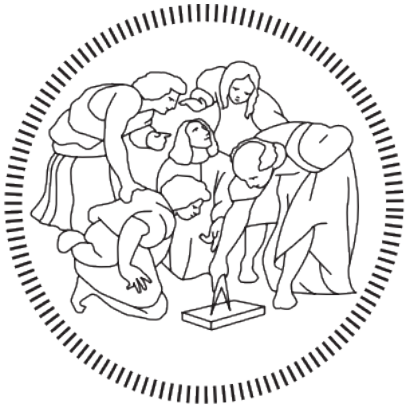

POLITECNICO
MILANO 1863

CodeKataBattle

Design Document

Software Engineering 2 project
Academic year 2023 - 2024

15 November 2023
Version 0.0

Authorss:
Tommaso Pasini
Elia Pontiggia
Michelangelo Stasi

Professor:
Matteo Camilli

Revision History

Date	Revision	Notes
15/11/2023	v.0.0	Document creation
TBD	v.1.0	First relese

Contents

1 Introduction **4**

2 Architectural Design **5**
2.1 Overview 5
2.2 Component view 6
2.2.1 Client components 6
2.2.2 Server components 7
2.2.3 Logical description of the data 9
2.3 Deployment view 10
2.4 Runtime View 11
2.4.1 User login 12
2.4.2 Create a tournament 12
2.4.3 Create a battle 13
2.4.4 Join a Battle Solo 14
2.4.5 Upload code 15
2.5 API endpoints 16
2.5.1 Search for a STU 16
2.5.2 Get owned tournaments 16
2.5.3 Get all tournaments 17
2.5.4 Get tournament details 17
2.5.5 Get battle details 19
2.5.6 Get the list of codes to evaluate 20
2.5.7 Evaluate code 20
2.6 Other design decisions 21
2.6.1 Web Application 21
2.6.2 Single Page Application 21
2.6.3 Relational Database 21
2.6.4 RESTful API 22

3 User Interface Design **23**

4 Implementation, Integration and Test Plan **27**
4.1 Development plan 27
4.1.1 Front-end 27
4.1.2 Back-end 27
4.2 Integration plan 28

5 Effort Spent **32**

1. Introduction

Questa parte qui è mooooolto simile a quella che c'è nel RASD, quindi aspettiamo di avere il RASD definitivo per fare il merge

2. Architectural Design

2.1 Overview

The CKB platform system is composed by a 3-tier architecture.
Its software application architecture is organized into three logical tiers: the presentation tier, or user interface; the application tier, where data is processed; and the data tier, where the data associated with the application is stored and managed.

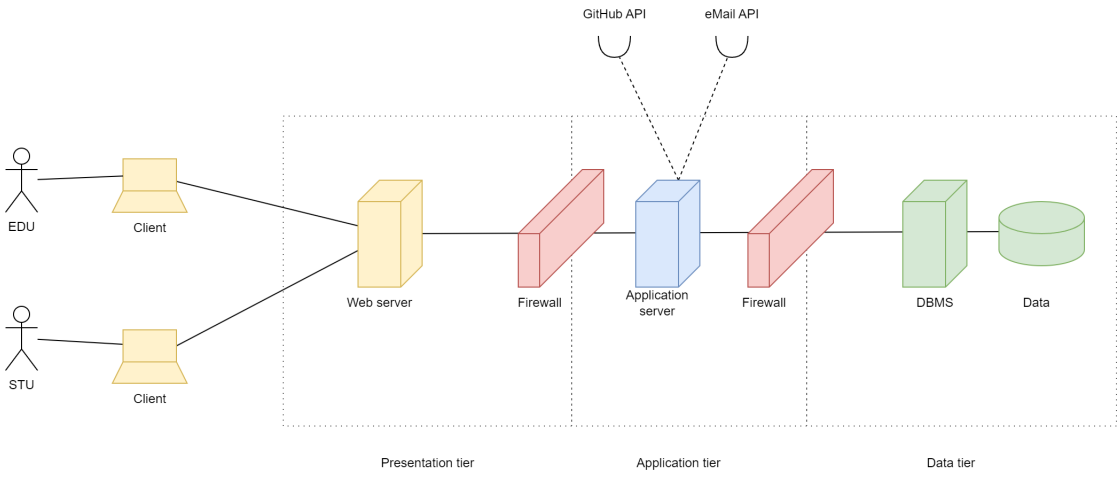

Figure 2.1: High level components diagram

The service will be accessed through a web interface, employing a Single Page Application (SPA). Utilizing an SPA is ideal for this application, as it facilitates extensive interaction without necessitating frequent page reloads.
The system's architecture is structured into distinct layers, with application servers interacting with a database management system and utilizing APIs for data retrieval and storage. Adhering to REST standards, the application servers are intentionally designed to be stateless, handling the login sessions for user thanks to the caching, following the best practices for web applications.
The system will include more than one firewall to ensure security.

