# CodeKataBattle

Diegoli Tommaso, Tagliani Fabio

06/01/2024

# Contents

# 1 Introduction

## 1.1 Purpose

An essential component of every programming course involves ensuring that theoretical explanations are complemented by rigorous practical training, ensuring a thorough comprehension by students. However, a challenge arises, especially in larger classes, where educators find it difficult to individually monitor and provide feedback on each assignment. The CodeKataBattle platform aims to address this issue by providing an interactive space where students can actively practice and assess their programming skills. Through automated evaluation, students can compare their results with their peers, fostering a collaborative learning environment. The platform consists of tournaments in which one or more educators can insert battles where teams of students can compete. The platform then provides a rank for each student in the tournament and a rank for the teams in a battle.

This document is here to help the development team build the system giving information about what architectural style are to be used and how to implement the different components of which the system is composed.
Additionally, interface's mockups are given and also a plan on how to test the implementation.

## 1.2 Scope

CodeKataBattle (CKB) is an innovative platform designed specifically for educators seeking to enhance their students' programming skills in a dynamic and engaging way. Because of that it should permits a seamless transition from the traditional way of teaching for the main stakeholders that will be impacted: educators and students.
In particular it should take into accounts educators and students following computing courses.

### 1.2.1 Architectural Choices

The implementation is one of a classic client-server system following the three-tier architecture. This is the most used approach to build a web-application in particular if it's used by very different devices.
More on this can be seen from section 2 onward.

## 1.3 Definitions, Acronyms, Abbreviations

- CKB: CodeKataBattle
- SSO: Single Sign-On
- RASD: Requirements Analysis and Specification Document
- DBMS: Database Management System
- API: Application Programming Interface

## 1.4 Revision History

- Version 1.0 (January 6th 2024 : first version);

## 1.5 Reference Documents

- "Assignment RDD AY 2023-2024.pdf".
- RASD: "RASD.pdf"

## 1.6 Document Structure

Other than this first chapter, this document contains 4 main chapters and 2 conclusive ones:

- **Architectural Design:** In this section a comprehensive overview of the system's architecture is given, encompassing a high-level view of its elements, then focusing on the component it's composed of, also showing how the different functionalities of the platform are implemented by those components. The section concludes with the specification of additional design choices.
- **User Interface Design:** This chapter is dedicated to the design of the user interface for the application, emphasizing the interaction between the system and its users. It introduces two distinct interfaces: one for the student of the CKB application and another for the educators. Mockups of the application's user interfaces are presented, along with links between them to facilitate an understanding of the flow between different screens.
- **Requirements Traceability:** In this section, each component is linked to the specific requirements it needs to fulfill. A single component might have to satisfy multiple requirements. It establishes the connections between the requirements outlined in the RASD and the components detailed in the initial chapter. This linkage serves as evidence that the design decisions align with the requirements, ensuring that the developed system can successfully achieve its goals.
- **Implementation, Integration, and Test Plan:** This section revolves around outlining the implementation plan for various components, emphasizing the significance and challenges associated with different functionalities of the system. A crucial aspect covered is the integration of diverse components and the strategy for testing to ensure smooth integration. The Implementation, Integration, and Test Plan details the process that developers must adhere to for implementing, integrating, and testing to ensure the accurate and proper functioning of the system.
- **Effort Spent:** This section details the time and resources invested in the project
- **References:** In the last section there are the references to the resources used in this document.

# 2 Architectural Design

## 2.1 Overview

The system will be structured on a three-tiered client-server architecture, comprising a presentation layer, a logic layer, and a data layer. In particular the data layer is composed by two Data Bases: one used to store only login information and the other focused on tournaments and battles data, can be motivated by several factors:

- **Increase Security:** Separating login information into a dedicated database adds an extra layer of security. User credentials, which are critical and sensitive data, can be more effectively isolated from other non-sensitive data related to tournaments and battles. This segregation reduces the potential impact of security breaches.
- **Scalability:** By having a separate database for login information, the system can scale login-related operations independently.

As the platform will be used on different user machines, such as students' notebooks and educators' desktops, the three-tiered architecture makes the client application as thin as possible (thin-client). Opting for a three-tier architecture over the traditional two-tier client-server model (which separates data and application logic) is motivated by several advantages:

- **Platform Flexibility and Scalability:** Different tiers can run on distinct machines, allowing us to choose the most fitting platforms for their execution. The components responsible for presentation, logic, and data can be deployed on separate physical or virtual machines. This allows to customize each tier independently to best suit its functionality, performance requirements, and resource needs.
- **Enhanced Reliability:** Improved system reliability is achieved because if one tier fails, the other tiers can continue to operate.
- **Security Benefits:** Direct interaction between the data layer and presentation layer is avoided, enhancing security. Additional security measures, such as firewalls between the application tier machines and the data machines, can be implemented.
- **Data Accessibility:** The separation allows for easy retrieval of data for various functionalities, facilitating future extensions of CKB platform and supporting data analysis tasks (e.g., studying students behaviour).

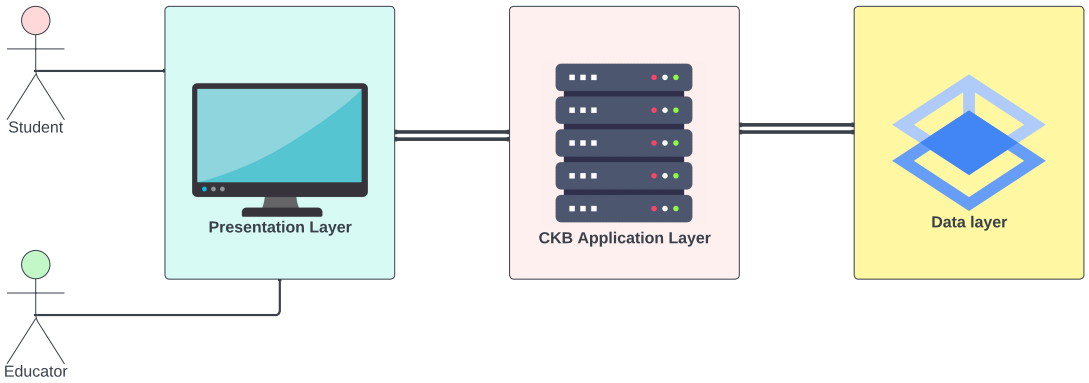


Figure 1: 3-Tier Architecture

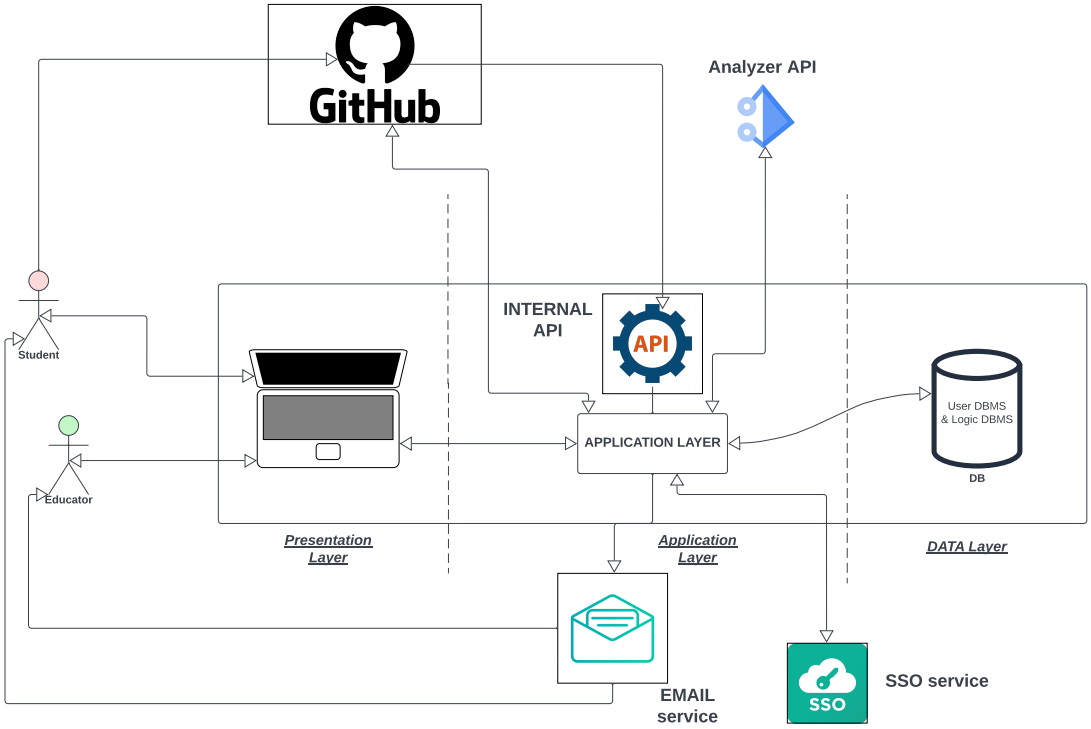


Figure 2: 3-Tier Architecture High Level

## 2.2 Component View

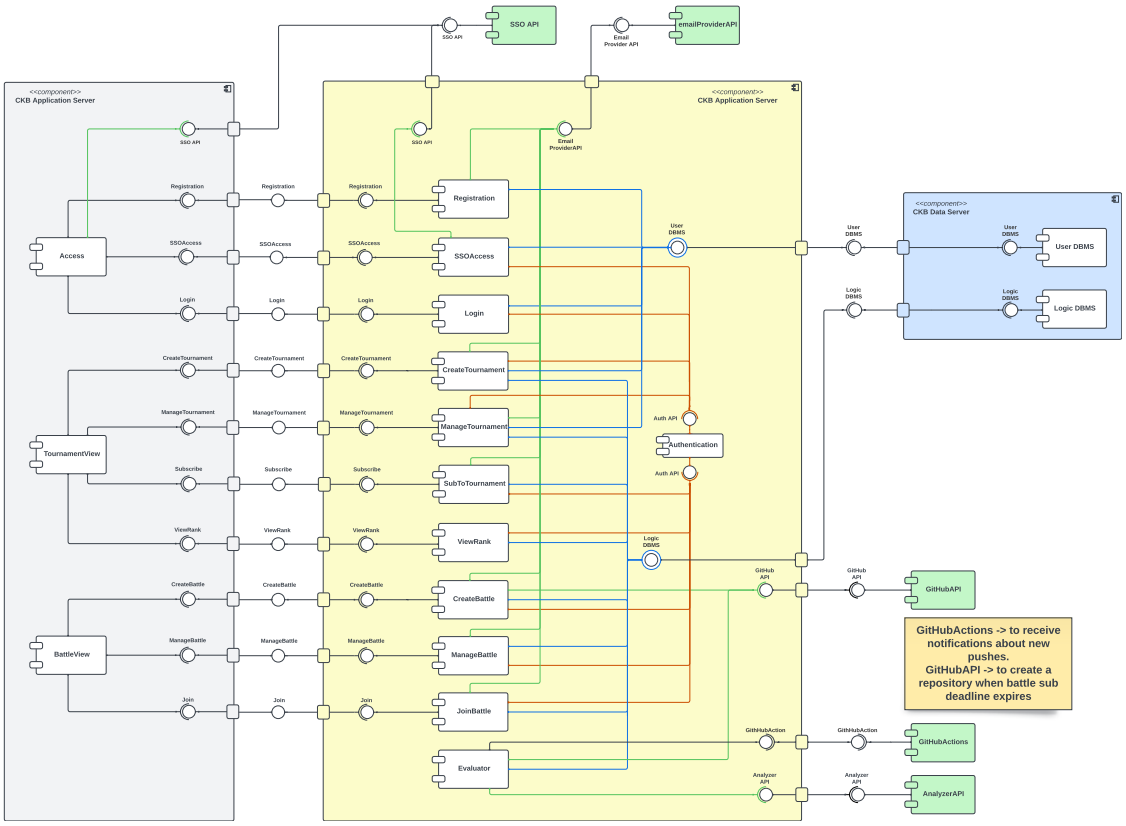


Figure 3: CKB UML Component Diagram

The diagram shows all the components needed to implement all the functionalities of the system

Other than the external services that are connected through the relevant APIs there are three main components.

**CKB Client App:** it includes all the subcomponents needed to show the interface of the web app and it redirects all the requests to the *Application Server* through its methods.
The main subcomponents of which its composed are:

- **Access:** This component provides registered/unregistered users the interfaces required to access the proper functions of the platform. In particular it provides a registration page for the classic registration with mail/pw, a page to login via mail/pw and the interface to complete the access to CKB via the external SSO service implemented.
- **TournamentView:** This component provides users the interfaces required to view and manage tournaments. In particular it is used to show the tournaments' list, a tournament's page,

the page to create a tournament, a tournament's management page and the page to invite other educators to the tournament.

- **BattleView:** This component provides users the interfaces required to view and manage battles. In particular it's used to show the battles' list, a battle's page, the page to create a battle, a battle's management page and the page join battles.

**CKB Application Server:** this component handles all the incoming requests and basically all the logic of the system, it also communicates with external services. It is divided in the following subcomponents:

- **Registration:** This component handles the registration requests. It checks the uniqueness of both the email and GitHub entered and in case of a student it directly register the user by saving the credentials in the *User DBMS*, while for an educator it sends the info to the platform's staff via an external email provider for them to review it.

- **SSOAccess:** This component handles the accesses through an external SSO service. With the token received it *SSOAccess* retrieves the user's infos. Then the component checks with the *User DB* if the user is new, in this case it requires the user to input the GitHub nickname and register the user's info to the *User DBMS*. On the other hand, if the users exists the component request the *Authentication* component to auth the user with its role.

- **Login:** This component handles the authentication of the user. It verifies the credentials with the *User DBMS* and if the infos match it authenticates the user with its role via the *Authentication* component.

- **Authentication:** This component is always contacted by every other component's to check if a request is allowed. It keeps the list of logged in users with their roles and verifies with the relevant DBMSs which are the allowed operations for each user.

- **CreateTournament:** This component handles the creation of a tournament. As part of its functionality, the component includes a check to ensure the uniqueness of the tournament's name. The component then relies on an external email provider to notify students when a new tournament is created.

- **ManageTournament:** This component handles all the possible requests to manage a tournament. Main functions it's responsible for are the deletion of the tournament, the removal of a subscribed student and the insertion of new allowed educators.

