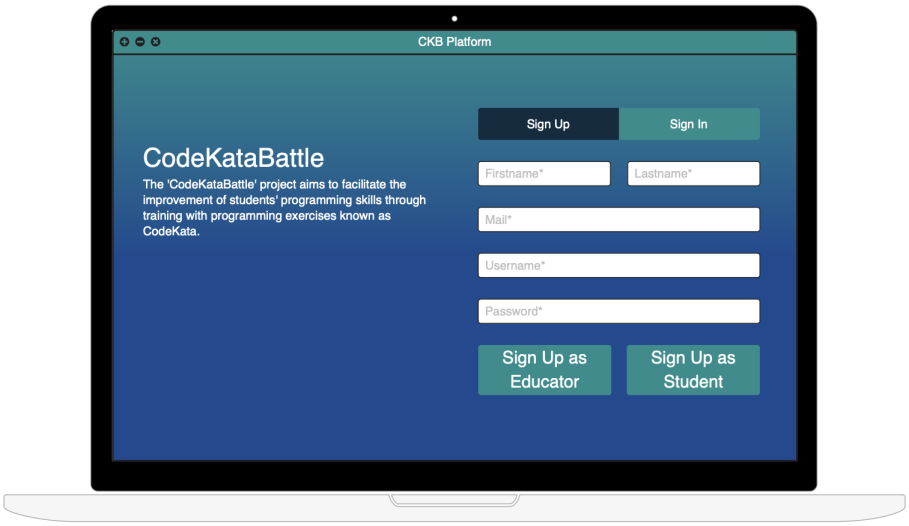

Figure 3.1: Registration Page on the Platform

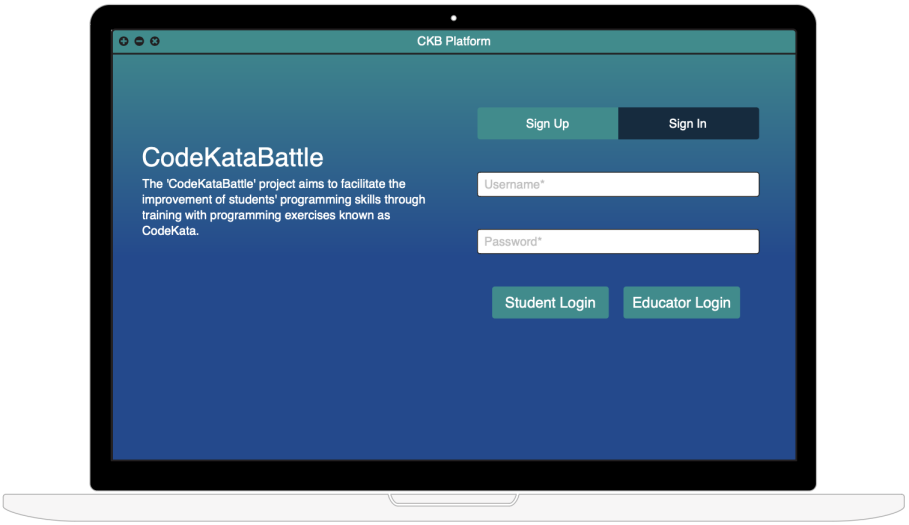

Figure 3.2: Login Page on the Platform

3.1 Student Interface

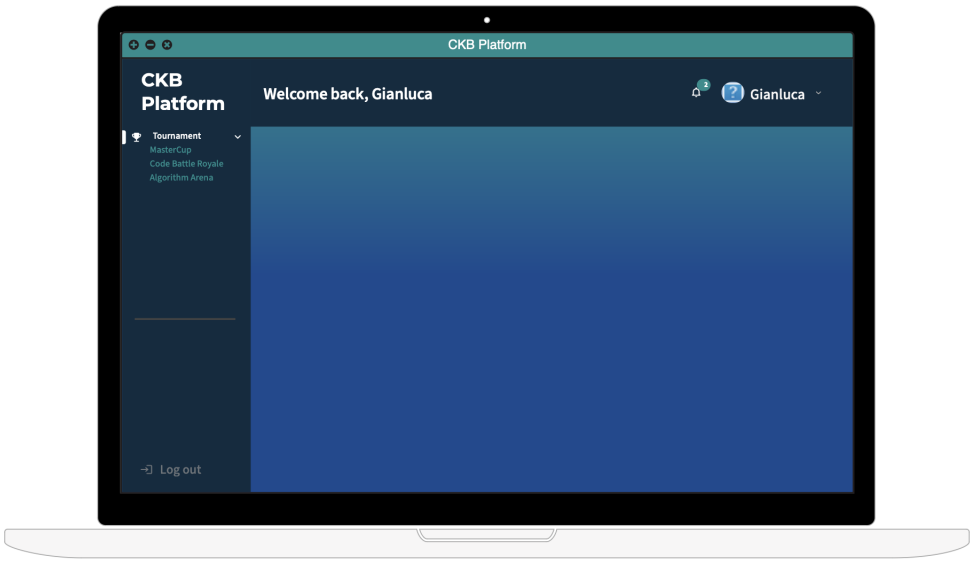

Figure 3.3: Student Dashboard

Figure 3.3 shows the dashboard for users accessing the platform as students, highlighting a list of available tournaments.

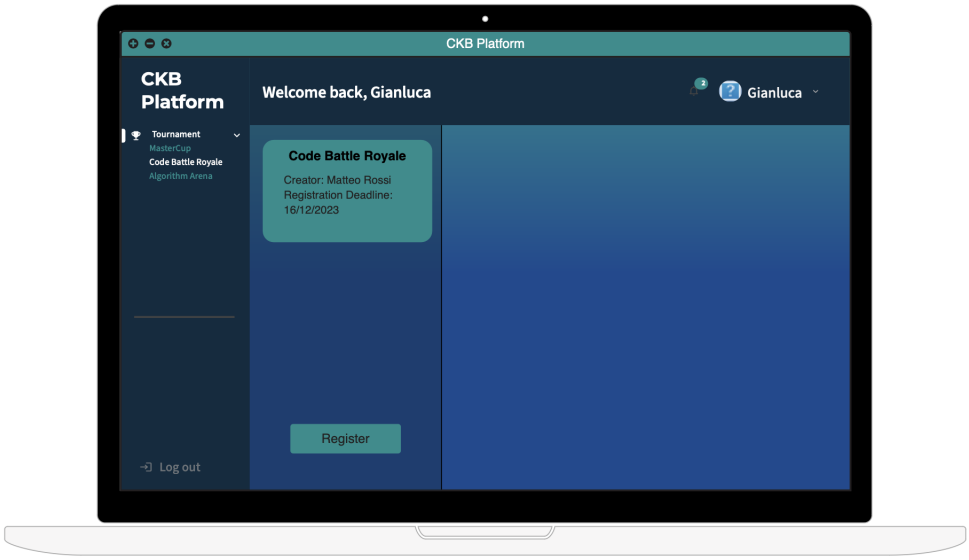

Figure 3.4: Student registers for a tournament

Figure 3.4 shows the detail screen of a tournament, which includes a box with the name of the tournament, the educator who created it and the registration deadline. In addition, there is a 'Register' button, which is active when the student has not yet registered for the tournament. By clicking on this button, the registration for the tournament can be completed.