2.2 Component view

The system is composed by the following components:

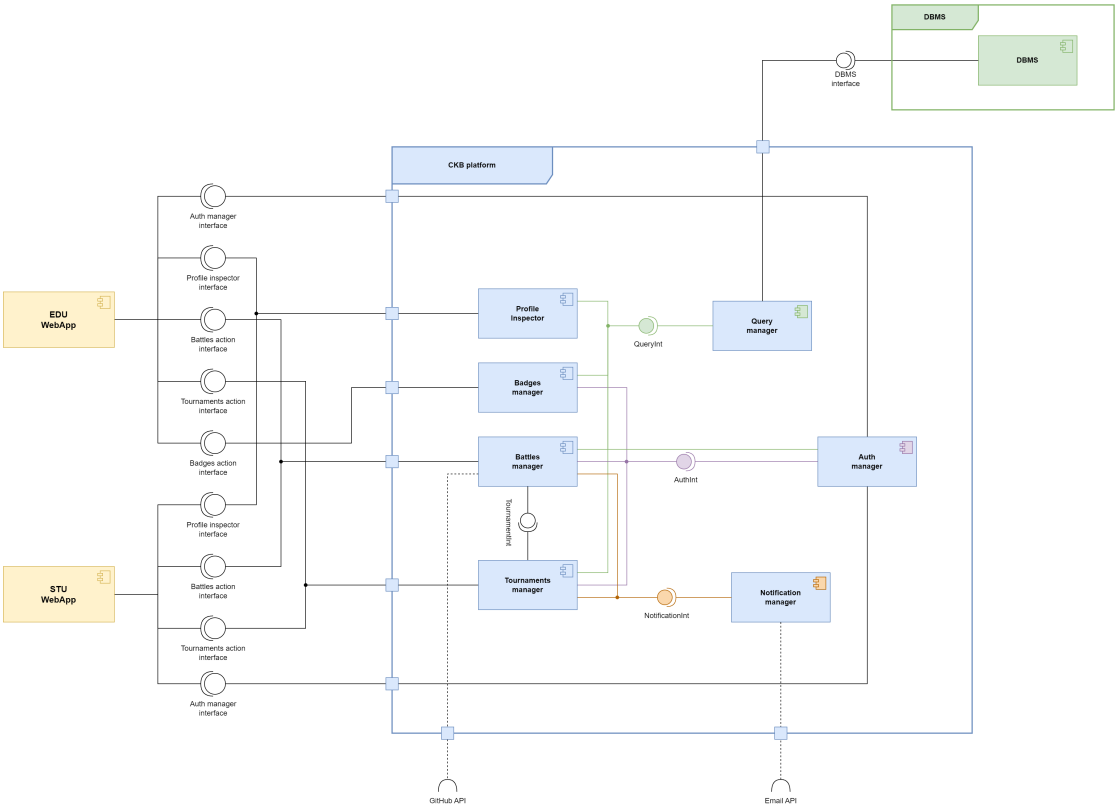

Figure 2.2: Component diagram

In order to maintain the readability of the diagram, the interfaces have been grouped with respect to their functionality; the complete set of endpoints is available later in the document.

2.2.1 Client components

The client components are represented by only a single component, a user-friendly WebApp, that will behave differently depending on the type of the user that is using it (whether it is an EDU or a STU).
The WebApp is interfaced with the server components through all the APIs offeded by the server (see later for a detailed description of the APIs).

2.2.2 Server components

Query manager

The query manager is the component that handles the queries made by the other components that need to access the database. It is responsible for the execution of the queries and for the communication with the database.
It is interfaced with all the internal models of the system that need to access the database, i.e. all the other components of the system exept for the notification manager.
It is interfaced with the database through the DBMS API, external to the system.

Auth manager

The auth manager is the component that handles the authentication of the users and the authorization of the requests made by the other components that need to access the database with respect to the user that made the request.
It is interfaced with all the internal models of the system that behave differently depending on the level of the user that made the request, i.e. the badges manager, the tournament manager and the battle manager.
It isn't interfaced with any external component.

Notification manager

The notification manager is the component that handles the need of the system to notify the users of some events, such as the start of a tournament or the end of a battle.
It is interfaced with all the internal models of the system that need to notify the users, i.e. the tournament manager and the battle manager.
It is interfaced with the Email API, external to the system.