- **CreateBattle:** This component handles the requests to create a battle. As part of its functionality, the component includes a check to ensure the uniqueness of the battle's name. In particular when a new battle is created the application server interacts with the **emailProviderAPI** to send the notification to the relevant students about a battle's creation. It is also responsible to set timers and handle the creation of the GitHub repository, the sending of the notification about the availability of the final battle's rank and the opening of the consolidation phase for the reviewer if it's required.

- **ManageBattle:** This component handles the requests manage a battle. These include the update of some battle's information like the registration deadline, submission deadline, minimum/maximum number of team's member. It also manage the requests to cancel a battle or remove a team from it. Eventually it's also responsible for the request to manually evaluate teams' codes for the battles that expect that in the settings.

- **ViewRank:** This component handles the requests to check rankings of the different tournaments. It also lets subscribed users and educators to check the ranks of the battles inside a tournament.

- **SubToTournament:** This component handles the requests of subscription of students to tournaments. It does some checks to avoid double subscriptions and late ones.

- **JoinBattle:** This component handles all the different requests to join a battle. It handles both the creation of a team for a battle with all the needed checks and modes (solo team or team with code) and also the joining of a user to a team via an invite code.

- **Evaluator:** This component is responsible to evaluate and update the scores of teams' codes. It keeps checking if new data is available coming from students' that set a GitHub actions to send new pushes' alerts.
  If a new push has been done and the battle's submission period isn't over yet then it retrieves the code with a pull from GitHub and start analyzing it and then assigns a score to the team. Firstly it runs the code locally on the test cases of the battle. It gives a score between 0-100 based on how many test cases the code managed to successfully passes (the higher amount, the higher score).
  Then it measure the time between the registration deadline and the push. It gives again a score between 0-100 where a lower time gives an higher score.
  Eventually it runs a series of tests with static analysis tools to assess the quality of the code based on the aspects that are selected in the battle's setting (between security, reliability and maintainability, etc). See Figure 13 for better understanding on how this is implemented. Here again the higher quality, the better score.
  The resulting score is a weighted average of the three scores (giving the 1st and last score a weight of 2/5 and 1/5 to the timeliness).

**CKB Data Server:** It includes the User and Logic DBMS that stores the users' credentials and the data needed for the logic of the platform. It provides different methods to manage and access the information stored.

## 2.3 Deployment View

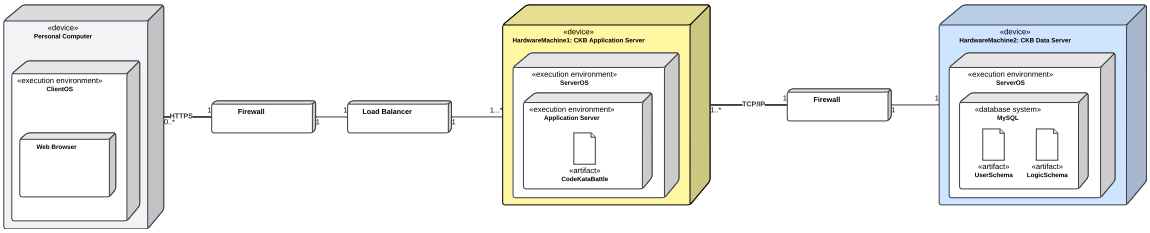


Figure 4: Deployment View

This diagram shows how the system is implemented, showing the physical machines, what's running on each of them and what protocols are used to let them communicate.
It clearly follows the three-tiered architecture as described before.

In the Figure 4, the presentation tier (in gray) shows devices running the platform on a web browser for both students and educators.

There is a single data server running both the User and Logic DBMS that store respectively the user credentials and all the data needed to run the system. Access to these servers is restricted to the machines in the application logic tier, facilitated through firewalls to uphold data protection as between Presentation layer and Application layer. The choice is done based on cost-efficiency. It is also possible to use different data server for each database in order to have better isolation and being able to better scale the system, at this time it's not needed. Also, full backups of them are to be performed at regular time intervals in order to ensure to always have a replica in case of necessity.

Aligned with the guidelines outlined in the RASD document to ensure high availability and reliability, redundancy is implemented. Consequently, multiple replicas of the application server have to be installed, complemented by a load balancer responsible for distributing requests across these machines.

## 2.4 Runtime View

In this section it's represented how the components described before interact when the different functionalities of the platform are performed. Only the most complex interactions are shown for each similar flow, to avoid too many repetitions.

To access the User/Logic DBMS the step to go through the Data Server is not shown for simplicity but should be done exactly like it's done with Application Server in all the flows.

It's important to notice that for all the functionalities (except registration/access ones) it's supposed that the user is already logged in. This is crucial for the authentication component to be able to distinguish requests coming from users of different roles (students/educators). Also it will avoid all the requests coming from users without an active session generated from the *authWithRole()* method (this check is not shown but it's always performed).

The first action performed from the user on the website is not shown but it's trivial to derive it from the method used.