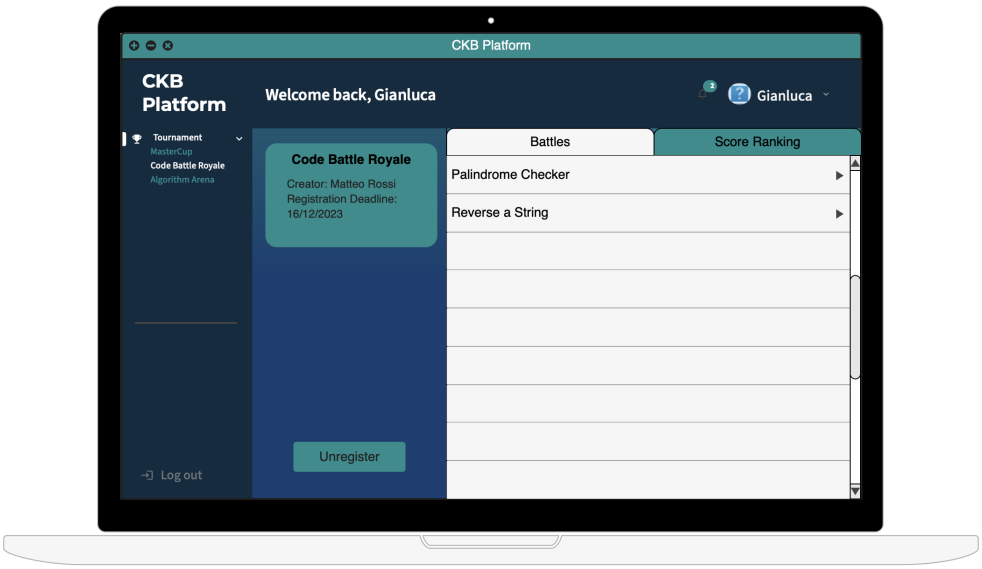

Figure 3.5: Tournament page for an enrolled Student

After completing registration, the page shown in Figure 3.5 will add a list of available battles within that tournament. In addition, the 'Register' button will be replaced by 'Unregister', allowing students to cancel their registration.

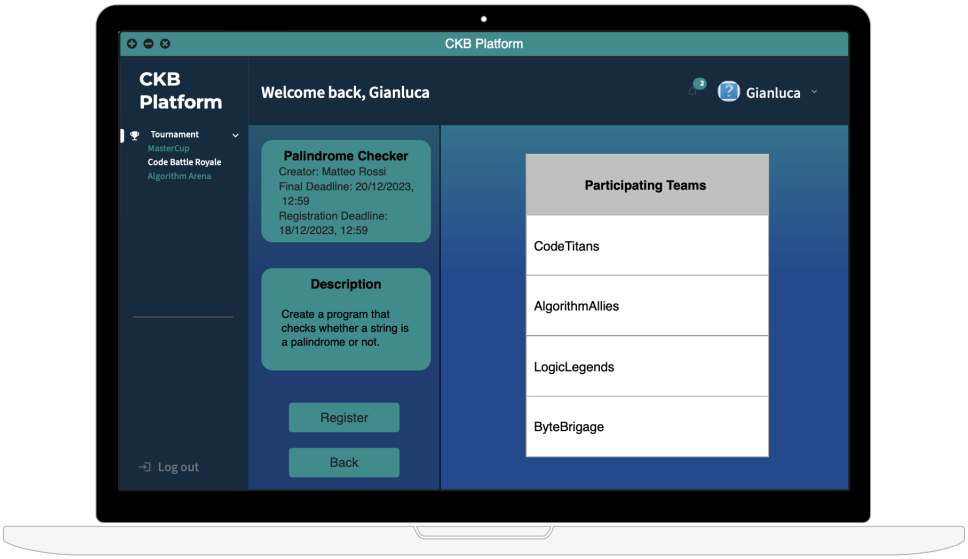

Figure 3.6: Page for a student to register for a battle

The page dedicated to a specific battle, as illustrated in Figure 3.6, has an arrangement in two distinctive sections. In the first box, basic information about the battle itself is provided: name, educator responsible for creation, registration deadline and final submission. In the second box, a brief description is provided.
In addition, there is a table listing the teams participating in the event and a 'Register' button to register for the battle.

Figure 3.7: Page for a Student to register for a battle and form a team

After pressing the 'Register' button, the interface displayed in Figure 3.7 presents the components described above. In addition, it includes a section specifying the minimum and maximum number of students allowed within a team. The page offers the possibility of entering the details required for battle registration, including the team name and a table containing the other students registered for the tournament, each associated with a checkbox. This allows students to select the other desired participants to form the team for the battle. Finally, there is a 'SignUp' button, which allows students to complete the registration process to participate in the battle.

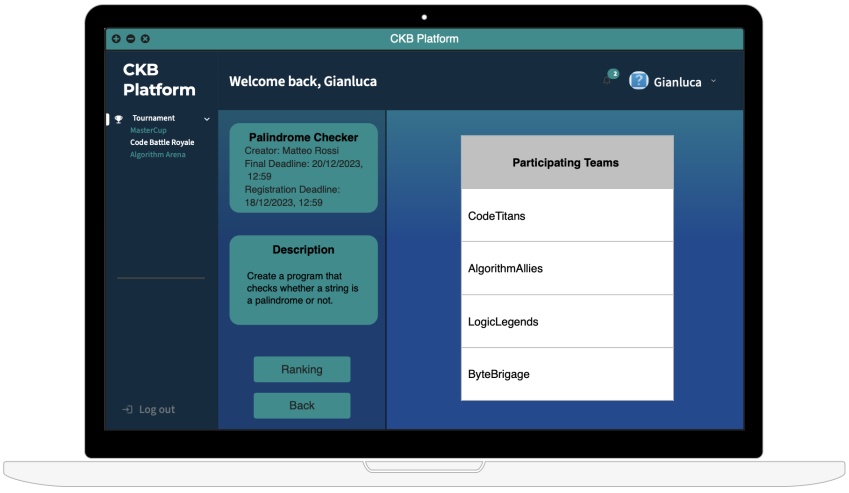

Figure 3.8: Page for a registered Student in the battle

After completing the registration process, the interface displayed in Figure 3.8 replaces the button previously labelled 'Register', as shown in Figure 3.4, with the 'Ranking' button. This button displays the current ranking of the battle.

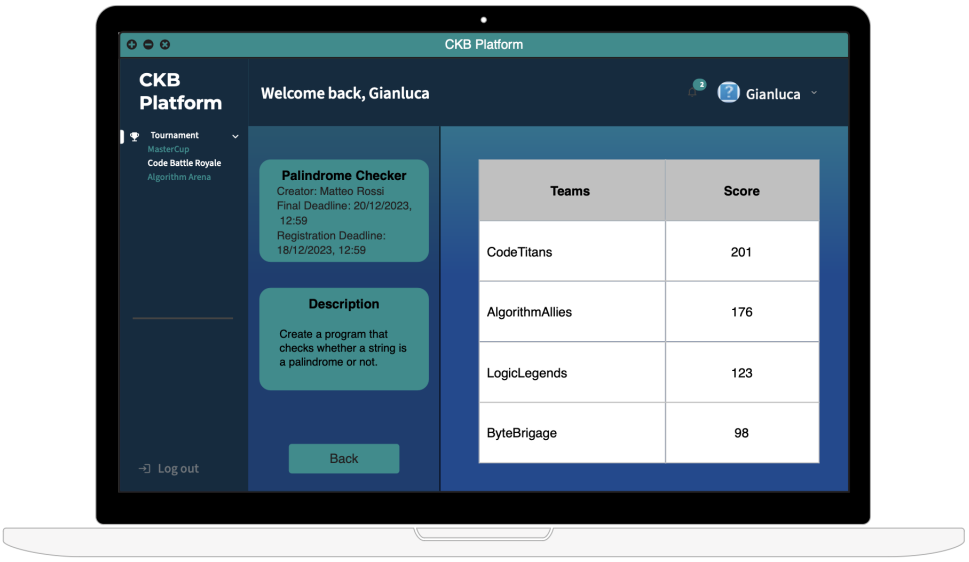

Figure 3.9: Page for the current battle ranking for a registered Student

After selecting the 'Ranking' button, the interface displayed in Figure 3.9 replaces the list of participating teams with the ranking of the teams and their current scores. In addition, a 'Back' button is available to return to the start page of the battle.
On the start page of the battle, identified in Figure 3.8, there is another 'Back' button, which takes you back to the initial interface of the tournament, as shown in Figure 3.5.