Badges manager

The badges manager is the component that handles the gamification badges.
It allows:

- the creation of new badges;
- the assignment of badges to the STUs;
- the visualization of the badges assigned to a user (?)

Qui devo interfacciare anche con battle e user? It is interfaced with the auth manager (since the creation of a badge is admissible only for the EDUs) and with the query manager (since it needs to access the database to store the badges).
It is interfaced with the EDU WebApp through the proprer, external to the system.

Tournaments manager

The tournament manager is the component that handles the management of the tournaments.
It allows:

- the creation of new tournaments;
- the closure of the torunaments
- the visualization of the tournaments;
- the excange of admin permission between EDUs;
- the subscription of the STUs to the tournaments;
- the visualization of the scores of the EDUs in the tournaments, and so of the ranking

It is interfaced with the auth manager (to allow and perform different actions depending on the level of the user that made the request), with the query manager (since it needs to access the database) and the notification manager (since it needs to notify the users of the start and the end of a tournament).
It is interfaced with the WebApp, both EDU and STU, through the proper APIs, external to the system.

Battles manager

specificare meglio The battles manager is the component that handles the management of the battles.
It allows:

- the creation of new battles;
- the subscription of the STUs to the battles;
- the visualization of the scores of the teams in the battles, and so of the ranking
- the visualization of the battles;
- the automatic evaluation of the battles;
- the eventual manual evaluation of the battles by the EDUs

It is interfaced with the auth manager (to allow and perform different actions depending on the level of the user that made the request), with the query manager (since it needs to access the database) and the notification manager (since it needs to notify the users of the opening of subscriptions, the start and the end of a bettle).
It is interfaced with the WebApp, both EDU and STU, through the proper APIs and with GitHub (to perform the manual evaluation), all external to the system.

Profile Inspector

The profile inspector is the component that handles the visualization of the profiles of the STUs and, consequently, of the badges that a STU has earned during the battles and the tournaments.
It is interfaced with the query manager (since it needs to access the database). It is interfaced with the WebApp, both EDU and STU, through the proper APIs, external to the system.

2.2.3 Logical description of the data

The data of the system is organized in a relational database, with the following entity-relationship diagram:

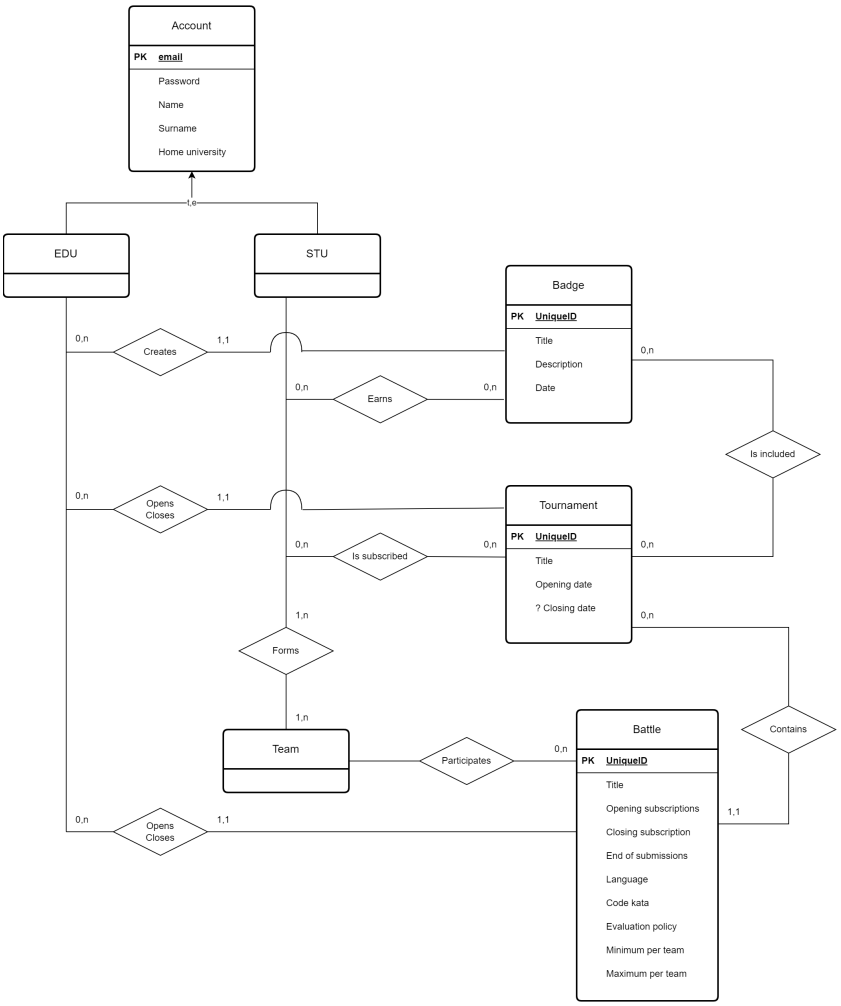

Figure 2.3: Entity-relationship diagram

2.3 Deployment view

The diagram 2.4 shows the deployment view of the system.