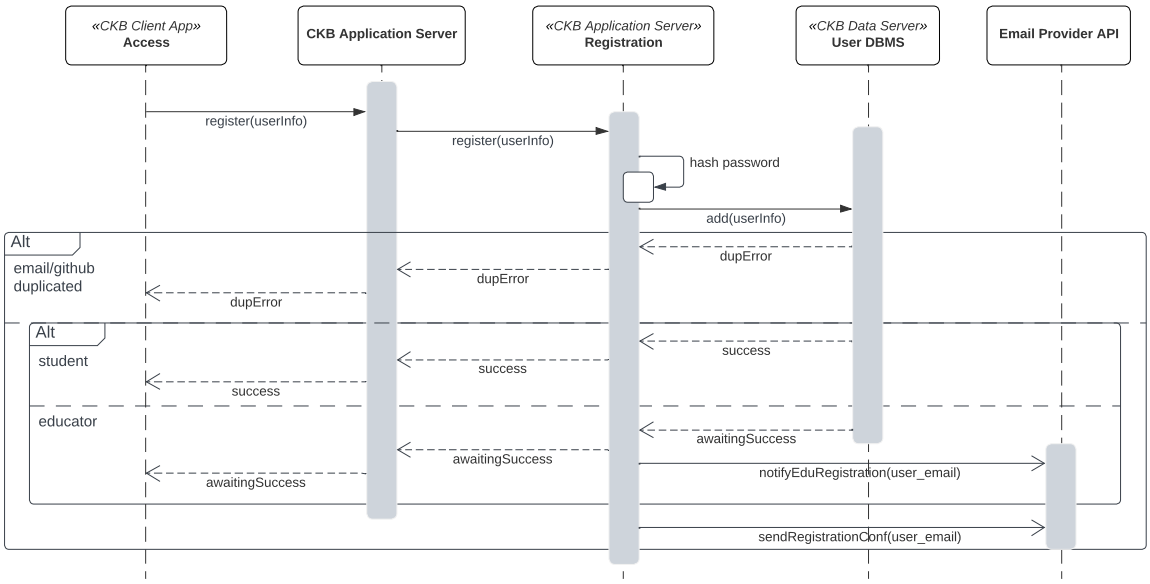


Figure 5: Student/educator registers to CKB

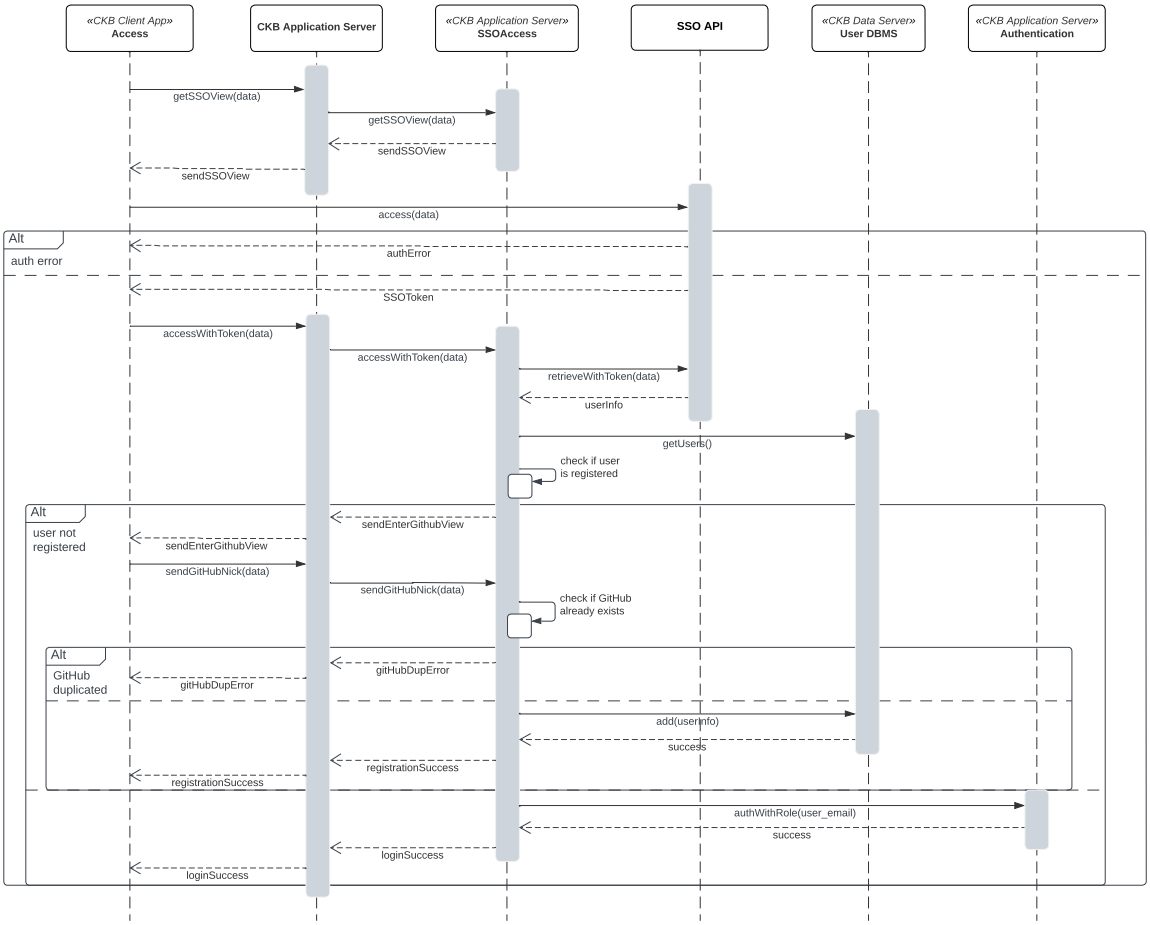


Figure 6: Student/educator registers/logins to CKB using an external SSO service

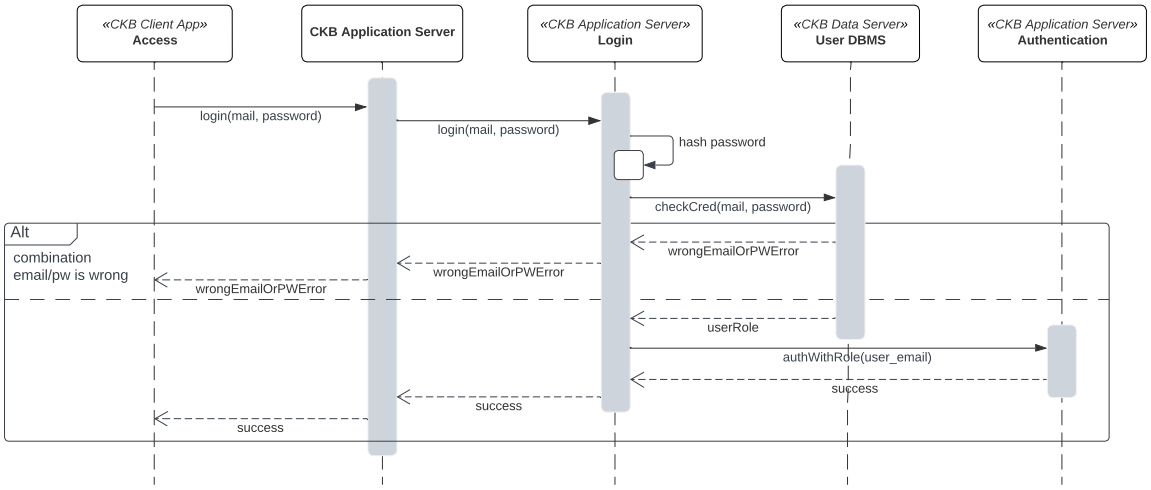


Figure 7: Registered user logins to CKB

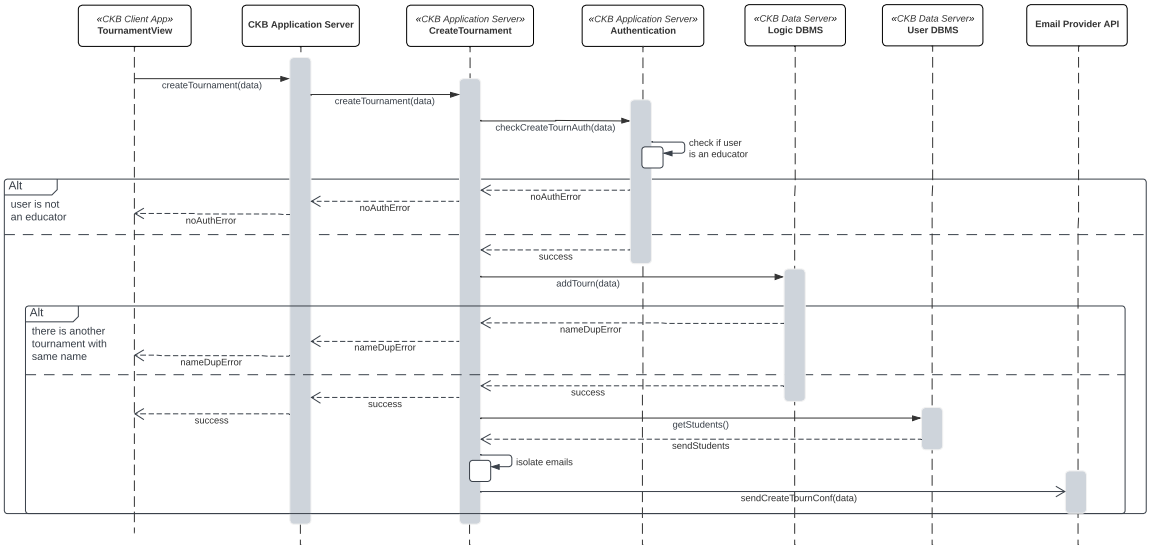


Figure 8: Registered user creates a tournament

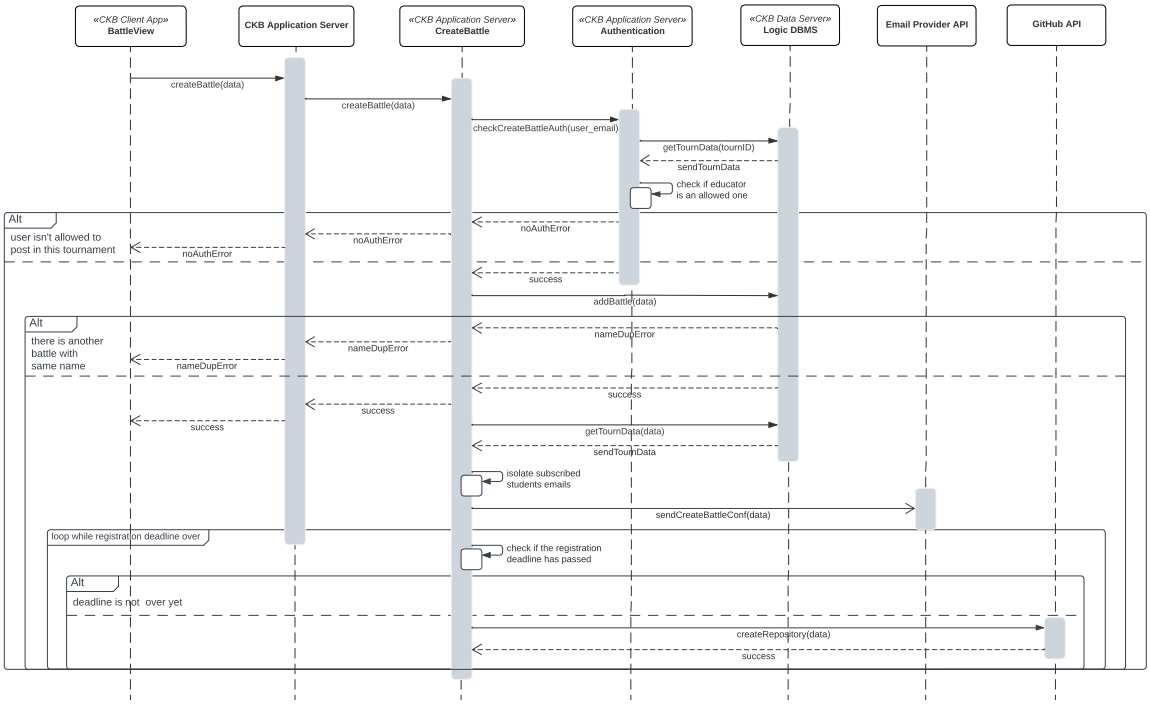


Figure 9: Registered user creates a battle

Here the *createBattle* component is also responsible to handle the end of the battle. So there should be another timer that check when the battle is over and the component can either directly calculate the final rank and send a notification to the students that joined the battle or start another timer for the consolidation phase in which the educator can manually evaluate codes. This is not hard and can be implemented inside another loop but would be too complex to read and so it's not shown in this diagram.

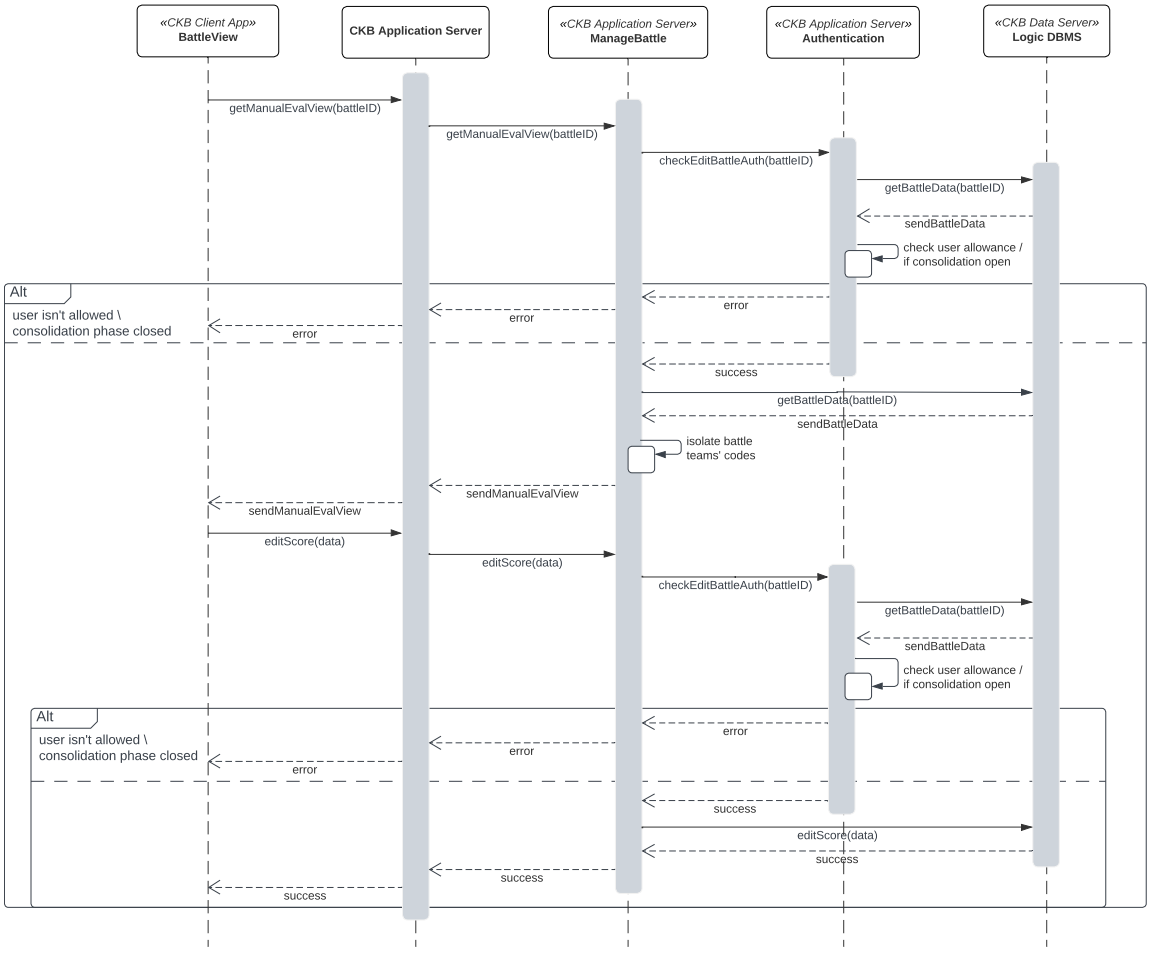


Figure 10: Registered user manually evaluates teams' codes

Very similar to the flow of editing a battle's settings in the next figure.

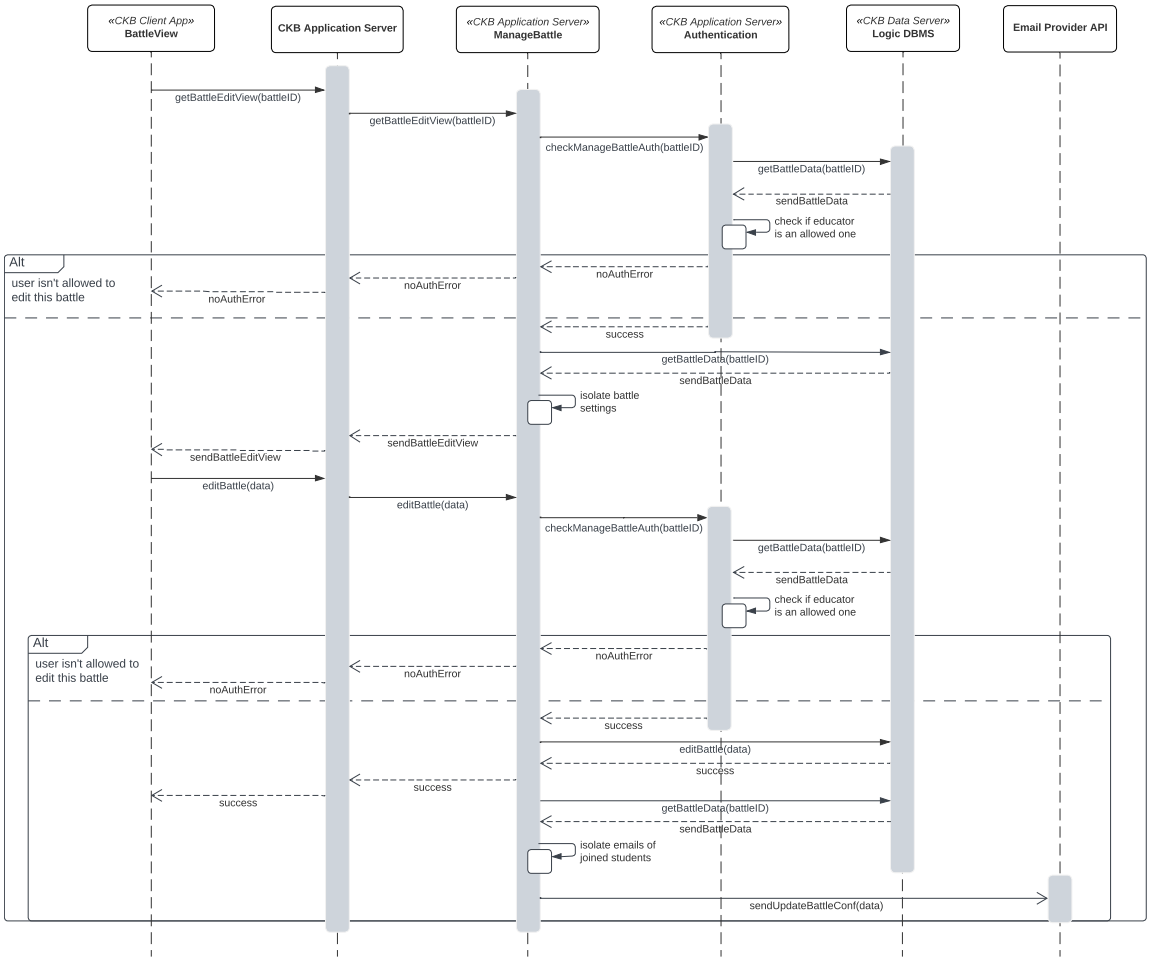


Figure 11: Registered user modifies a battle's settings

In figure 11 the diagram for a random battle's setting update is represented. The procedure shown is the one to update the deadlines. Modifying the max(min) number of teams's members requires a check on the battle's data to see if there are teams that exceed the new max (are below the new min) and an error message to be sent back if so.
Removing a team requires one more request to shows the teams' list.
To delete the battle, instead, there is not the need to send back the settings so it's a simple request that implies the auth phase and an update of the Logic DBMS.

The *manageTournament* component is not shown because it involves the same steps illustrated above.
The only different operation is the functionality to grant other educators the ability to post battles. Here the component simply checks with the *User DBMS* if the email is correct and updates the tournament data with the new email.

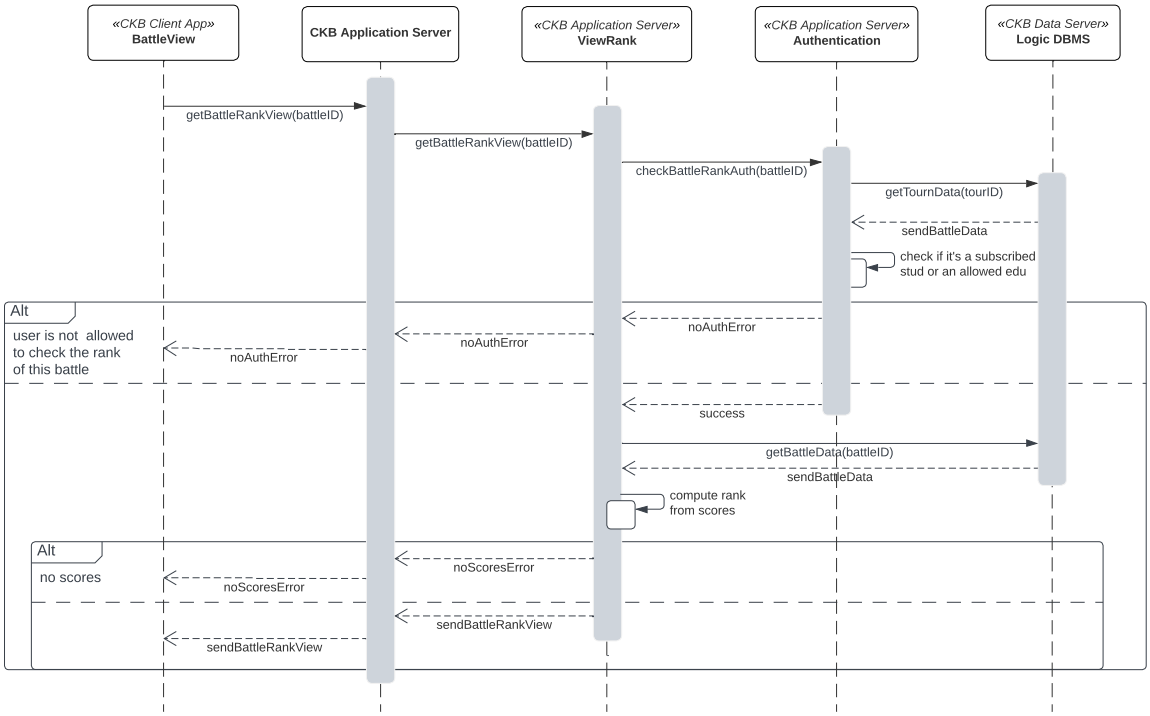


Figure 12: Registered user checks a battle's rank

The process to check tournaments' rankings is the same but the auth phase is not needed as tournaments' rankings are available to all users.