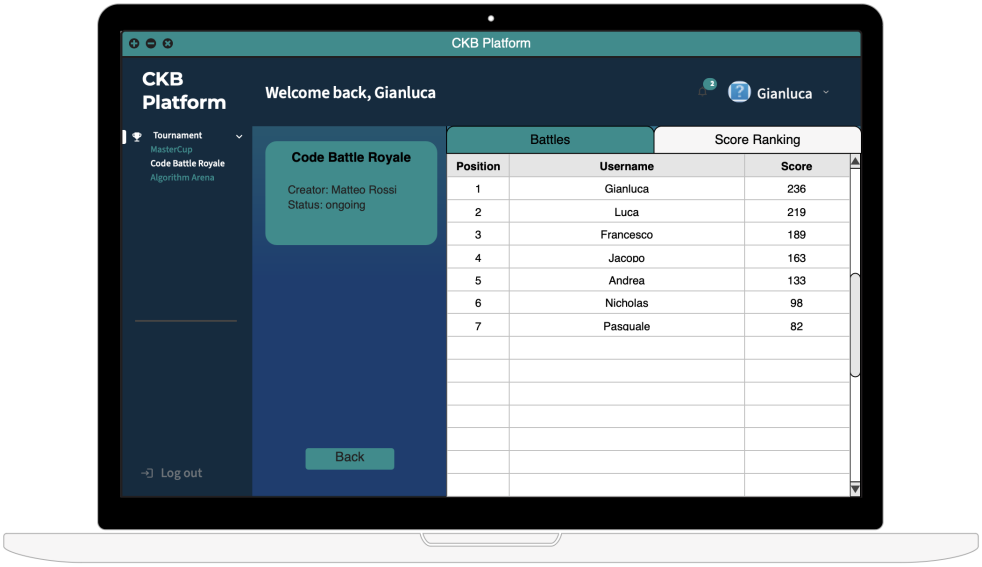

Figure 3.10: Tournament Ranking for a Registered Student

Once back on the tournament start page, you can select the "Score Ranking" view shown in Figure 3.10, which displays the current tournament ranking.

Figure 3.11: Student Notifications Page

Figure 3.11 shows the interface relating to student notifications. The interface is divided into two distinctive sections: a section on invitations to participate in other battles with other teams and a section on generic notifications.

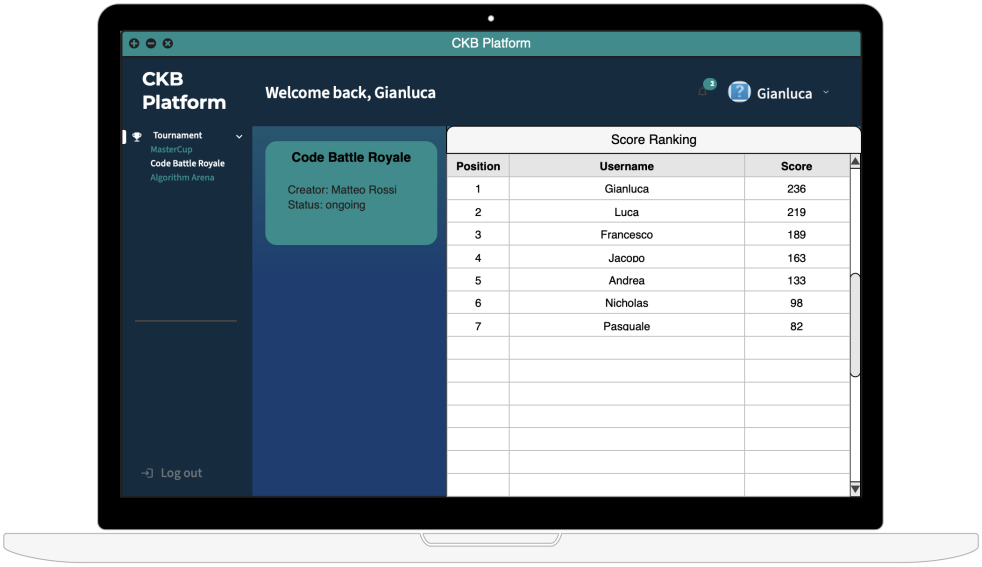

Figure 3.12: Current Tournament page for non-registered users

In addition, if the registration deadline for the tournament has expired, users accessing the tournament page, as shown in Figure 3.12, will have access to a view containing the tournament details and the updated standings.

3.2 Educator Interface

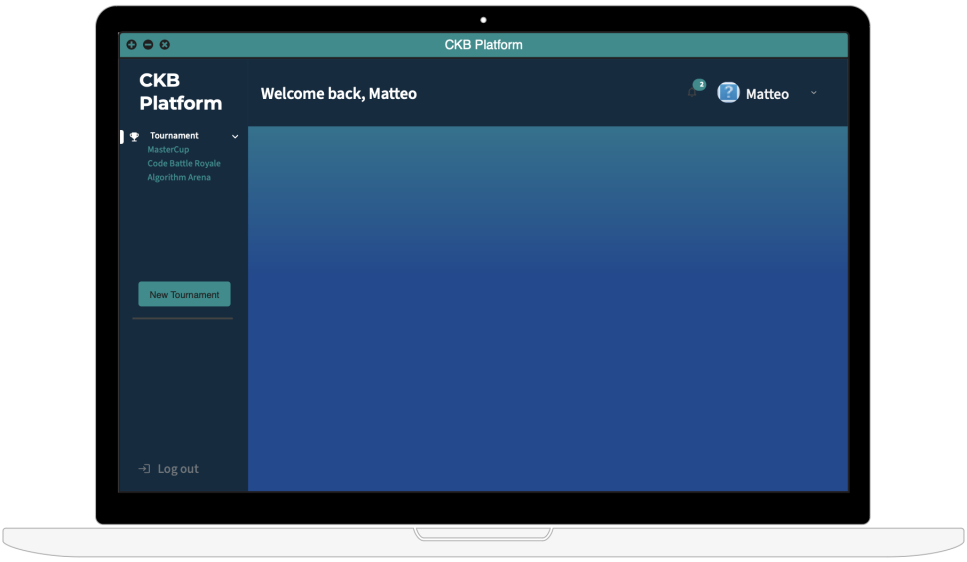

Figure 3.13: Educator Dashboard

Figure 3.13 shows the dashboard for users accessing the platform as educators. We highlight the list of tournaments available on the platform and the presence of a 'New Tournament' button to create a new tournament.