Our system comprises two essential components: a static web server and an application server. The static web server serves as the entry point for clients to access the SPA, while the application server furnishes the necessary APIs for the SPA's functionality. To optimize performance, we have opted for distinct solutions for these components.
The static web server will be hosted on a CDN (Content Delivery Network) on cloud, exploiting its edge location caches and reverse proxies to ensure rapid response times. On the other hand, the application server, containing both a business logic layer and a data tier, will find its home on a cloud provider. This decision offers numerous advantages over traditional in-house hosting, including:

- **Scalability and Flexibility**: The cloud infrastructure allows for the dynamic addition or removal of resources like virtual machines, performance cores, or memory as per the evolving needs. Load balancing services further enable the application server to adapt seamlessly to changes in traffic or workload.
- **Security**: Enhanced security features, such as live monitoring and firewalls, contribute to safeguarding the application server against potential data breaches, cyberattacks, and other security threats.
- **Cost-efficiency**: The cloud provider's pay-as-you-go model ensures cost efficiency by charging only for the utilized resources. This approach helps in reducing overall costs, making it a financially prudent choice.

These attributes position a cloud provider as an ideal hosting solution for large, high-traffic applications. The selected cloud provider must respect all these features to effectively meet our system requirements.

mettere la cdn in mezzo, come sulle slides di tiw
The components of the system are explaineed in the following:

- **PC**: Personal computer of the user, it suffices to have a wordking O.S. and a browser installed that supports JavaScript and HTML5 in order to use the system.
- **CDN**: As said before, the CDN is used to host the static web server, that serves as the entry point for clients to access the SPA. It will allow it to be downloaded without affecting the performance of the main application server. The SPA is static and all of its code is run on the client's machine, so there is no need for any logic to be implemented on the CDN side.
- **Cloud provider**: The cloud provider is used to host the application server. It will allow the system to be scalable and flexible, to be secure and to be cost-efficient. It is composed by:

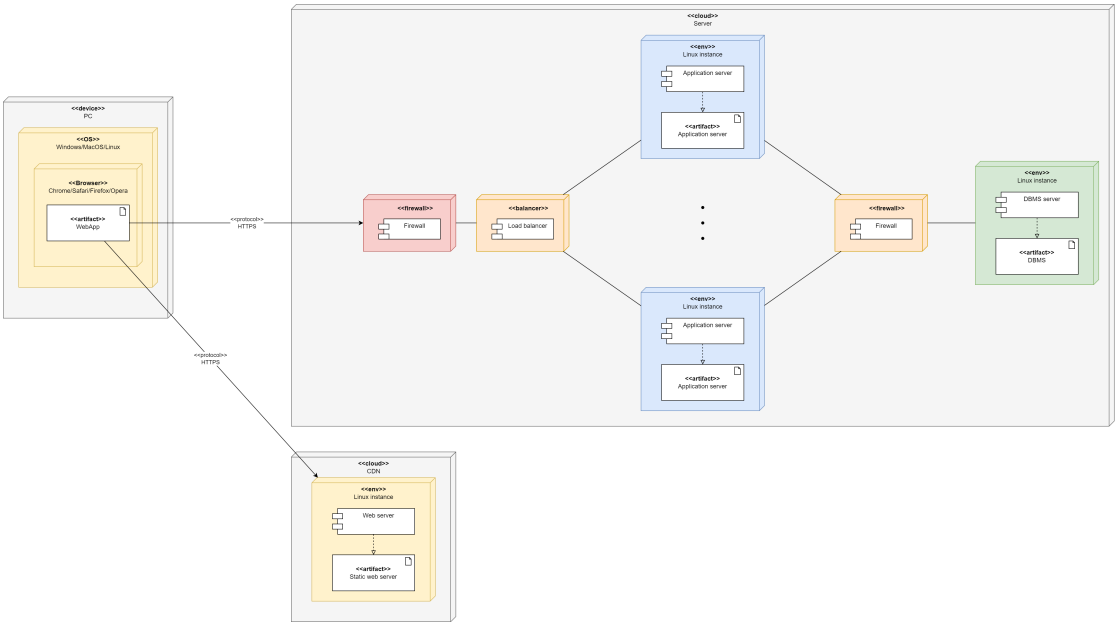

Figure 2.4: Deployment diagram

- **Load balancer**: The load balancer is used to distribute the traffic between the different instances of the application server. It is used to make the system scalable and flexible.
- **Application servers**: The application servers are used to host the application server. They will be in an array of instances, so that the load balancer can distribute the traffic between them. The number of instances can be changed dynamically, so that the system can adapt to the traffic. They are used to make the system scalable and flexible.
- **DBMS server**: The DBMS server is used to host the database.
- **Firewalls**: They are used to make the system secure, and are placed between the load balancer and the external world and between the application servers and the DBMS server. They provide an additional layer of security by blocking or allowing traffic based on predetermined rules. This helps to protect the system from unauthorized access or malicious attacks