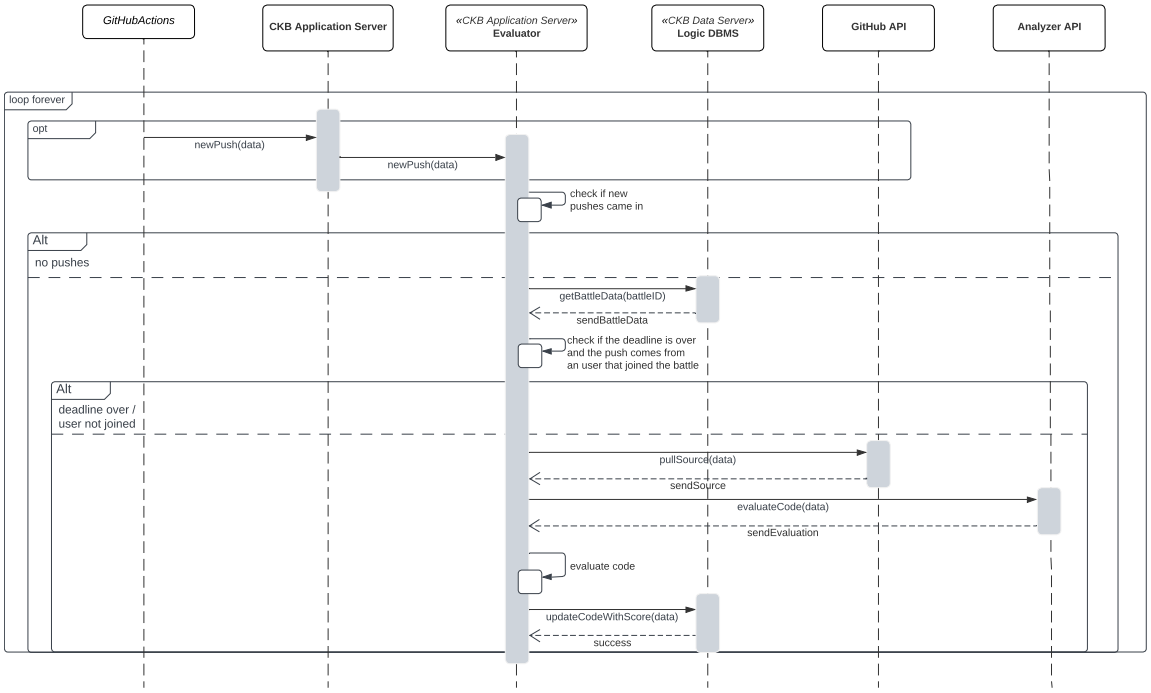


Figure 13: Listening to new GitHub pushes and evaluating them

This component is responsible to keep listening to alerts of new GitHub pushes sent to the CKB platform due to a GitHub action. For all the pushes sent before the submission deadline the *Evaluator* component must retrieve the codes, evaluate them and update the score in the *Logic DBMS*. For what concerns the first two scores that has to be given to the code they can be done locally by the *Evaluator* component. For the firsts the evaluator must run the code on the tests case while for the second it just has to check the time between the push that was catched and the registration deadline from the battle's settings.
For the score based on the static analysis tools, local testing tools can be installed and used but the most used solutions are web based and have to be accessed via an API. For the two most known solutions (Codacy and Sonar) there is also the requirement that the teams' members join the same organization (that could be the CKB organization) and they fork the battle's repository inside the same organization. All of those must be added as domain assumption in the RASD if implemented. Here it's supposed that the system is using a random service that lets just send the code and the parameters that must be evaluated and return the analysis.

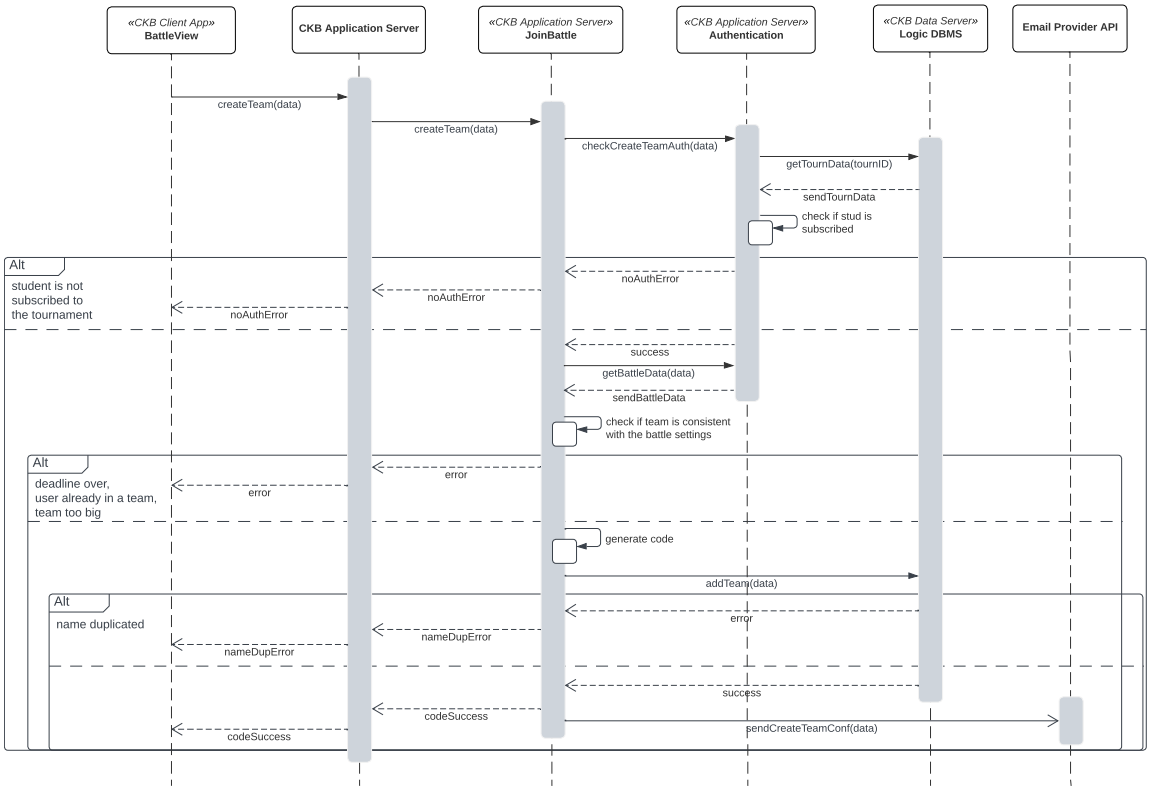


Figure 14: Registered user creates a team for a battle

This diagram shows the creation of a team of number of members bigger than 1. When creating a solo team the step to create the code is not needed.
Joining a team via code instead it's the same but it's not needed to check battle's settings to see if they are consistent with the team about to be created. The only check is to see if the user is already in another team or if the team is full. The request to the *Logic DBMS* is not a *createTeam()* one but a *updateBattle()* request to add the user to the team.

The *subToTournament* is not shown as it's very simple, the only check required is to the *Authentication* component to see if the user is a student and to the tournament's settings to see if the user is already subscribed.

## 2.5 Component Interfaces

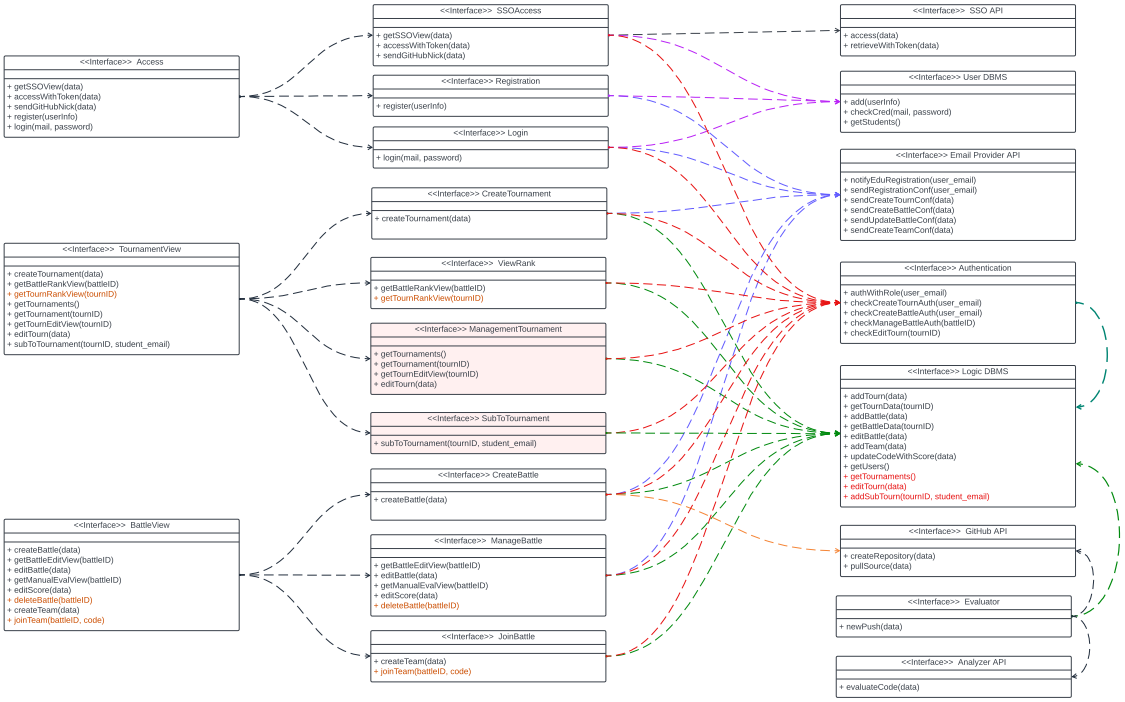


Figure 15: Component Interfaces diagram

The diagram represents the interfaces offered by the different components that were introduced in *section 2.2*, specifically showing the methods for each of them.

Also (in red) there is the representation of interfaces and methods of which the flow was not represented directly in the sequence diagrams of the *section 2.4* but that are still part of the system.

## 2.6 Selected architectural styles and patterns

**Three Tier Architecture:** As previously introduced, the system is based on the three tier architecture as it's the preferred architecture for a web platform.
The *Presentation Layer* provides the users with an interface to access the platform and gather information from the users that are sent to the application layer. In our system it's the CKB Client App.
The *Application Layer* is where the logic of the platform is executed, it handles and replies to the requests coming from the users and is also the intermediary between the presentation and the data tier. In our system it's represented by the CKB Application Server.
The *Data Layer* is where the data relevant to the app is stored and retrieved. In our case it's the

CKB Data Server where data about the users and the logic of CKB is stored.

## 2.7 Other design decisions

One of the main limitation of the architecture of the architecture of the platform is that the coding space is not internal but it depends on GitHub and in particular there is the assumption that all users setup correctly the *GitHub Actions* otherwise their work won't be taken into account.
To get better results the system could check periodically the availability of updates in all the forked repository of the main battle's repository and check if it's been done by a GitHub nick of an user that joined the battle.
This is a better approach to avoid missing evaluations but is way more resource consuming.

**Hashing:** as defined in the RASD it's advised to not store user credentials in plain text in the User DBMS. For this reason both the *Registration* and *Login* component are responsible to save passwords in a safe way. For this reasons to each password is added a salt (which is stored too) and then the password is hashed with SHA-256 before being saved in the DB.

# 3 User Interface Design

In this section there is an overview of the main interfaces provided to users when they access CKB. In particular there is a distinction between what it's presented to the two main actors: educators and students.

## 3.1 CKB Educator Interface

### 3.1.1 Educator Sign-Up

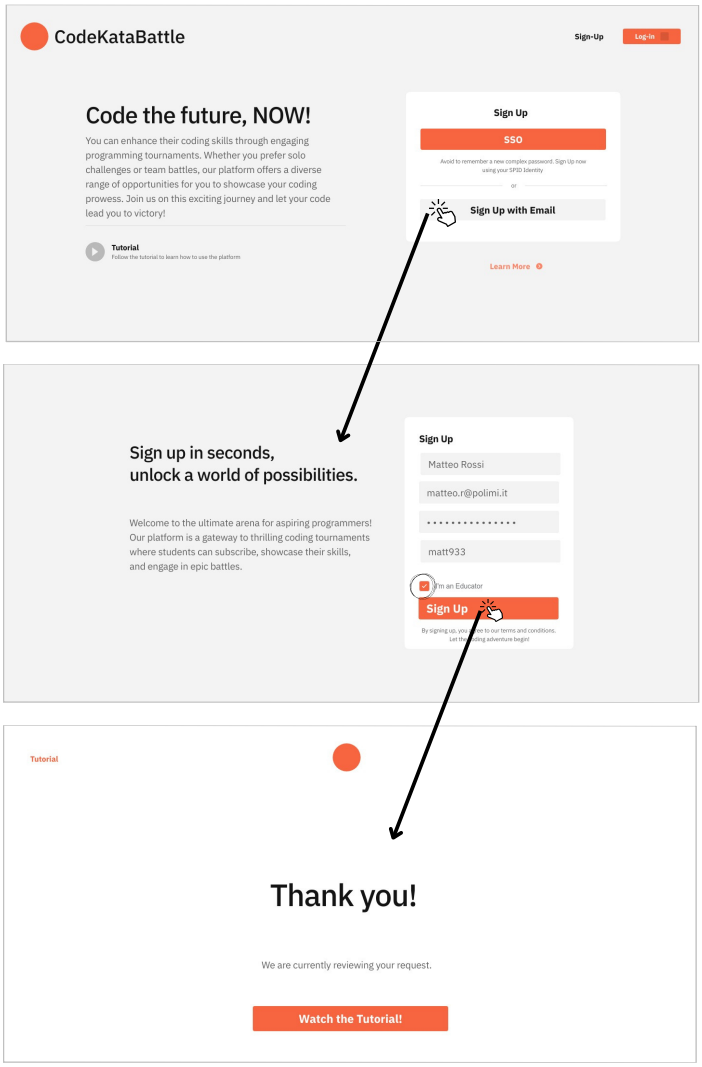


Figure 16: Educator Sign-Up

Here an educator from the CKB home page decides to sign-up with the email. Registration form is shown and he fills it with name and surname, email, GitHub nickname and he selects "I'm an educator". Since it is an educator, the registration is not immediate but he has to wait the approval from the CKB staff, so a waiting page is shown.