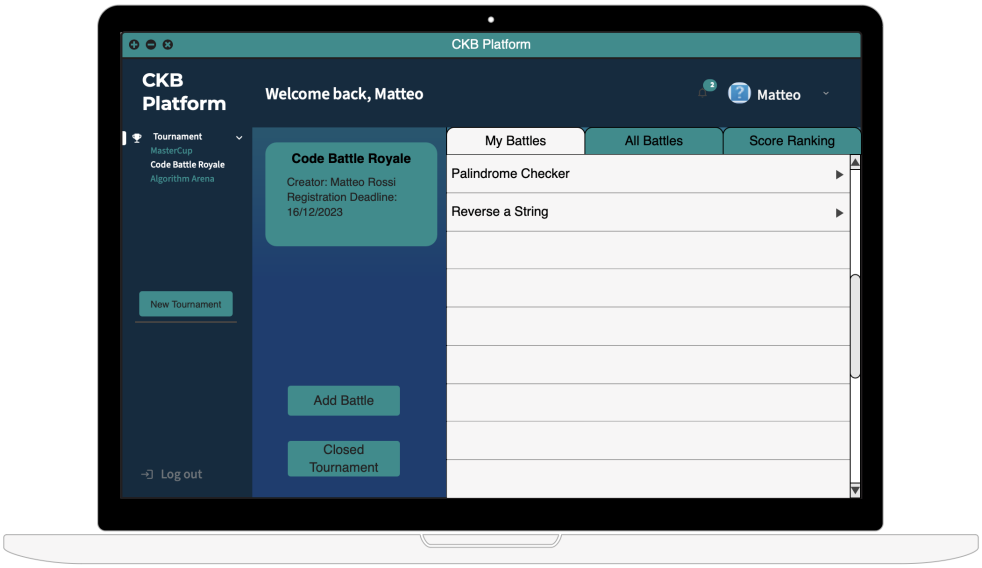

Figure 3.14: Tournament page for the Creator Educator

Figure 3.14 shows the specific page of a tournament selected in the left section. In this context, the educator, who also acts as the creator of the tournament, has the options to add battles or close the tournament (only if the battles have ended).
The page offers three distinctive sections:

- "My Battles", shown in Figure 3.14 presents the list of battles created by the educator within the tournament.

- "All Battles", shown in Figure 3.15 shows the entire list of battles in progress.
- "Score Ranking", visible in Figure 3.16 shows the ranking of the tournament.

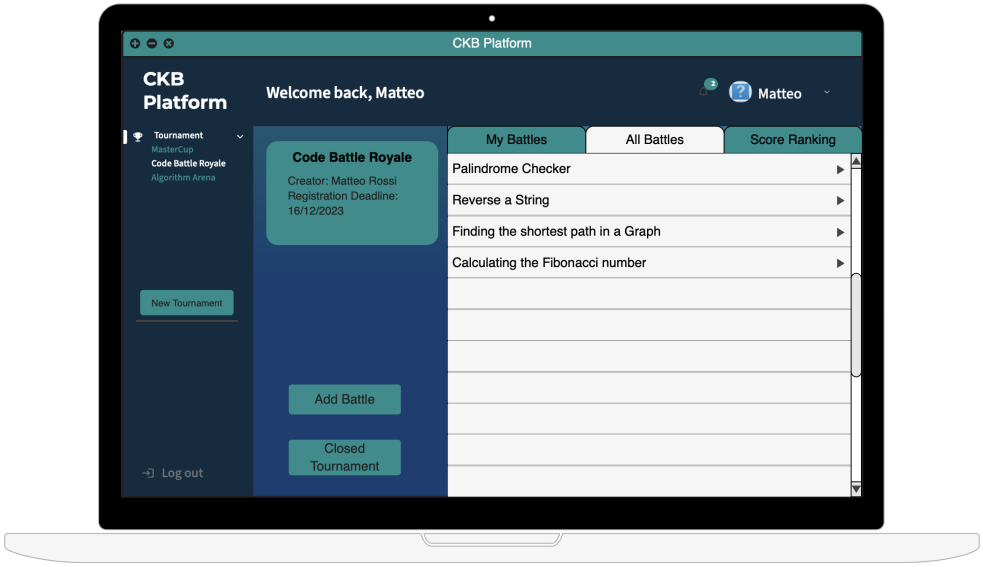

Figure 3.15: List of ongoing Battles within the Tournament

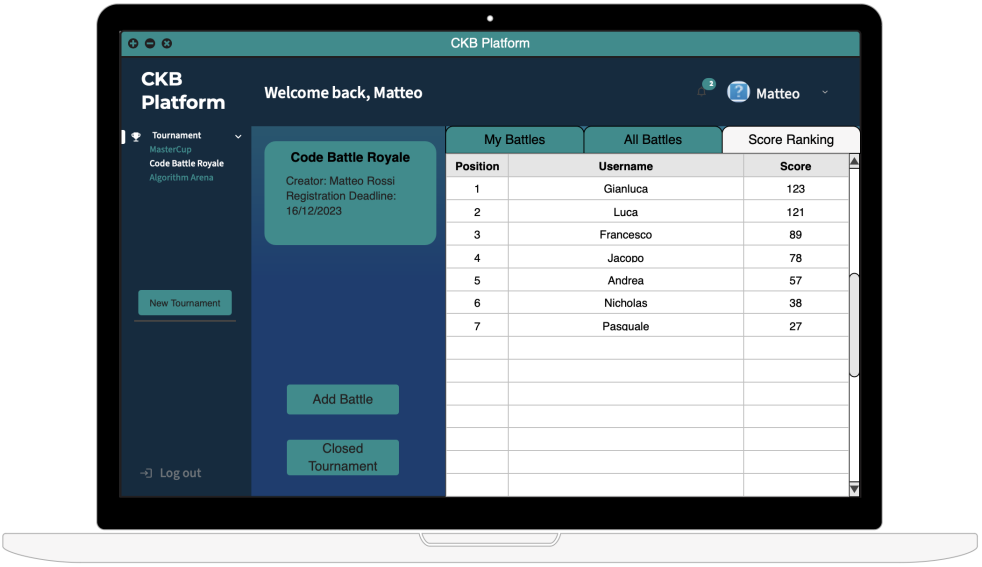

Figure 3.16: Tournament Ranking

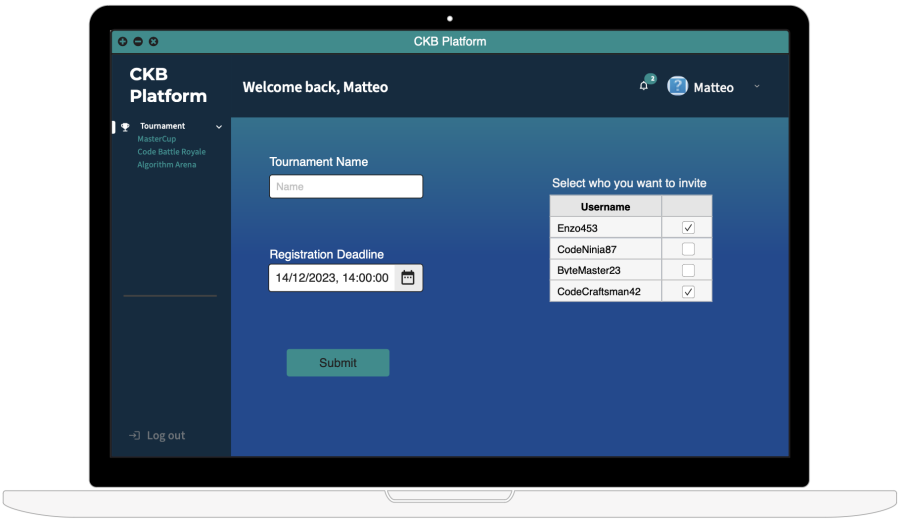

Figure 3.17: Educator creates a Tournament

The educator initiates the creation of a new tournament through the "New Tournament" option displayed in Figure 3.13. This action takes the educator to a specific page, illustrated in Figure 3.17, where he enters the name of the tournament, sets the "Registration Deadline" indicating the closing date for registrations, and selects the educators hewishes to invite, via a series of checkboxes. Finally, confirm the creation by pressing the "Submit" button.