Da qui in avanti in questo capitolo dobbiamo prima parlarne un secondo su cosa metter (secondo me la sezione *API endpoints*, visto che c'è prevalentemente server-side, dovrebbe scriverlo chi si occuperà del backend)

2.4 Runtime View

This section describes the most important components interactions of the system.

2.4.1 User login

At the beginning the end user must log in to use the main functions of the application. The login is done by enetring the email and password. If the credentials are present in the database and correct the process will be successful and it will be able to open its dashboard, otherwise it will have to repeat the procedure. (Figure 2.5)

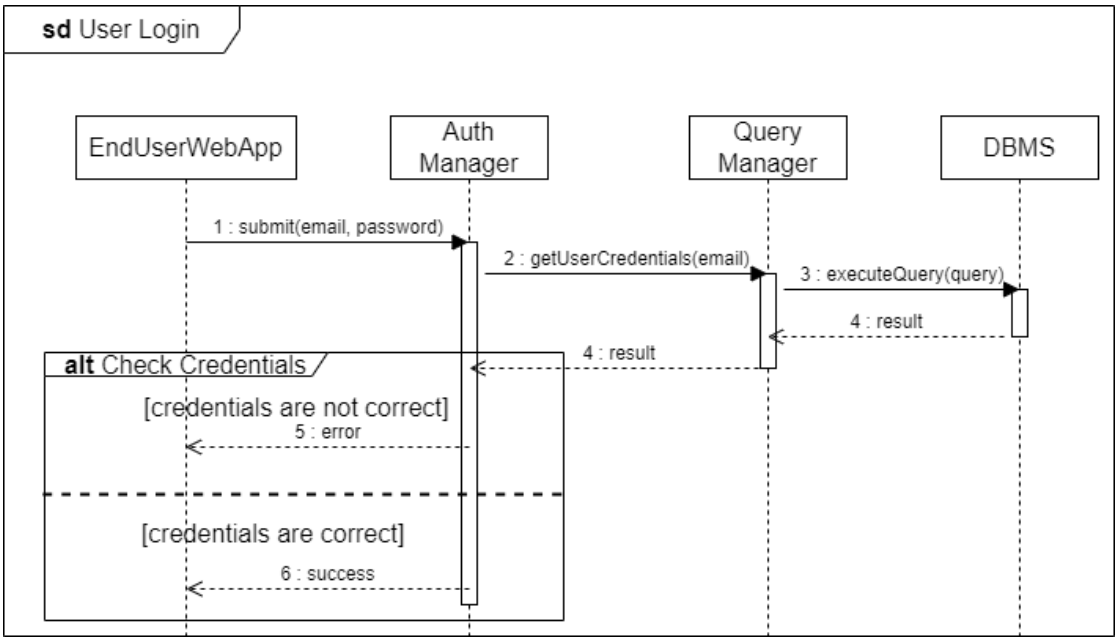

Figure 2.5: Runtime View of User Login Event

2.4.2 Create a tournament

The following sequence diagram is used to explain how to create a tournament. The end user from its device can create a tournament entering the correlated details through the related component, which are sent to the database. If the tournament has been created successfully, the correlated page is created.(Figure 2.6)