### 3.1.2 Tournament Creation

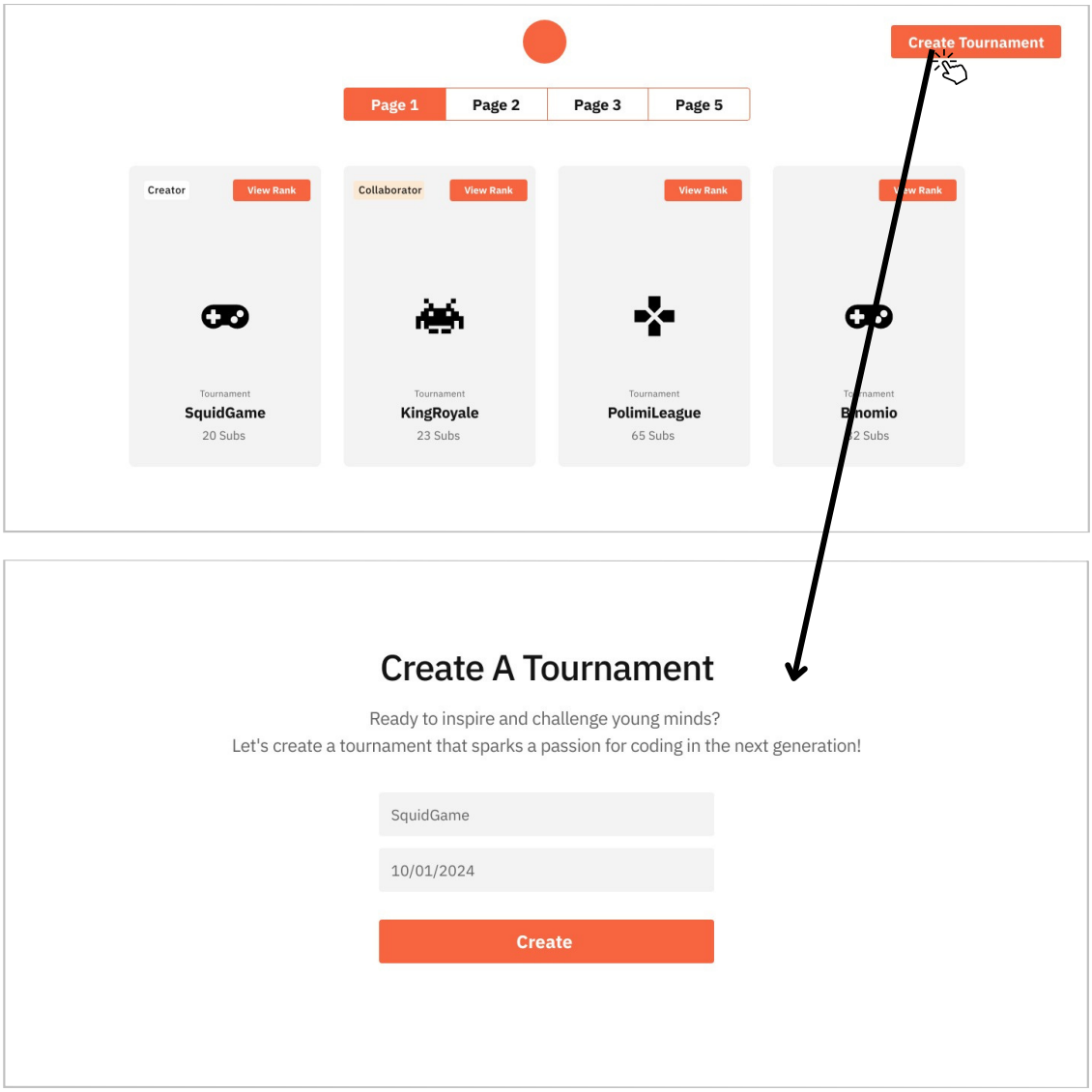


Figure 17: Tournament Creation

Educator creates a tournament by filling the form inserting tournament's name and registration deadline. Tournament is now created successfully and the list of managed tournaments is shown.

### 3.1.3 Battle Creation

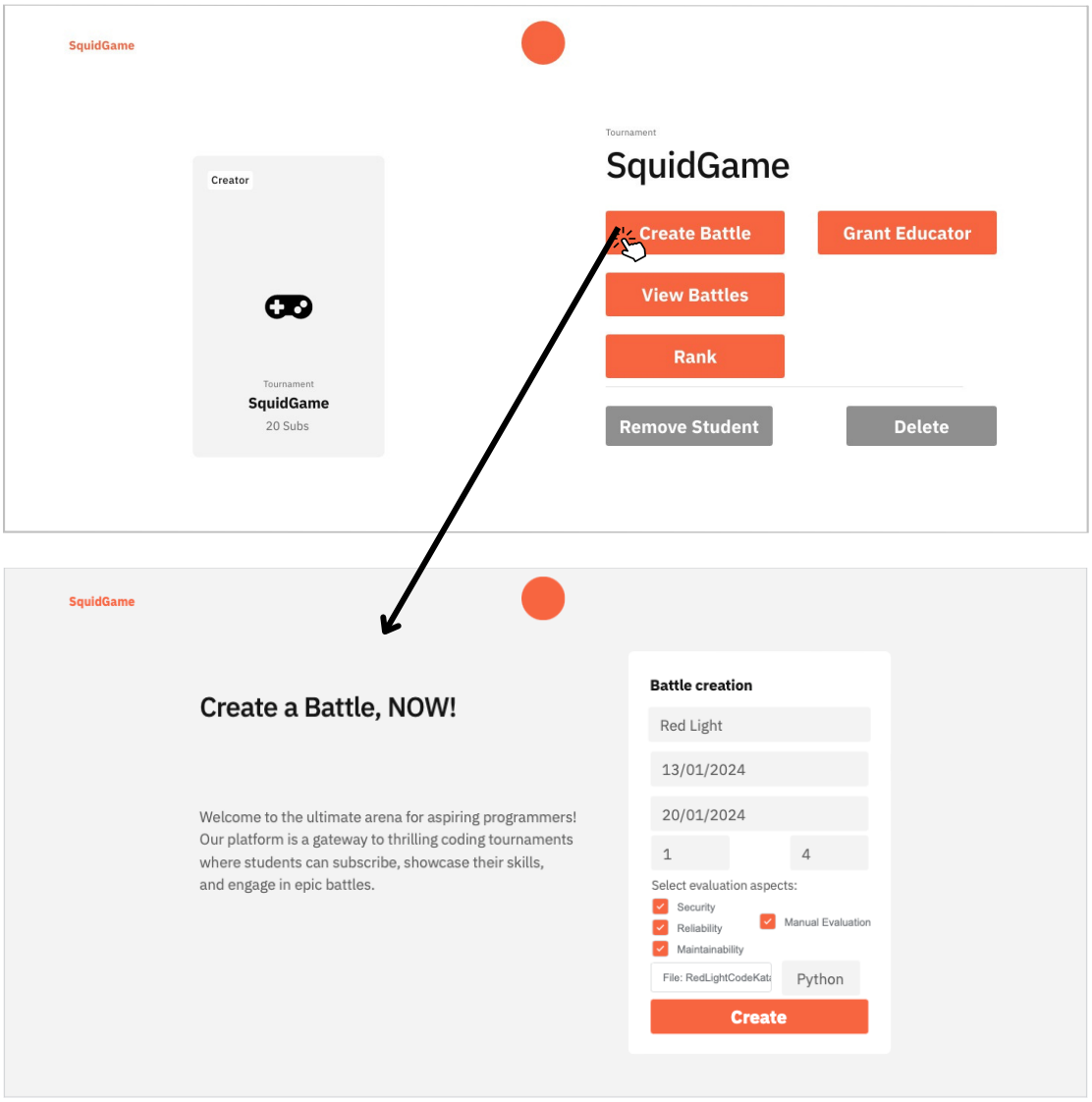


Figure 18: Battle Creation

Educator is now on the "SquidGame" tournament management page and he starts the creation of a new battle by clicking the dedicated button. He fills the form inserting the battle name, registration deadline, submission deadline, minimum and maximum number of team's members, selects the manual evaluation required for the battle, he flags the aspects for the evaluation, uploads the CodeKata file and he selects the battle programming language code. The operation has success

and the list of managed battle is shown including the new one.

### 3.1.4 View Battle Rank

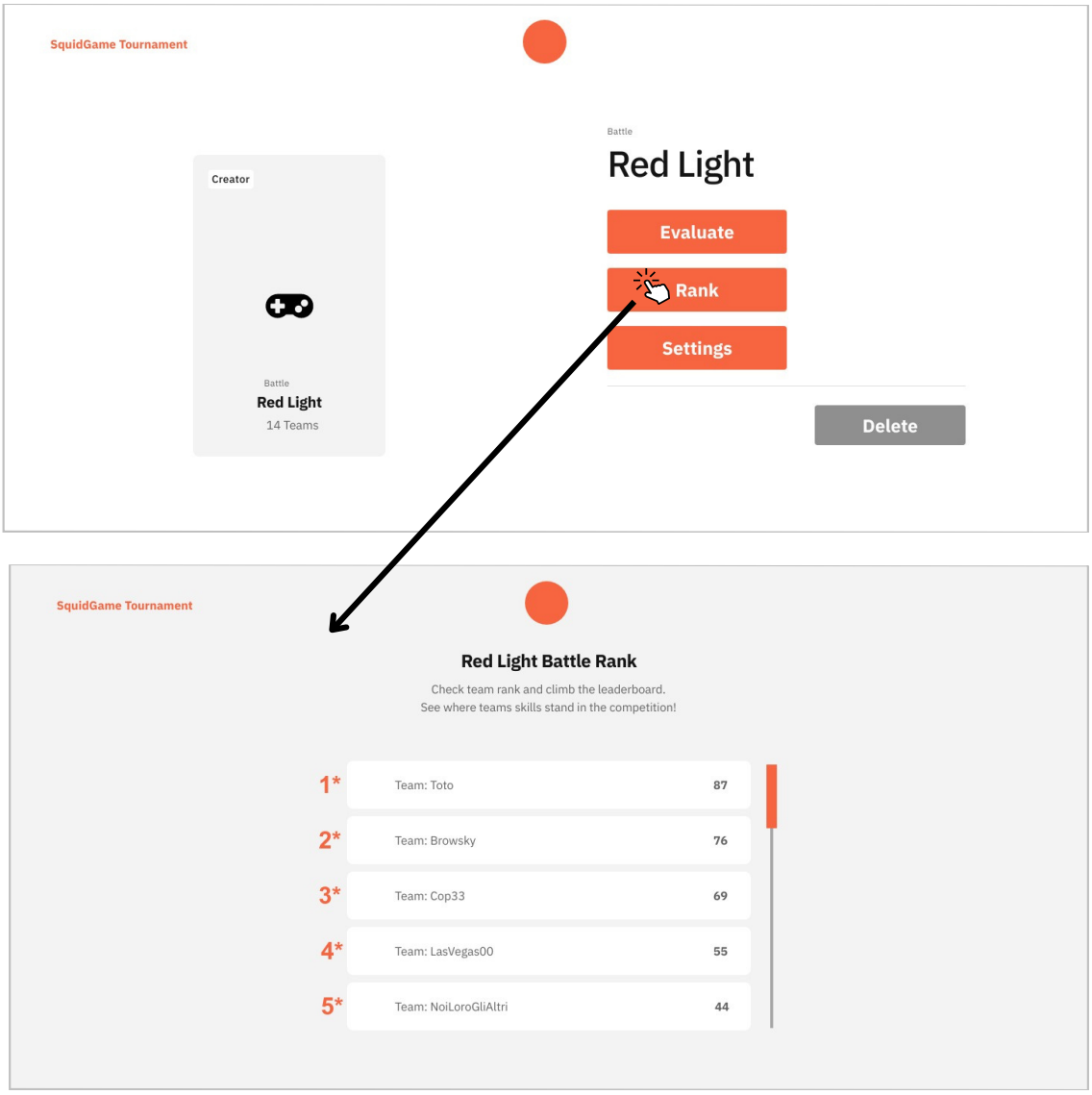


Figure 19: Battle Rank

Educator is in management battle page and he decides to view battle's rank by clicking the dedicated button. Now the battle's rank is shown with all teams and related scores.

## 3.2 CKB Student Interface

### 3.2.1 Student Sign-Up

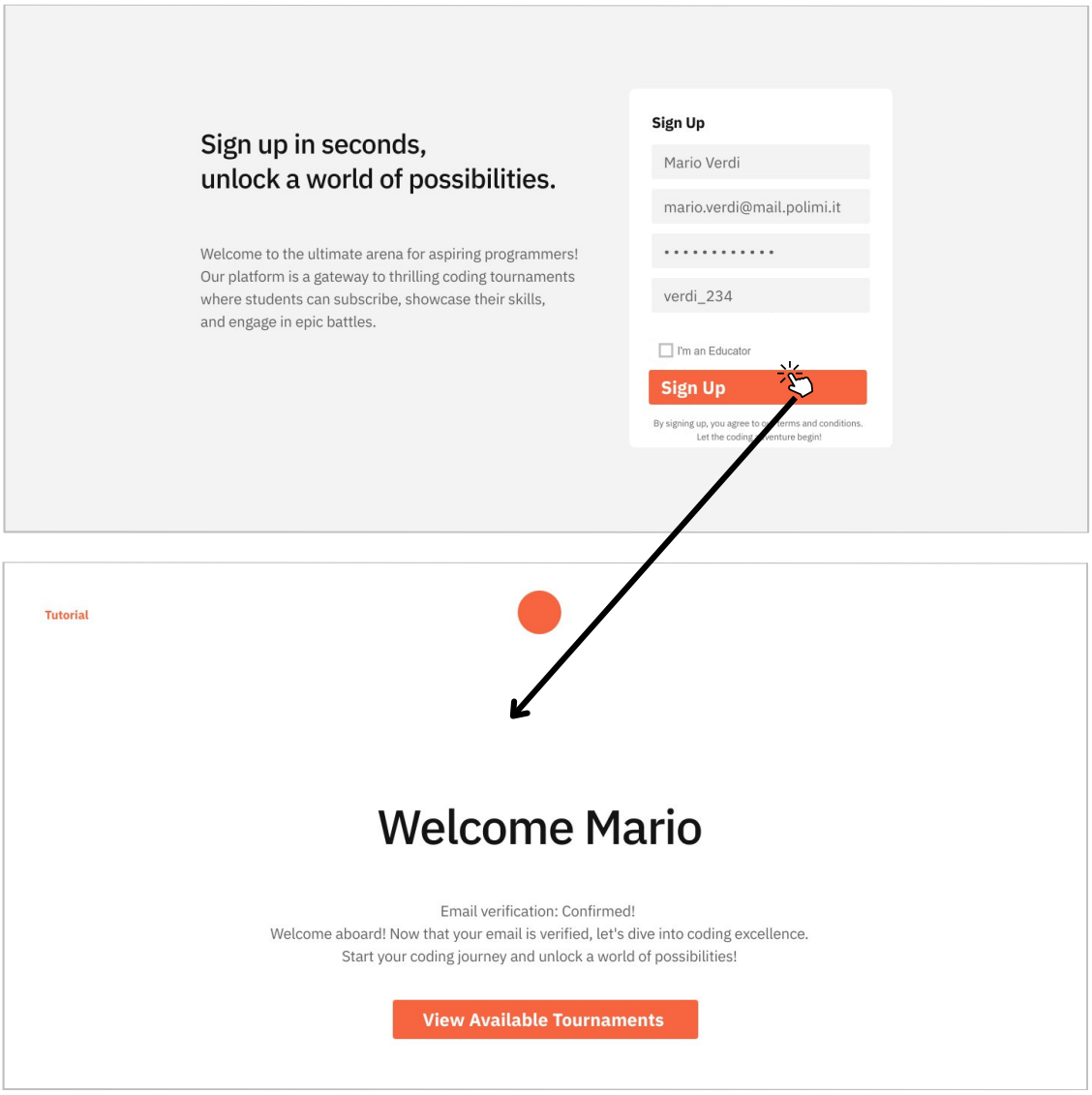


Figure 20: Stundent Sign-Up

A student is on the home page of CKB web-app and he decides to sign-up by clicking the button. The registration form is shown and he fills it with name and surname, email, password and he doesn't select the flag "I'm an educator". After having verified the email a success page is shown.