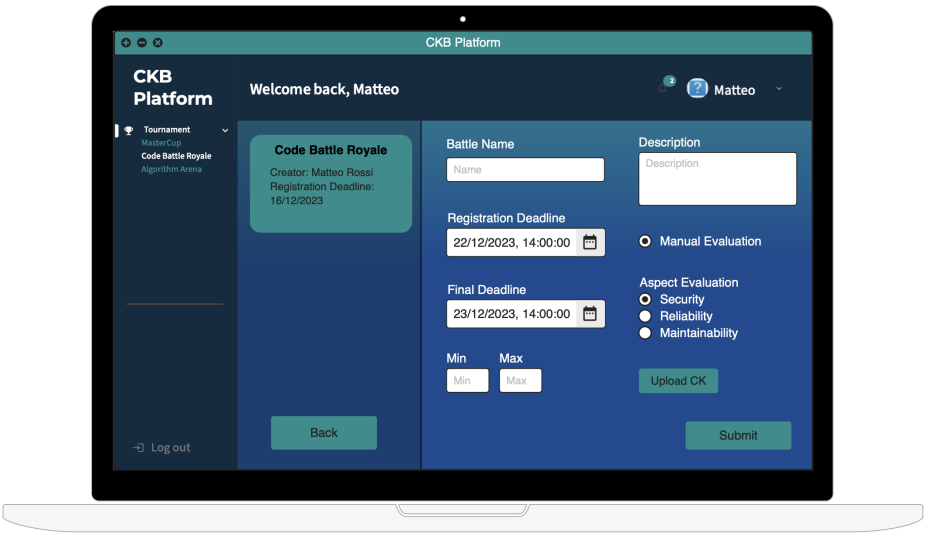

Figure 3.18: Educator creates a Battle

The educator, to add a new battle within a tournament, uses the "Add Battle" option displayed in Figure 3.14. This action takes the educator to a specific page, illustrated in Figure 3.18, where he enters the name of the battle, defines the "Registration Deadline", the "Final Deadline", a short description, defines whether or not to carry out a manual evaluation, selects the evaluation aspect and uploads the CodeKata file. Subsequently, the educator confirms the creation by pressing the "Submit" button.

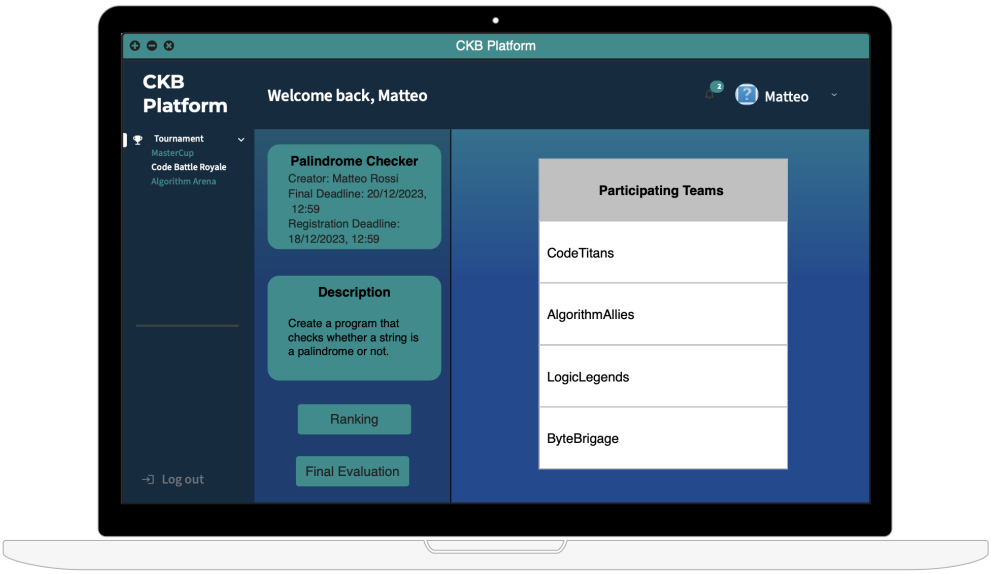

Figure 3.19: Page for an Educator at the end of a battle

Figure 3.19 shows the page dedicated to the educator who created the battle and it has come to an end. There is a 'Final Ranking' button, which allows the educator to carry out the final evaluation of the project. In addition, there is a 'Send' button to confirm the values entered.

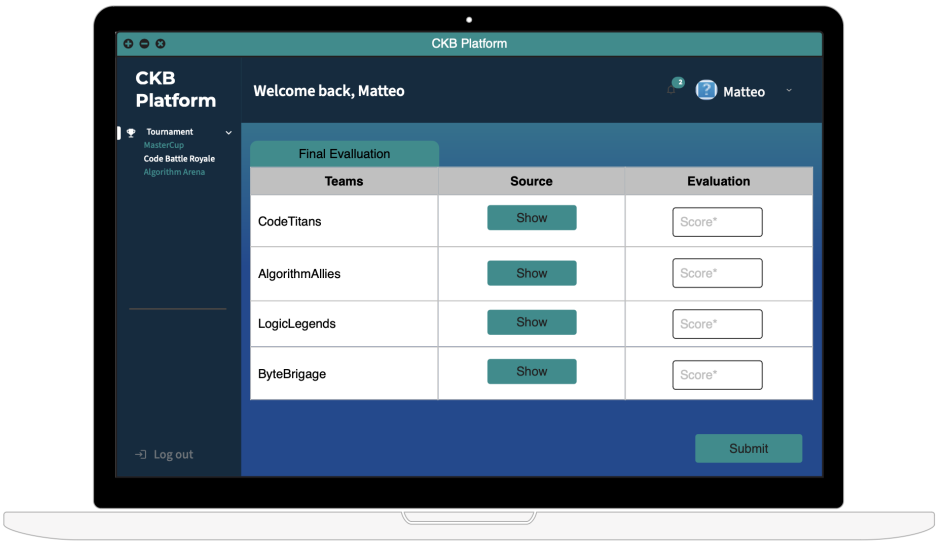

Figure 3.20: Educator Final Evaluation

Clicking on the button described above takes you to the page shown in Figure 3.20. This page shows a table in which each row corresponds to the name of the team, accompanied by a "Show" button to access the files of that team and a text box to enter the score. In addition, there is a "Back" button to return to the previous page.