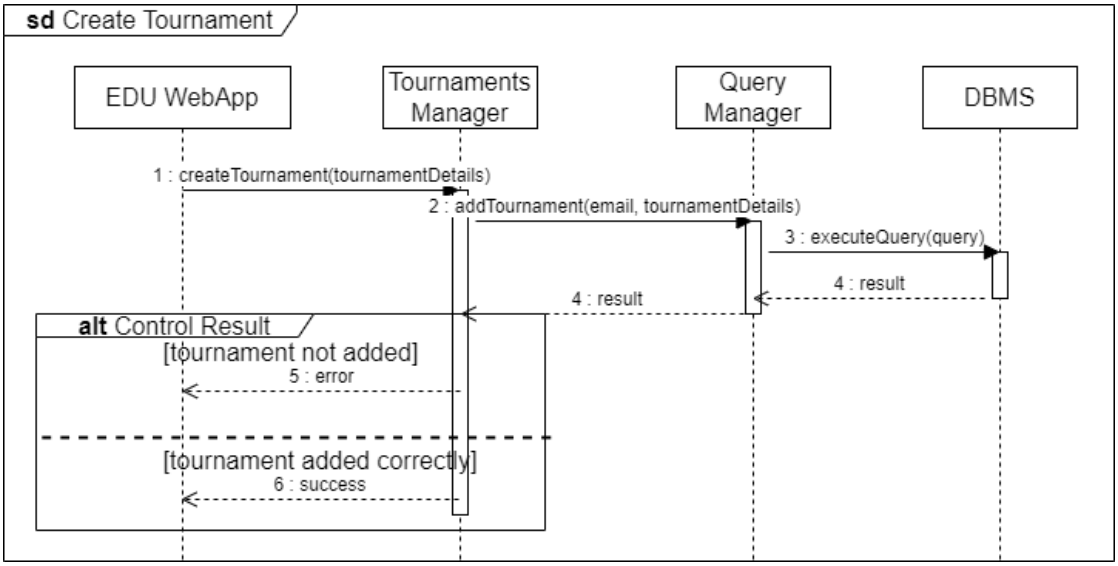

Figure 2.6: Runtime View of Create a Tournament Event

2.4.3 Create a battle

The following sequence diagram is used to explain how to create a battle within a tournament. The end user from its device can create a battle entering the correlated details through the related component. If the user is a granted educator for the current tournament, then the details of the new battle are sent to the database and if the battle has been created successfully, the correlated page is created and all the users subscribed to the correlated tournament are notified. (Figure 2.7)

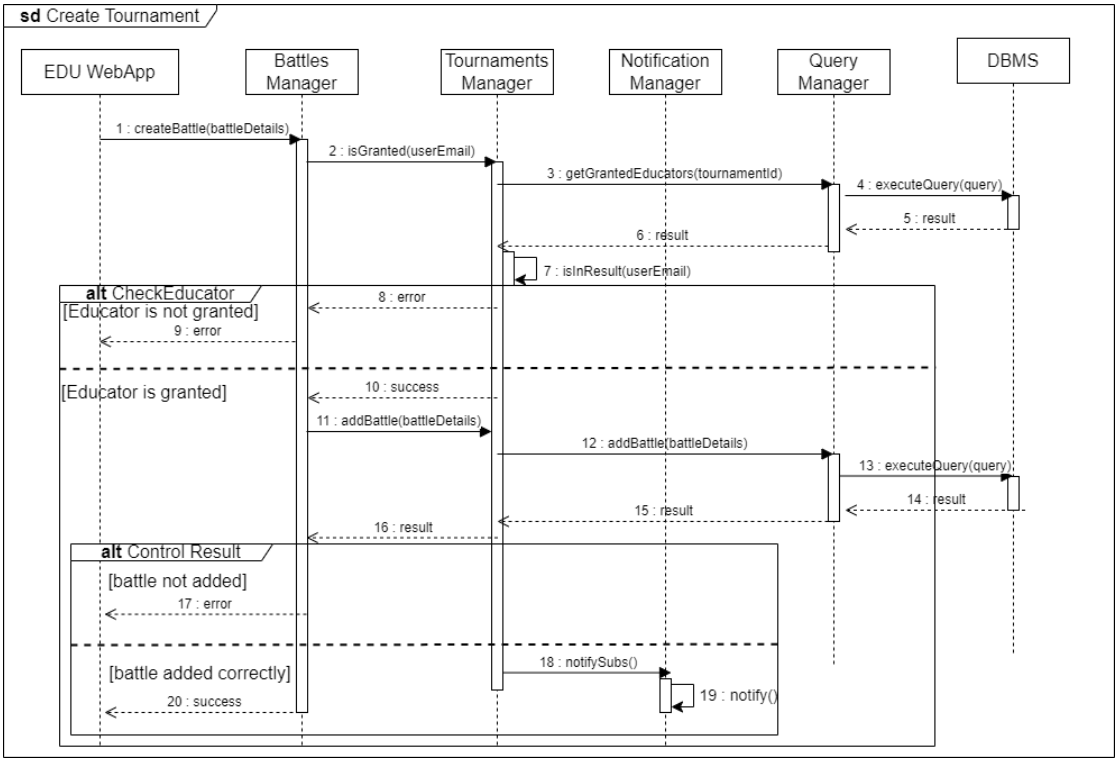

Figure 2.7: Runtime View of Create a Battle Event

2.4.4 Join a Battle Solo

In this sequence diagram is shown how an user can subscribe to a battle. By joining the battle, the system add to the database the email of the user into the subscribed email of that battle. If the user decides to invite other members, the inserted emails are notified. (Figure 2.8)