### 3.2.2 Tournament Subscription

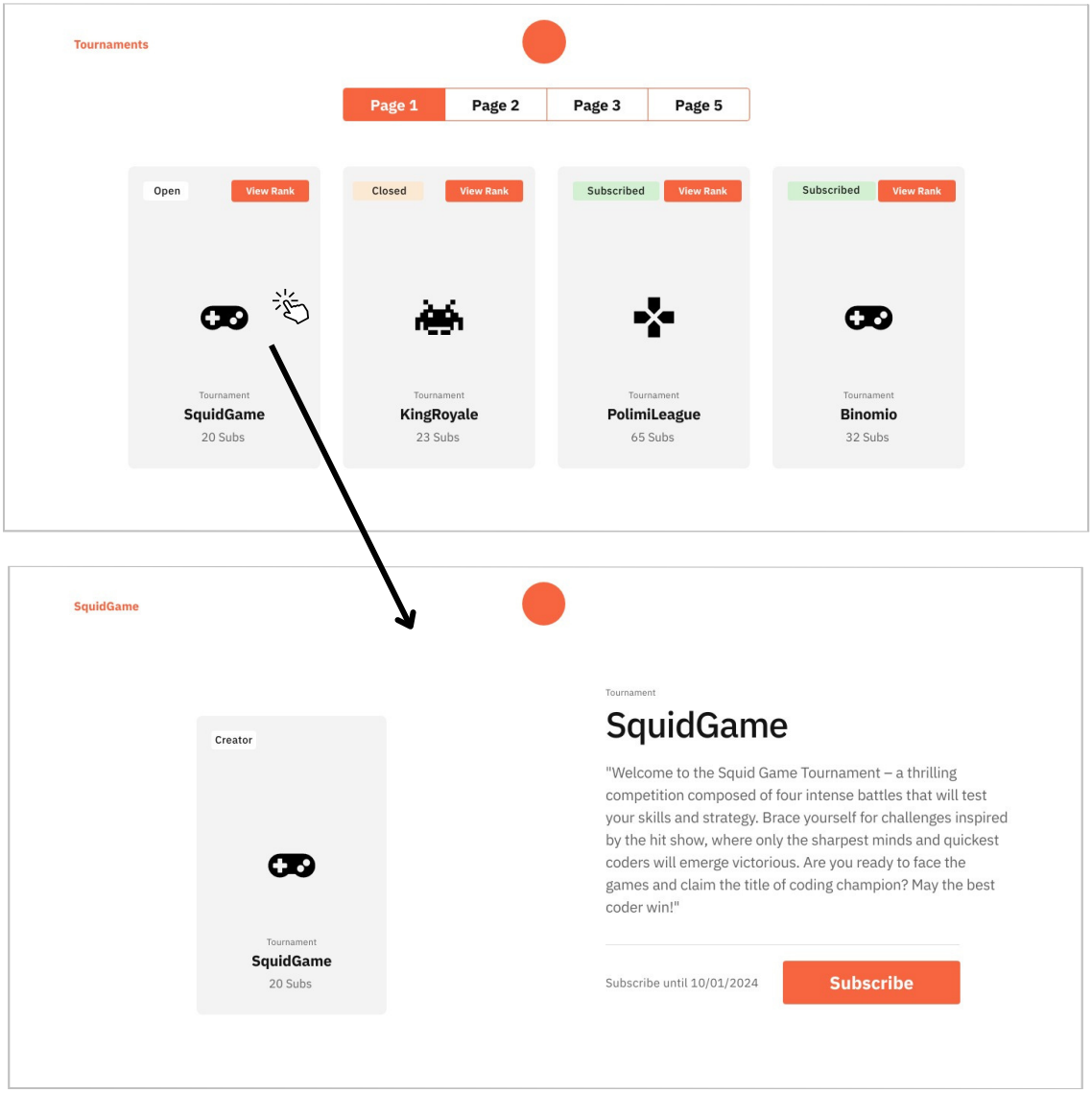


Figure 21: Tournament subscription

The student is now on a page that shows all tournaments (open, closed and already subscribed) and he decides to subscribe to "SquidGame" tournament. He clicks on it and the tournament page is shown indicating a description, registration deadline and "Subscribe" button. The student clicks on it and if the subscription has success he enters on it and the page of all related battles is shown.

### 3.2.3 Join A Battle

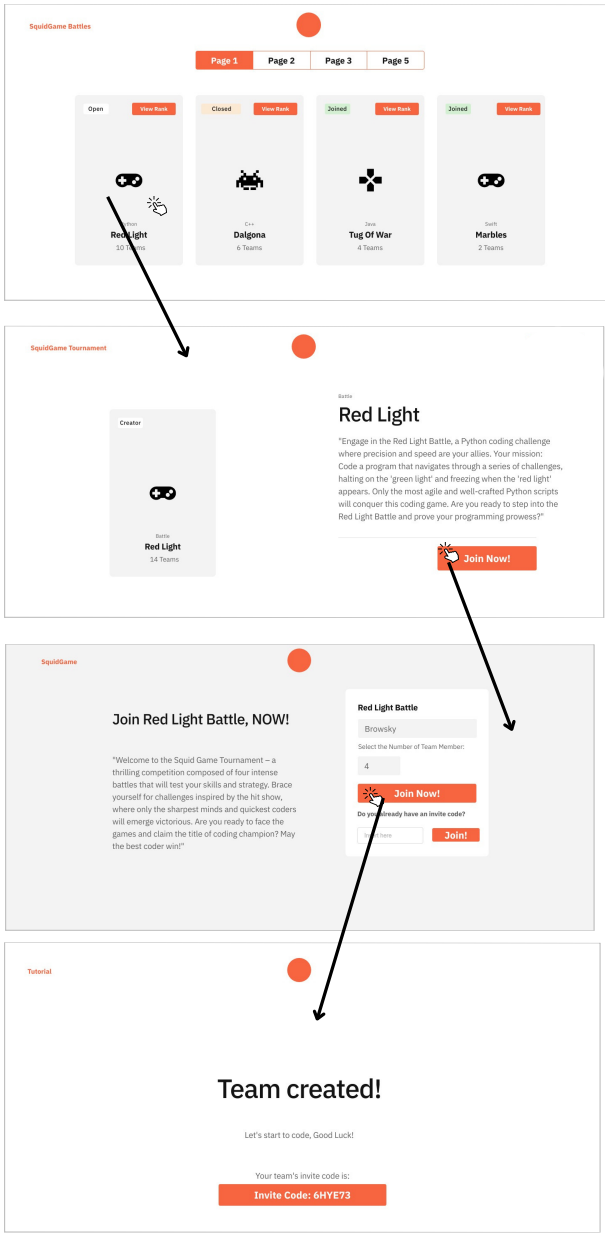


Figure 22: Join a battle

The student is on "Red Light" battle and he clicks on "Join Now!". A form is shown and he fills it with the name of the team and the number of team's members. The team is created successfully and a page with the invite code is shown.

### 3.2.4 Manual Evaluation

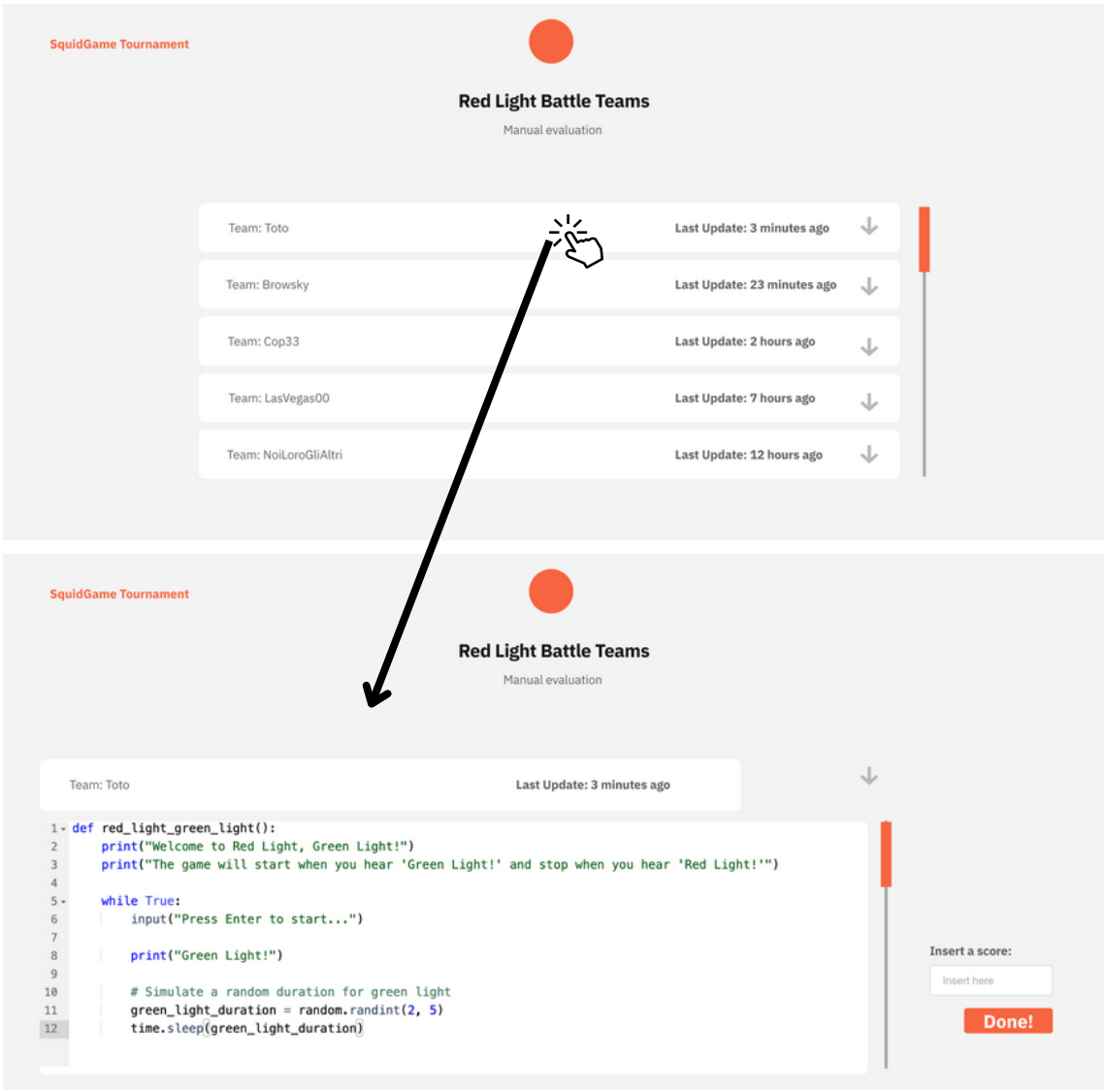


Figure 23: Manual evaluation

For "Red Light" battle is required the manual evaluation by the educator. So he enters in the management battle and the in the evaluation section. Here is shown the list of teams that have submitted at least one time and the last code update. The educator can select a team and it is shown the last code submitted, he now can assign a score and submit.