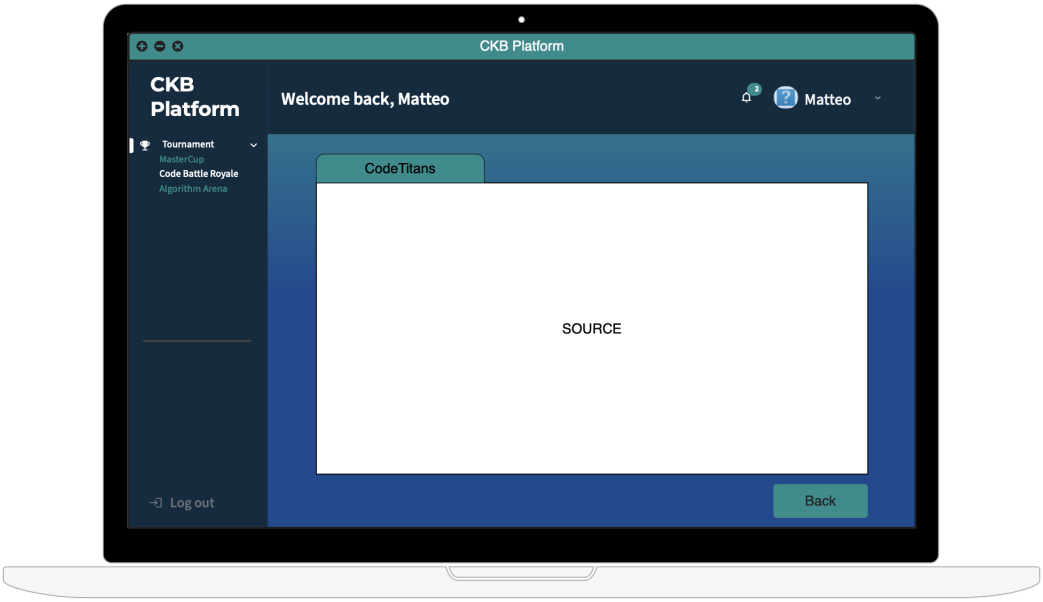

Figure 3.21: Code page of the selected team

After clicking on the 'Show' button, the page shown in Figure 3.21 is accessed, where the name of the selected team is displayed together with its code.

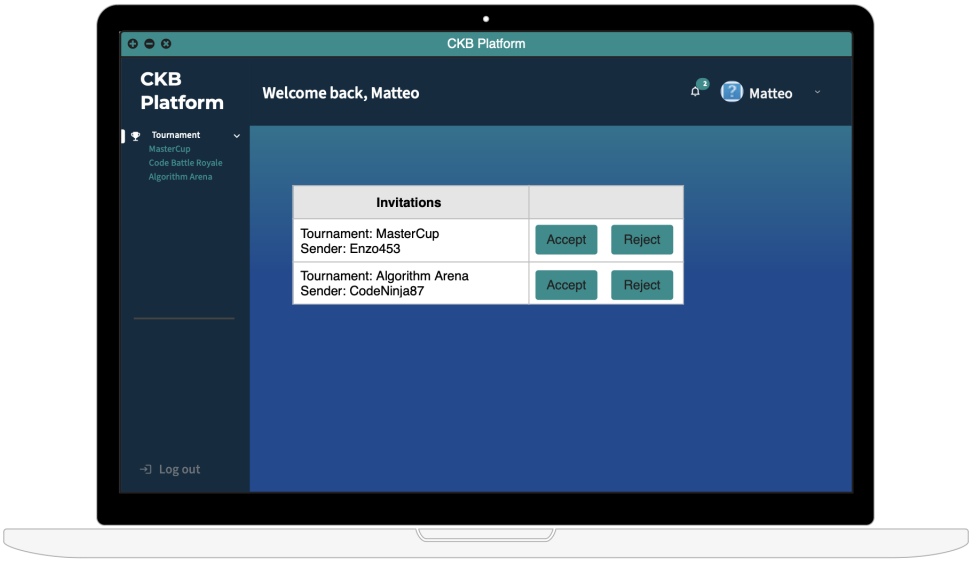

Figure 3.22: Educator Notifications Page

Figure 3.22 shows the notification section for the educator. This area shows the invitations received to participate in the creation of battles within a tournament.

4 Requirement traceability

In this section, each component of each microservice will be presented in detail, accompanied by a list of the specific requirements that each component satisfies. This insight will provide a comprehensive understanding of the functionality and responsibilities assigned to each constituent element within the microservice architecture. In certain contexts, as will be highlighted, some components are grouped together because they meet identical requirements. Documentation corresponding to these requirements is presented in a unified manner, aiming to simplify the process of understanding.

4.1 User Credentials Microservice

UserHandler component:

- [R1] The system allows the student to register by entering firstname, lastname, email, username, and password.
- [R2] The system allows registered students to log in by entering username and password.
- [R3] The system allows the educator to register by entering firstname, last-name, email, password and username.
- [R4] The system allows registered educator to log in by entering username and password.
- [R12] The system notifies all students registered on the platform to inform them of the creation of a new tournament.

UserRegistrationManager component:

- [R1] The system allows the student to register by entering firstname, lastname, email, username, and password.
- [R3] The system allows the educator to register by entering firstname, last-name, email, password and username.

UserLoginManager component:

- [R2] The system allows registered students to log in by entering username and password.
- [R4] The system allows registered educator to log in by entering username and password.

UserManager component:

- [R12] The system notifies all students registered on the platform to inform them of the creation of a new tournament.

DataManager component:

- [R1] The system allows the student to register by entering firstname, lastname, email, username, and password.
- [R2] The system allows registered students to log in by entering username and password.
- [R3] The system allows the educator to register by entering firstname, last-name, email, password and username.
- [R4] The system allows registered educator to log in by entering username and password.
- [R12] The system notifies all students registered on the platform to inform them of the creation of a new tournament.

4.2 Tournament Microservice

TournamentHandler, TournamentManager, DataManager components:

- [R5] The system allows the educator the ability to create of a tournament, allowing him to set a deadline for entries.
- [R6] The system allows, during the creation of the tournament, the educator to invite other educators to participate in the tournament.
- [R7] The system allows the educator who created the tournament, to close the tournament if all battles are completed
- [R8] The system allows students to register for the tournament by the deadline.
- [R9] The system allows students to unsubscribe from the tournament.
- [R13] The system, at the end of each battle, updates each student's score by summing the scores obtained in each completed battle in which he participated, reflecting these changes in the overall tournament ranking.
- [R14] The system when the educator closes the tournament, notifies all students in that tournament of the availability of the final ranking
- [R15] The system allows all members of the platform to see the list of ongoing tournaments.
- [R16] The system allows all members of the platform to see the ranking of a tournaments, with the relative score associated with each student participating in the tournament
- [R17] The system sends a notification to educators who have been invited by the educator who created the tournament to participate in the creation of battles within the tournament
- [R18] System allows educators to accept or decline invitations to participate in the tournament.
- [R36] The system sends a notification to all students participating in a tournament when a new battle is created.