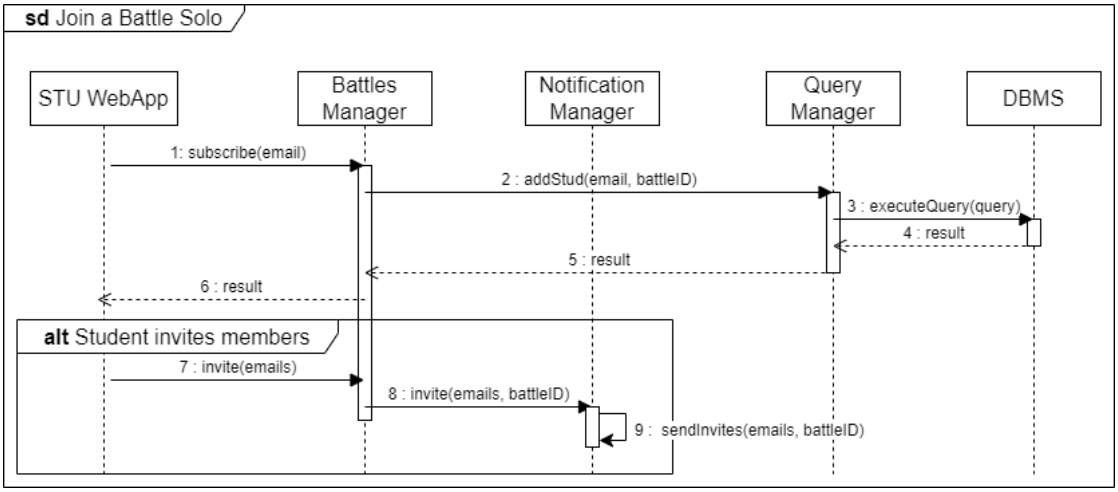

Figure 2.8: Runtime View of joining a Battle Solo

2.4.5 Upload code

In this case, the GitHub API notify the system which through the related component computes the new score and update it by sending the new score to the database. Finally, the system updates the battle ranking with the new score. (Figure 2.9)

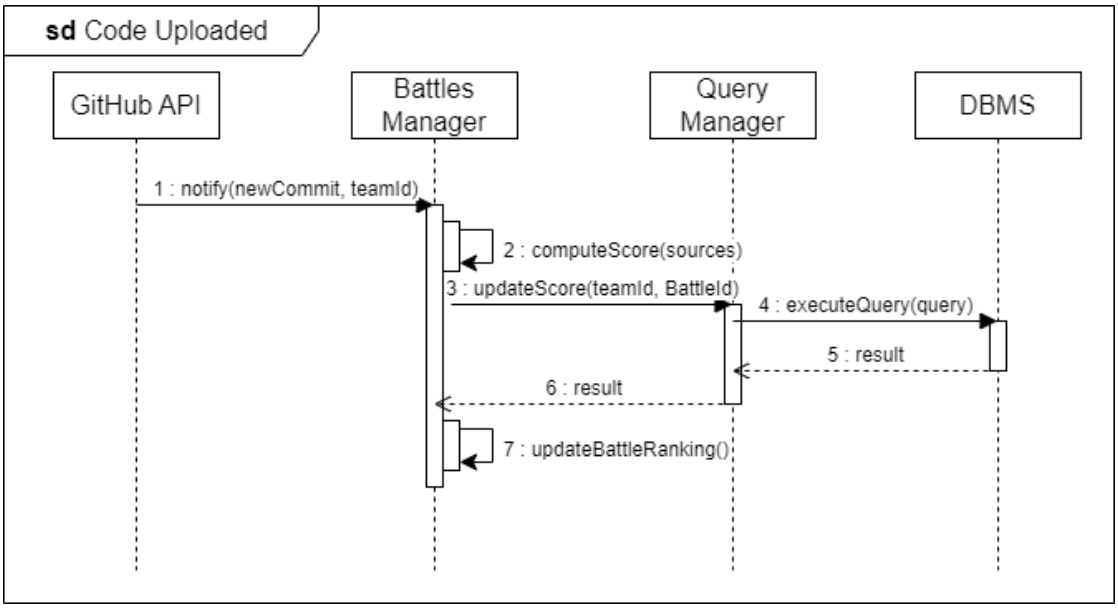

Figure 2.9: Runtime View of Notification of a new commit of a team

2.5 API endpoints

The following section describes one by one the endpoints of the APIs offered by the system.
Qui vi lascio tutte le api con le relative informazioni che mi sevono, poi i path per raggiungerle gestiteveli voi come vi è più comodo e fatemi sapere

Common APIs

2.5.1 Search for a STU

users.js
This endpoint allows the anyone who writes in the searchbox to search for a STU by name, surname or email.