# 4 Requirements Traceability

In this section a set of tables is presented, they describe how the requirements defined in the RASD map with the design elements described in this document.

| | |
|---|---|
| **Requirements:** | R1 - The system must allow unregistered student to sign up<br>R2 - The system must allow unregistered educator to sign up<br>R3 - The system must notify the user about the successful registration via an external mail service |
| **Components:** | Client App:<br>• Access: shows to the user the dedicated page to sign-up<br>Application Server:<br>• Registration: handles the request to register by querying and sending what to save to the User DBMS. It then sends the info about the result of the operation to the client to be displayed<br>• user DBMS: used to check if the user is already registered and if not it saves user information<br>External API:<br>• Email Provider API: sends registration confirmation |

| **Requirements:** | R4 - The system must allow educators/students to access the platform via an external SSO service |
|---|---|
| **Components:** | Client App:<br>• Access: shows to the user the dedicated page access the external SSO service<br>Application Server:<br>• SSO Access: handles the access by querying the external SSO service to gather user info and then communicate with the two components below<br>• Authentication: if the user is already registered it's used to login the user by giving to the user the corresponding role with permissions (educator / student)<br>• user DBMS: to check if the user is already registered and if not to save its info |

| **Requirements:** | R5 - The system allows registered student to login<br>R6 - The system allows registered educator to login |
|---|---|
| **Components:** | Client App:<br>• Access: shows to the user the login page<br>Application Server:<br>• Login: checks if the user info are correct and gives the access to CKB platform. It then sends the info about the result of the operation to the client to be displayed<br>• Authentication: gives to the user the corresponding role with permissions (educator / student)<br>• user DBMS: used to check user credentials |

| | |
|---|---|
| **Requirements:** | R7 - The system must allow educators to create tournaments (setting name and subscription deadline)<br><br>R12 - The system must notify registered students about the creation of a new tournament via an external mail service |
| **Components:** | Client App:<br><br>• TournamentView: provides to the educator the page to create the tournament<br><br>Application Server:<br><br>• CreateTournament: handles the tournament's creation process communicating with the components below to check if it's possible to create the tournament and if so to save the data. It then sends the info about the result of the operation to the client to be displayed<br><br>• Authentication: checks if the user that sends the requests is logged in and it's an educator<br><br>• Logic DBMS: stores new tournament data<br><br>• User DBMS: retrieves all students' data to filter emails<br><br>External API:<br><br>• Email Provider API: using all students emails retrieved previously, it is used to notify students about the creation of a new tournament |

| | |
|---|---|
| **Requirements:** | R8 - The system must allow the creator of a tournament to delete it and to remove students from it, students affected must be notified via an external mail service<br><br>R9 - The system must allow the creator of a specific tournament to invite other colleagues to post battles in it and notify them via an external mail service |
| **Components:** | • TournamentView: provides to the educator the page to manage the tournament<br><br>Application Server:<br><br>• ManageTournament: handles the tournament's deletion process communicating with the components below to get the students' list and to remove the students from the tournament data. It then sends the info about the result of the operation to the client to be displayed<br><br>• Authentication: checks if the user that sends the requests is logged in and is the creator of the tournament<br><br>• user DBMS: used to check if the indicated educator exists<br><br>• logic DBMS: removes deleted tournament, add granted educator for that tournament<br><br>External API:<br><br>• Email Provider API: retrieves the emails of the student subscribed to the tournament and notifies them about the update. It also sends the email to the invited educators |

| **Requirements:** | R10 - The system must allow users to see all the available tournaments |
|---|---|
| **Components:** | Client App:<br>• TournamentView: provides to all users the page to list all available tournaments<br>Application Server:<br>• Authentication: checks if the user that sends the requests is logged in<br>• ManageTournament: provides to the TournamentView client app component the list of all available tournaments<br>• logic DBMS: used to retrieve all available tournaments |

| **Requirements:** | R11 - The system must allow users to see tournaments' ranks |
|---|---|
| **Components:** | Client App:<br>• TournamentView: provides to all users the page to show the tournament rank<br>Application Server:<br>• Authentication: checks if the user that sends the requests is logged in<br>• ViewRank: creates the rank and provides it to the TournamentView client app component<br>• logic DBMS: used to retrieve all data in order to create the rank |

| | |
|---|---|
| **Requirements:** | R13 - The system must allow registered students to subscribe to a tournament. Students must be notified of the successful subscription via an external email service |
| **Components:** | Client App:<br>• TournamentView: provides the page to subscribe to a tournament<br>Application Server:<br>• Authentication: checks if the user that sends the requests is logged in<br>• SubToTournament: handles the subscribtion process storing in the logic DBMS the subscription of the user in the tournament. It then sends the info about the result of the operation to the client to be displayed<br>• logic DBMS: stores the student subscription<br>External API:<br>• Email Provider API: sends an email to the student about the succesfull subscription to the tournament |

| | |
|---|---|
| **Requirements:** | R14 - The system must allow the creator of a tournament or the educators that were granted the permission to post in it, to create battles in it (this includes setting name, battle's deadline, the aspects of the codes to be evaluated automatically, the min and max members per team and uploading the CodeKata)<br><br>R15 - The system must be able to create a GitHub repository for each battle<br><br>R18 - The system must notify students about the creation of a new battle in a tournament they are subscribed for via an external mail service |
| **Components:** | Client App:<br><br>• BattleView: provides the page to create to create a new battle<br><br>Application Server:<br><br>• Authentication: checks if the user logged in and if it is an educator with permissions to create a battle in that tournament<br><br>• CreateBattle: handles the battle's creation process communicating with the Logic DBMS the battle's data. It then sends the info about the result of the operation to the client to be displayed<br><br>• Logic DBMS: stores new battle's data and it is used to retrieve students subscribed to that tournament (useful to get emails addresses)<br><br>External API:<br><br>• Email Provider API: used to send the notification about the creation of a new battle<br><br>• GitHub API: used to automatically create the GitHub battle repository when the registration deadline expires |

| | |
|---|---|
| **Requirements:** | R16 - The system must allow the creator of a specific battle to delete it, to change minimum and maximum number of students per team, to modify the registration deadline, to change the final submission deadline, to remove a team. Students affected must be notified via an external mail service<br><br>R22 - The system must allow the creator of a battle to evaluate team's codes for that battle if the battle's settings expect it |
| **Components:** | Client App:<br>• BattleView: provides the page to manage a battle and the page to manually evaluate the code, if required.<br>Application Server:<br>• Authentication: checks if the user logged in and if it's the creator of the battle<br>• manageBattle: handles the battle's management process communicating with the Logic DBMS to retrieve and update the battle's settings, the team's that joined, the team's codes with the corresponding score. It then sends the info about the result of the operation to the client to be displayed<br>• Logic DBMS: used to update battle's settings, insert the educator personal score for a given team and retrieve battle's students data<br>External API:<br>• Email Provider API: sends notification about the update of battle's settings |

| **Requirements:** | R17 - The system must allow subscribed students to a tournament and educators that were granted permissions for the tournament to see all the available battles in that tournament |
|---|---|
| **Components:** | • BattleView: shows the page that contains the list of battles of a given tournament<br>Application Server:<br>• Authentication: checks if the user is logged in and if it is a student who subscribed to the tournament of which the battles are part of or it's an educator with permissions for that tournament<br>• manageBattle: provides to the BattleView client app component the list of battles of a tournament<br>• Logic DBMS: used to retrieve all battles for a tournament |

| **Requirements:** | R19 - The system must allow subscribed students to a tournament and educators that were granted permissions for the tournament to see ranks of battles in that tournament |
|---|---|
| **Components:** | Client App:<br>• BattleView: provides to users the page to show the battle rank<br>Application Server:<br>• Authentication: checks if the user is logged in and if it is a student who subscribed to the tournament of which the battle is part of or it's an educator with permissions for that tournament<br>• ViewRank: creates the rank and provides it to the BattleView client app component<br>• logic DBMS: used to retrieve all data in order to create the rank |

| | |
|---|---|
| **Requirements:** | R20 - The system must allow subscribed students to a tournament to create a team for a battle in that tournament. Student must be notified of the successful join via an external email service<br><br>R21 - The system must allow subscribed students to a tournament to join a team for a battle in that tournament by providing the team's invite code. Student must be notified of the successful join via an external email service |
| **Components:** | Client App:<br>• BattleView: shows the page for a student to create/join a team<br>Application Server:<br>• Authentication: checks if the user is logged in and if it is a student who subscribed to the tournament of which the battle is part of or it's an educator with permissions for that tournament<br>• JoinBattle: handles the battle's joining process communicating with the Logic DBMS to check if team is consistent with the battle's settings, and if registration deadline is not expired. It also checks the code entered by students that want to join a team. It then sends the info about the result of the operation to the client to be displayed<br>• Logic DBMS: adds team data, adds user to a team, stores invite code, retrieves team data, retrieves battle's settings<br>External API:<br>• Email Provider API: sends notification about team creation confirmation and about the successful join to a team |

| | |
|---|---|
| **Requirements:** | R23 - The system must be able to pull repositories from GitHub<br><br>R24 - The system must be able to automatically evaluates teams' codes after each push on GitHub before the submission deadline based on the number of test cases they pass, the timeliness with respect to the deadlines and the set quality aspects with static analysis tools<br><br>R25 - The system must have an API to be able to be informed by a GitHub Action of new students' pushes |
| **Components:** | Application Server:<br><br>• Evaluator: It waits until a notification of a new pushes is available coming from students' that set up a GitHub actions to send new pushes' alerts. Then it pulls the code and evaluate it locally and also sends it to the Analyzer API<br><br>• logic DBMS: used to update team's sources codes and scores<br><br>External API:<br><br>• Analyzer API: it is used to automatically evaluate the code based on the aspects for that battle and returns the corresponding scored |

| | |
|---|---|
| **Requirements:** | R26 - The system must notify students that joined a battle that the final rank for the battle is available via an external mail service |
| **Components:** | Application Server:<br>• ViewRank: update the scores with the score inserted by the educator if the manual evaluation is required and then communicate with the email provider API to send the final rank<br>• logic DBMS: used to retrieve all data in order to create the rank and to send the notification<br>External API:<br>• Email Provider API: sends notification about final rank available for the battle |

# 5 Implementation, Integration and Test Plan

The final chapter outlines the implementation of the CKB platform, detailing the integration of its components and outlining methods for testing. Implementation, integration and testing are closely interrelated, often resulting in the integration order aligning with the implementation order.

## 5.1 Implementation

The platform is created gradually, following a straightforward plan. The concept is to break down the system into various functions or features described above. The approach for implementing the CKB system begins with the development of each sub-component. Once a sub-component is completed, it is tested. The implementation of the system will follow a hybrid approach, combining elements of both the bottom-up and thread strategies. This hybrid strategy aims to leverage the strengths of each approach for a more comprehensive and effective implementation. Combining these strategies involves incorporating thread testing at various integration levels, ensuring that the integrated components function correctly together to support specific threads of execution:

- **Integration with Thread Testing:** After individual modules have been tested bottom-up and integrated into higher-level structures, thread testing can be applied to verify the functionality of specific processes that involve the integrated components. Thread testing helps identify any issues related to the interaction between modules.
- **Iterative Approach:** The integration process can be iterative, with bottom-up integration followed by thread testing and then further integration of additional components. This iterative process continues until the entire system is fully integrated and tested.
- **Bug Detection:** Bottom-up testing is effective for detecting and resolving bugs within individual modules, while thread testing helps uncover issues related to the interaction and communication between modules.

### 5.1.1 Main Modules:

**M1 - Sign-up and Login:** This module is responsible for the user registration (educator/student) using both traditional login and password approach and the Single Sign-On method. It interacts with the user database to store information, with the SSO API and with the Authentication module to generate sessions.

**M2 - Authentication:** This module stores user sessions to check if a user is logged in and also checks for every request if the user performing it has the correct authorizations.

**M3 - Creation and management of a tournament:** This module is for the educators and allows to create tournament and manage it by adding new collaborators or delete it. It interacts with logic database and email API.

**M4 - Creation and management of a battle:** This module is for the educators and allows to create battles, modify their settings and delete them if necessary. It interacts with logic database and email API.

**M5 - Subscribe a tournament and join a battle:** This module is for students that are already subscribed to a tournament and they wants join a battle alone, in a team or creating it. This module interacts with logic database and email API.

**M6 - Evaluator:** This module is related to the automatic evaluation of codes pushed on GitHub for a given battle by joined teams. It interacts with GitHub API to download pushes once a API call from GitHub Actions comes. In addition, it is responsible for educators only and for battles where manual evaluation is required.

**M7 - Ranking:** This module is responsible for the generation of the ranking and it interacts with the logic DMBS in order to acquire the data and with email provider API in order to send notification when the final rank for a battle is available.

## 5.2 Integration Testing Plan:

The main idea is that, combining bottom-up integration testing with a focus on thread interactions, brings efficiency and reliability to the development process.
Threads integration is used only for certain modules in order to provide a first approach of user-visible program feature. Thanks to this, if a module contains a different version of the same feature, using the thread approach it's possible to firstly develop one of them and tests it and then implement the other option (like classic log-in and SSO).
Overall, the concept reflects a strategic approach to incorporating thread interactions into the integration testing process, contributing to a more robust and flexible software development cycle.

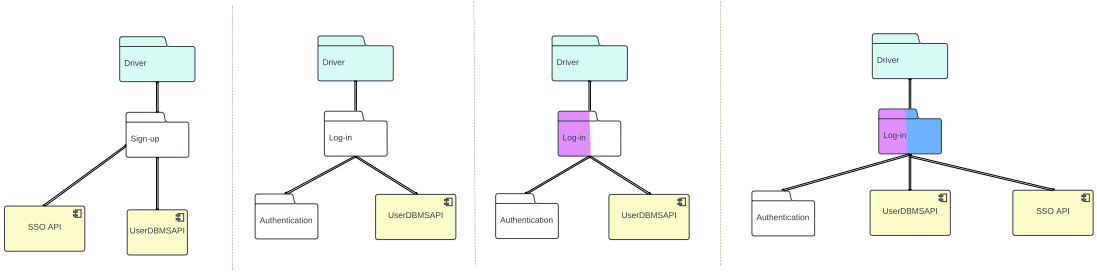


Figure 24: Authentication and Login/Sign-up module

This figure illustrates how the bottom-up and thread approach are used to implement the authentication and login/sign-up module. After integrating and testing the Authentication component, the login and sign-up functionalities have been integrated, beginning with the classic login approach (username/password, represented by the purple color in the figure). Subsequently, the Single Sign-On (SSO) module (depicted in blue) is added.

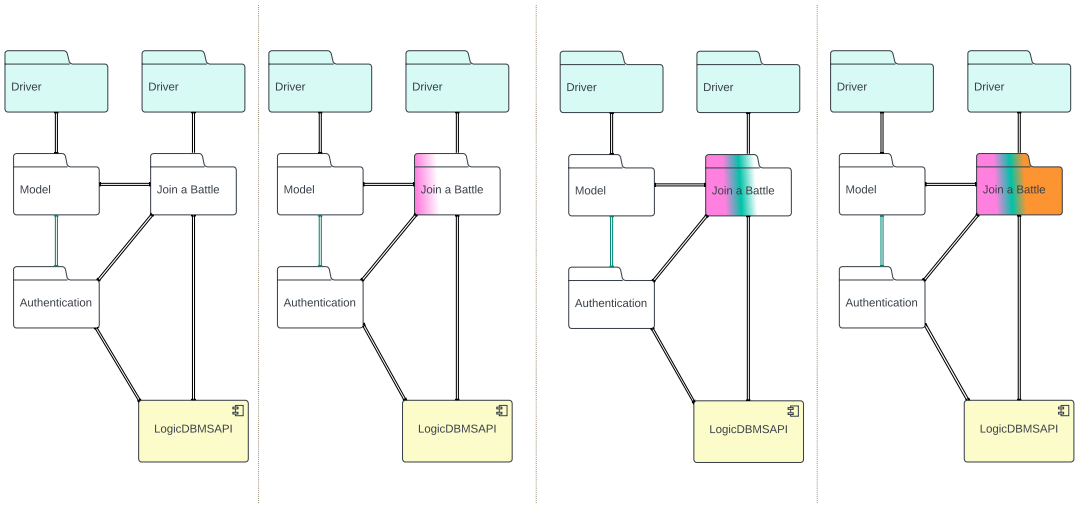


Figure 25: Join a battle module

The same approach as before is considered in the Figure 25 where the Join a battle functionality is added integrating one functionality at each step, starting from join a battle solo (purple), then by creating a team (green) and finally join a team by inserting an invite code (orange).