TournamentClosurePublisher component:

- [R14] The system when the educator closes the tournament, notifies all students in that tournament of the availability of the final ranking

TournamentCaller component:

- [R7] The system allows the educator who created the tournament, to close the tournament if all battles are completed

4.3 Battle Microservice

BattleHandler, BattleManager, DataManager components:

- [R10] The system allows students enrolled in the tournament to see the list of all battles(unstarted, ongoing, and completed).
- [R11] The system allows educators to see the list of all battles(unstarted, ongoing, and completed) in that tournament.
- [R13] The system, at the end of each battle, updates each student's score by summing the scores obtained in each completed battle in which he participated, reflecting these changes in the overall tournament ranking.
- [R19] The system allows all educators participating in the tournament to create battles by uploading the CodeKata and specifying the description, entry deadline, final project delivery deadline, minimum and maximum number of students in each team, if desired manual evaluation, and qualitative aspects of the source to be evaluated.
- [R20] The system when the battle is over allows the educator who created the battle to see each team's final project.
- [R21] The system allows the educator to evaluate each team's final projects.
- [R22] The system allows students registered for the tournament to see all the specifics of a battle.
- [R23] The system allows students participating in the tournament to register for the battle by the deadline.
- [R24] The system allows students participating in the battle to create teams by inviting other students participating in the same tournament.
- [R25] The system when the battle enrollment expires must create the GitHub repository with the CodeKata in it.
- [R26] The system when the battle enrollment expires sends the GitHub repository link to all members of each team and the instructions to fork the repository and set up an automated workflow.
- [R27] The system must retrieve the source from the team repository after each push by the team.
- [R28] The system assigns a integer score from 0 to 100 to the project retrieved from the GitHub repository, according to the following aspects: functional aspects (number of test cases passed), timeliness (difference between the last team commit and the battle registration deadline), and source quality level, the latter evaluated by a third-party tool Sonarqube.
- [R29] The system allows students participating in the battle to see each team's current score.

- [R30] The system allows students participating in the battle to see the current ranking of the battle.
- [R31] The system allows the battle educator to see the current ranking.
- [R32] The system allows the battle educator to see the score of each team.
- [R33] The system, at the end of the battle, allows students participating in tournament to see the final ranking.
- [R34] The system, at the end of the battle, allows educators participating in tournament to see the final ranking
- [R35] The system, at the end of the battle, notifies all students participating in the battle of the availability of the final ranking.
- [R36] The system sends a notification to all students participating in a tournament when a new battle is created.
- [R37] The system sends a notification to students who have been invited by another student to join and form a team.
- [R38] The system offers the possibility for students to accept or reject invitations received from other students to form a team.

TeamManager component:

- [R20] The system when the battle is over allows the educator who created the battle to see each team's final project.
- [R21] The system allows the educator to evaluate each team's final projects.
- [R24] The system allows students participating in the battle to create teams by inviting other students participating in the same tournament.
- [R37] The system sends a notification to students who have been invited by another student to join and form a team.
- [R38] The system offers the possibility for students to accept or reject invitations received from other students to form a team.

InviteTeamPublisher component:

- [R37] The system sends a notification to students who have been invited by another student to join and form a team.

Evaluator and SonarQubeResponsible components:

- [R19] The system allows all educators participating in the tournament to create battles by uploading the CodeKata and specifying the description, entry deadline, final project delivery deadline, minimum and maximum number of students in each team, if desired manual evaluation, and qualitative aspects of the source to be evaluated

- [R28] The system assigns a integer score from 0 to 100 to the project retrieved from the GitHub repository, according to the following aspects: functional aspects (number of test cases passed), timeliness (difference between the last team commit and the battle registration deadline), and source quality level, the latter evaluated by a third-party tool Sonarqube.

GitHubResponsible component:

- [R25] The system when the battle enrollment expires must create the GitHub repository with the CodeKata in it.
- [R26] The system when the battle enrollment expires sends the GitHub repository link to all members of each team and the instructions to fork the repository and set up an automated workflow.
- [R27] The system must retrieve the source from the team repository after each push by the team.

GitHubLinkPublisher component:

- [R26] The system when the battle enrollment expires sends the GitHub repository link to all members of each team and the instructions to fork the repository and set up an automated workflow.

BattleClosurePublisher component:

- [R35] The system, at the end of the battle, notifies all students participating in the battle of the availability of the final ranking.

FinalScoreBattlePublisher component:

- [R13] The system, at the end of each battle, updates each student's score by summing the scores obtained in each completed battle in which he participated, reflecting these changes in the overall tournament ranking.

4.4 Notifications Microservice

NotificationsHandler, NotificationsManager,NotificationsCaller and DataManager components:

- [R12] The system notifies all students registered on the platform to inform them of the creation of a new tournament.
- [R14] The system when the educator closes the tournament, notifies all students in that tournament of the availability of the final ranking
- [R17] The system sends a notification to educators who have been invited by the educator who created the tournament to participate in the creation of battles within the tournament
- [R26] The system when the battle enrollment expires sends the GitHub repository link to all members of each team and the instructions to fork the repository and set up an automated workflow.
- [R35] The system, at the end of the battle, notifies all students participating in the battle of the availability of the final ranking.
- [R36] The system sends a notification to all students participating in a tournament when a new battle is created.
- [R37] The system sends a notification to students who have been invited by another student to join and form a team.

5 Implementation, Integration and Test Plane

5.1 Overview

In the last section, we will explore the practical implementation of the system, focusing on component integration and the approach to validation and verification. Dijkstra wisely argued that "testing can reveal bugs, but not prove their absence." Therefore, testing will be aimed at finding as many bugs as possible before release. Finally, we stress the crucial importance of clear documentation and comments in the code, making it easier to understand and maintain the system over time.

5.2 Implementation plan

In the context of a microservice architecture, the deployment strategy is closely related to the functional division of the application into distinct units, commonly known as microservices. After a thorough analysis of the functional requirements and domain specifications, we identify domain boundaries that delineate the responsibilities of each microservice.
We then assign each microservice to a dedicated team consisting of developers, testers, and other specialists. Each team assumes full responsibility for the development and maintenance of its microservice, operating in isolation and independently to optimize efficiency and skill specialization.
In addition, it is essential to establish a dedicated team responsible for developing the interface and all its functionality. Each team develops its microservice in parallel, followed by an isolated testing phase. The preliminary configuration of the Gateway API is an essential prerequisite for the subsequent global system test, during which the interaction between the microservices is verified. This phase is crucial to ensure synergistic collaboration among the microservices and effective response to user interface requests.

5.3 Component Integration and Testing

At this stage, each team proceeds with the autonomous and parallel development of its own microservice. Initially, the focus is on building the essential components for autonomous microservice operation, initially excluding those dedicated to asynchronous communication between microservices or notifications to the interface.