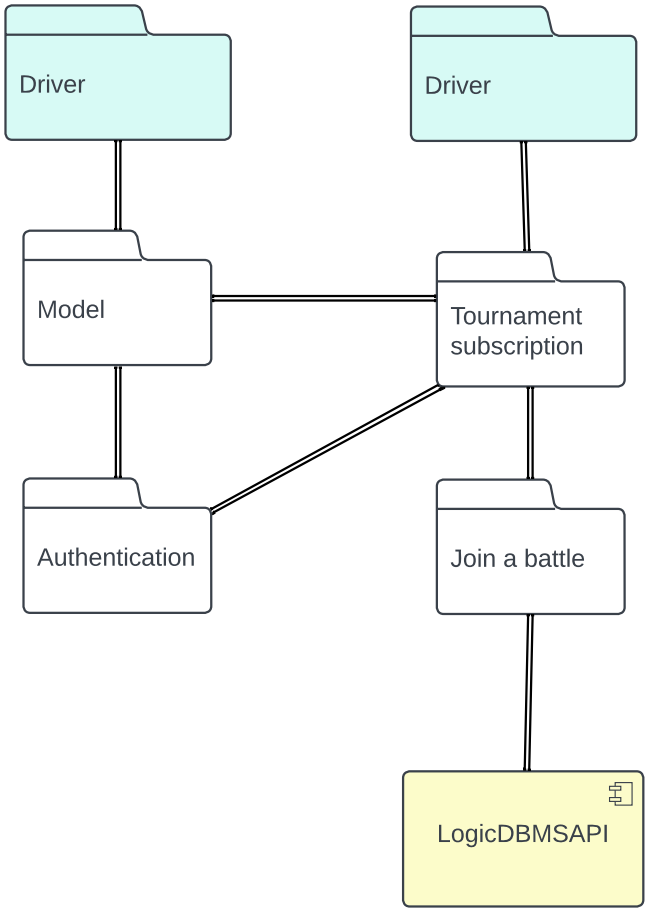


Figure 26: Tournament subscription module

Here is added the parent module that allows to students to subscribe a tournament.

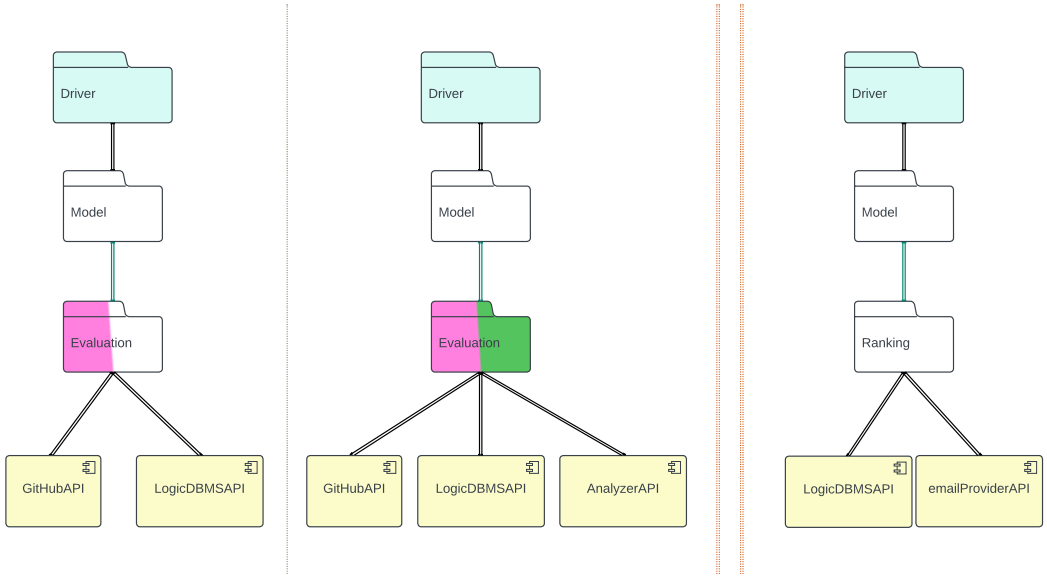


Figure 27: Evaluation and Ranking modules

The first half of Figure 27 shows how evaluation module has been integrated, since it offers two functionalities (automatic and manual evaluation) following the threads approach firstly the manual one is implemented, integrated and tested and finally the automatic one is added. The second half of the figure shows the integration of ranking module. The two integration (evaluator and ranking) can be developed, tested and integrated in parallel.

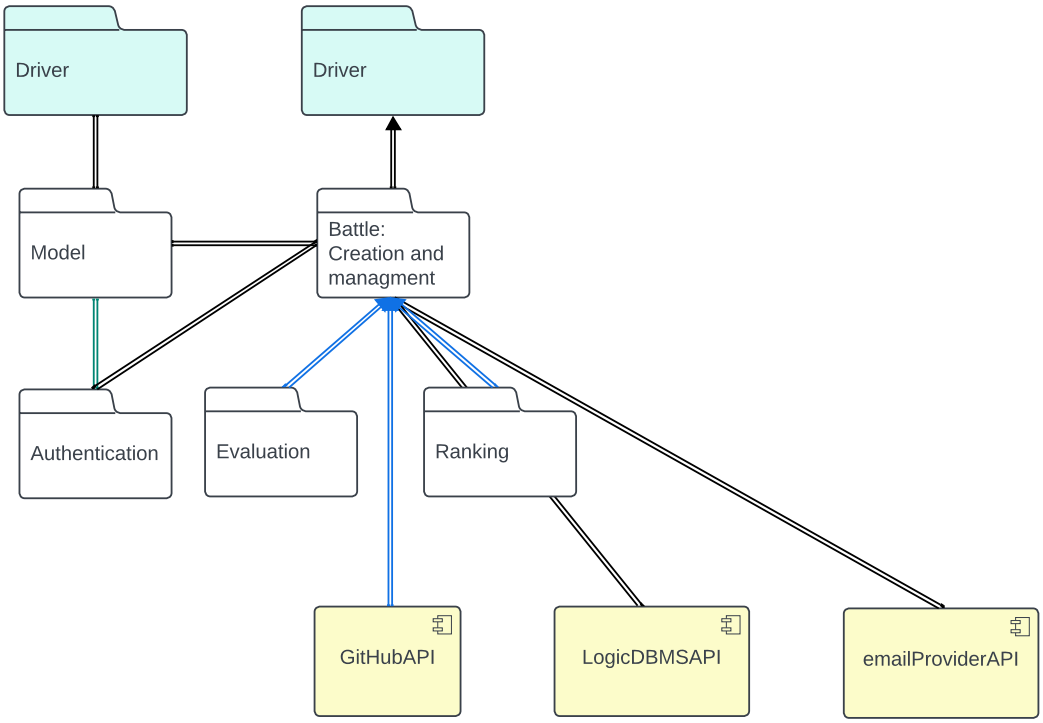


Figure 28: Battle Creation and Management module

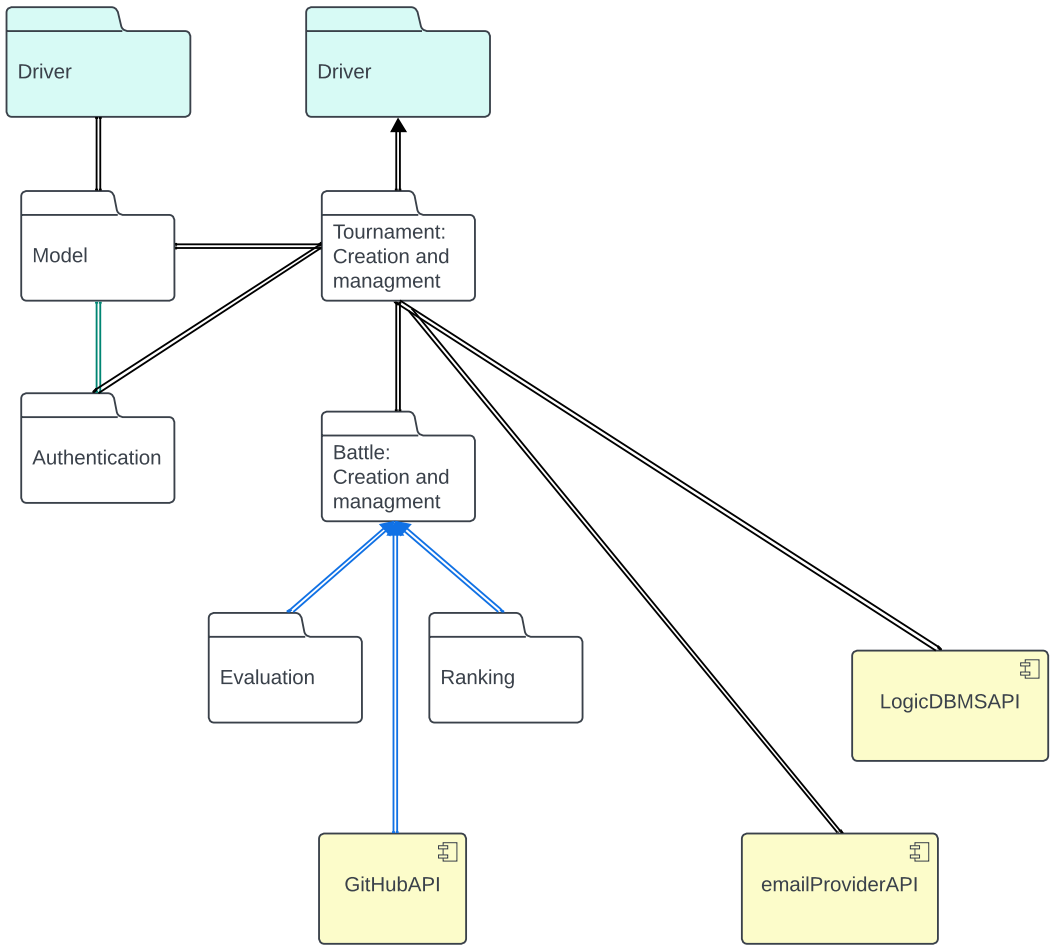


Figure 29: Tournament creation and Management module

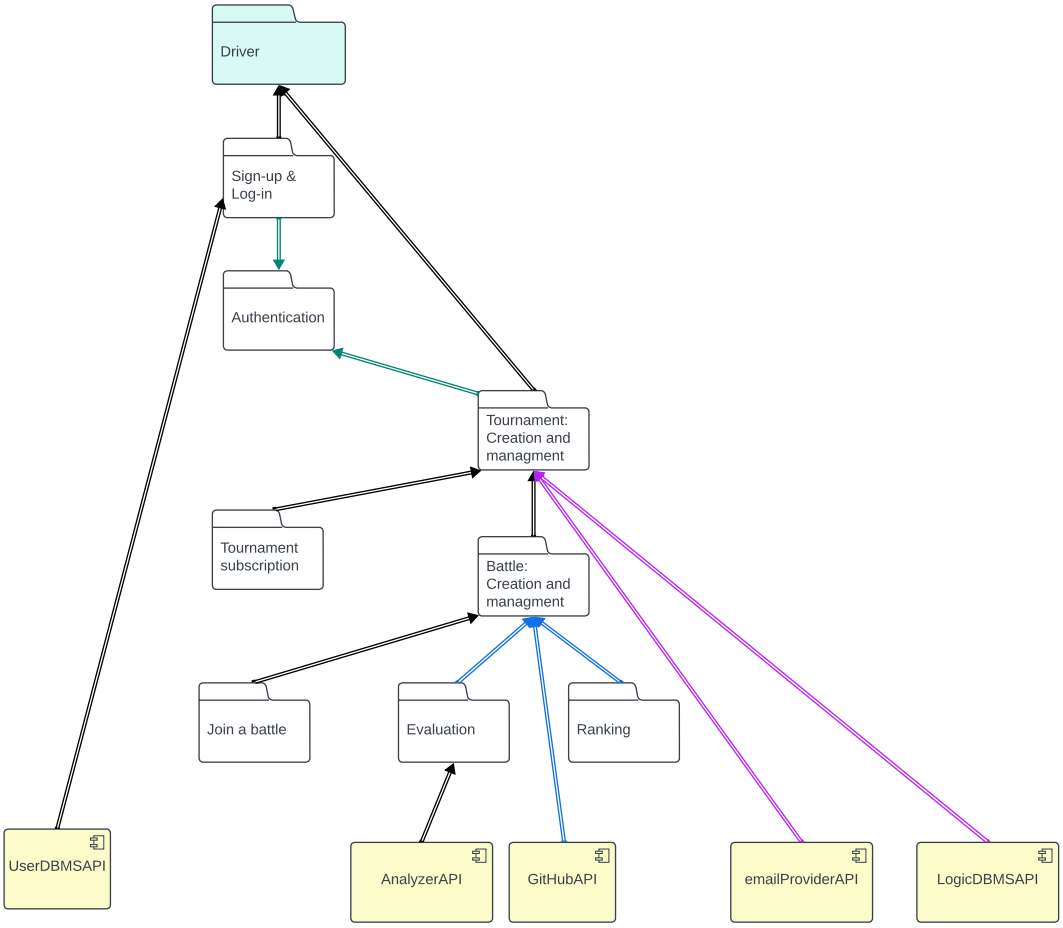


Figure 30: Entire system

# 6 Effort

The timetables provided below are only an estimate of the time dedicated to writing and discussing each specific chapter of this document by the team. These durations were not formally tracked during the document's creation and are solely based on the team members' personal perceptions of the time invested.

**Diegoli Tommaso**

| Section | Effort (in hours) |
|---|---|
| 1 | 3 |
| 2 | 17 |
| 3 | 5 |
| 4 | 4 |
| 5 | 1 |
| **TOT:** | **30** |

**Tagliani Fabio**

| Section | Effort (in hours) |
|---|---|
| 1 | 3 |
| 2 | 6 |
| 3 | 10 |
| 4 | 7 |
| 5 | 3 |
| **TOT:** | **29** |

# 7 References


- SSO Access: https://www.researchgate.net/figure/Single-sign-on-login-sequence-using-a-provider-
fig3_272881174
- GitHub actions: https://docs.github.com/actions
- GitHub documentation: https://docs.github.com/en
- Integration Testing: https://www.geeksforgeeks.org/steps-in-bottom-up-integration-testing/
- Codacy: https://www.codacy.com/
- Sonar: https://www.sonarsource.com