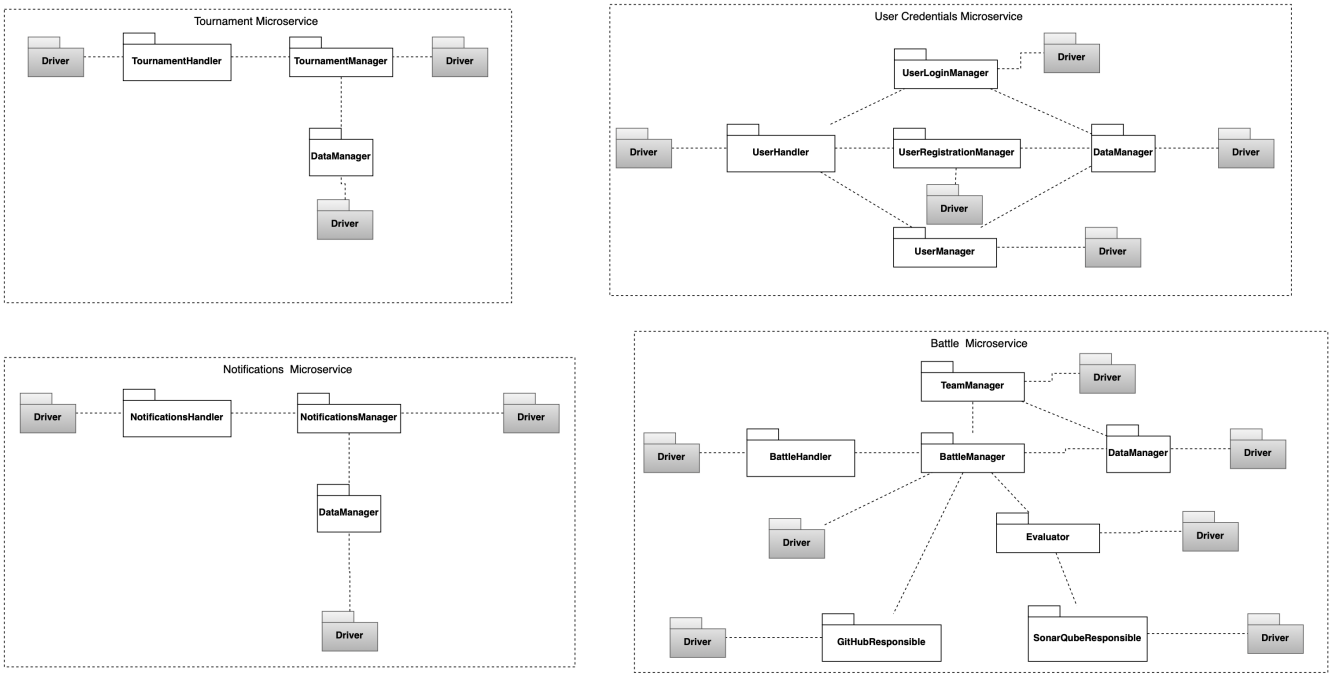

Figure 5.1: Microservice development without external communication

Next, the components that enable communication between microservices and the transmission of notifications to the interface are developed. These components are tested to assess their effectiveness, as well as to verify the overall interaction between the various microservices.

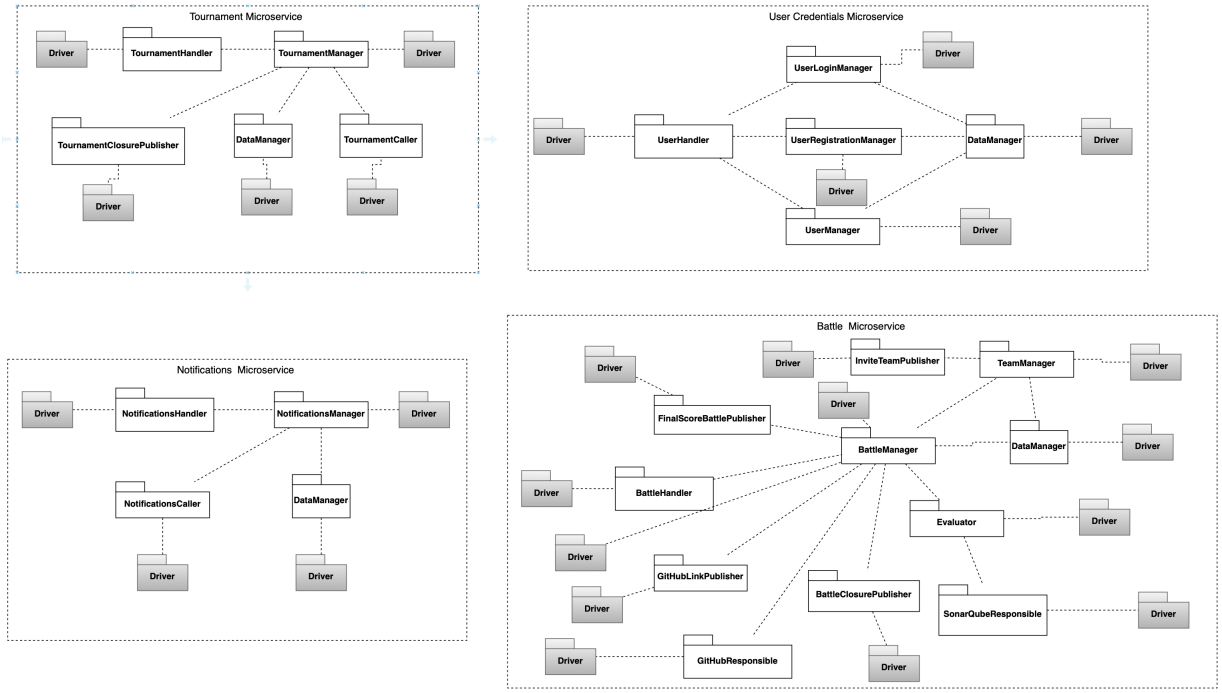

Figure 5.2: Microservice development complete

5.4 System testing

Upon the full integration of the system, it becomes imperative to subject it to comprehensive testing as a unified entity to validate the fulfillment of both functional and non-functional requirements. System testing serves as the instrumental approach for such validation. Furthermore, it is essential to ensure that the testing environment closely mirrors the production environment to facilitate a realistic assessment.
System testing encompasses various types, each tailored to address distinct aspects of the system's performance and compliance. These types include:

- **Functional testing** consists in verifying whether the system satisfies all the requirements specified in the relative Requirement Analysis and Specification Document (RASD). Furthermore, during this phase, it might be possible to think of new features that might improve the user experience.
- **Performance testing** is a type of testing that focuses on evaluating the performance of a system, measuring its ability to respond effectively to a given workload or maintain acceptable performance under various conditions. This type of testing is critical to identify any performance problems and to optimize the system before release.
- **Usability testing** is a type of testing designed to evaluate how user-friendly a system or application is and provides a satisfactory user experience for end users. This type of testing is critical to identifying usability problems, improving the user interface, and optimizing the overall user experience.
- **Load testing** is a testing practice that simulates the expected or maximum operational load on an application, system or website to evaluate its performance and identify any limitations or problems. The main objective of load testing is to measure how well the system responds to a significant amount of concurrent users, transactions or requests, checking whether it can handle the expected load without degrading performance.
- **Stress testing** is a form of performance testing that involves putting a system or application under an excessive or stressful load to evaluate its robustness, stability, and ability to maintain acceptable performance even under conditions of intensive use or beyond expected limits.

5.5 Additional specifications on testing

When completing the system, it is crucial to perform extensive testing, focusing on workloads to identify possible network saturation issues. It also carefully evaluates the effectiveness of messaging between different microservices, verifies the proper configuration of the Gateway API, and pays special attention to security, especially with regard to the User Credentials microservice.
During development, it is essential to regularly obtain feedback from users and stakeholders, especially those who requested the system and end users. This iteration occurs each time a feature is implemented. A detailed description of the functionality is provided for verification of expectations, followed by distribution of alpha versions to gather feedback on any issues and measure overall satisfaction.

6 Time Spent

Raffaele Russo

Chapter	Effort (in hours)
1	8
2	25
3	15
4	8
5	9

Biagio Marra

Chapter	Effort (in hours)
1	8
2	25
3	12
4	10
5	10

7 References

- Diagrams made with: draw.io
- Runtime view made with: StarUML
- Mockups made with: moqups.com
- fig[2.22]: https://middleware.io/blog/microservices-architecture/