Method: **GET**
URL Parameters:

- `query`: the string that the user has written in the searchbox

Response: JSON array

```
[
    {
        "id": Number,
        "name": String,
        "surname": String
    },
    ...
]
```

EDU WebApp

2.5.2 Get owned tournaments

ownedTournaments.js
This endpoint allows the EDU to get the list of the tournaments that it owns, it will be accessed every time the EDU opens the dashboard.

Method: **GET**
URL Parameters:

- `id`: the id of the EDU that is requesting the list of the tournaments

Response: JSON array

```
[
    {
        "id": Number,
        "name": String,
        // first name of the tournament creator
        "first_name": String,
        // last name of the tournament creator
        "last_name": String,
        // if the tournament is open (true) or closed (false)
        "active": Boolean,
    },
    ...
]
```

2.5.3 Get all tournaments

allTournaments.js
This endpoint allows the EDU to get the list of all the tournaments on the CKB platform, it will be accessed whenever the EDU clicks on the "Show all tournaments" button on the dashboard.

Method: **GET**
URL Parameters: none
Response: JSON array (same as the previous endpoint)

```
[
    {
        "id": Number,
        "name": String,
        // first name of the tournament creator
        "first_name": String,
        // last name of the tournament creator
        "last_name": String,
        // if the tournament is open (true) or closed (false)
        "active": Boolean,
    },
    ...
]
```

2.5.4 Get tournament details

tournamentInfo.js
This endpoint allows the EDU to get the details of a tournament, it will be accessed

when an EDU selects a tournament from the list of the tournaments in the dashboard.

Method: **GET**
URL Parameters:

- `id`: the id of the tournament that the EDU wants to get the details of
- `edu_id`: the id of the EDU that is requesting the details of the tournament

Response: JSON array of one element

```
[
    {
        "id": Number,
        "name": String,
        "language": String,
        "participants": Number,
        "phase": Number,
        "remaining": String(Date),
        "battles": [
            {
                "id": Number,
                "name": String,
                "language": String,
                "participants": Number,
                "phase": Number[1 - 4],
                // Time remaining in the current phase
                "remaining": String(Date)
            },
            ...
        ],
        "ranking": [
            {
                "id": Number,
                "name": String,
                "points": Number
            },
            ...
        ]
    }
]

/*
    phases of the battles:
    1 - Registration
```

```
    2 - Code Submission
    3 - Manual evaluation
    4 - Finished, closed
*/
```

2.5.5 Get battle details

battleInfo.js
This endpoint allows the EDU to get the details of a battle, it will be accessed when an EDU selects a battle from the list of the battles in a tournament view.

Method: **GET**
URL Parameters:

- `id`: the id of the battle that the EDU wants to get the details of
- `edu_id`: the id of the EDU that is requesting the details of the battle

Response: JSON array of one element

```
[
    {
        "id": Number,
        "title": String,
        "description": String,
        "language": String,
        "opening": String(Date),
        "registration": String(Date),
        "closing": String(Date),
        "min_group_size": Number,
        "max_group_size": Number,
        "link": String,
        "phase": Number[1 - 4],
        "tournament_name": String,
        "tournament_id": Number,
        // Could be reduntant with phase, but it's easier to
           check
        "manual": Boolean,
        "ranking": [
            {
                "id": Number,
                "name": String,
                "score": Number
            },
            ...
```

```
        ]

    }
]
```

2.5.6 Get the list of codes to evaluate

mannualEval.js
This endpoint allows the EDU to get the list of the codes that need to be evaluated, it will be accessed when an EDU clicks on the "Manual evaluation" button in the battle view, if it is in the manual evaluation phase.

Method: **GET**
URL Parameters:

- `id`: the id of the battle that the EDU wants to get the list of the codes to evaluate of

Response: JSON array

```
[
    {
        "id": Number,
        "team": String,
        "score"?: Number
    },
    ...
]
```

2.5.7 Evaluate code

code.js
This endpoint allows the EDU to evaluate a code, it will be accessed when an EDU clicks on the "Evaluate" button from the table of the codes to evaluate.

Method: **GET**
URL Parameters:

- `group_id`: the id of the group that the EDU wants to evaluate
- `battle_id`: the id of the battle that the EDU wants to evaluate

Response: JSON object

```
{
    "group_id": Number,
```

```
    "battle_id": Number,
    "language": String,
    "code": String,
    // Name of the group
    "name": String,
    // Score of the group, null if not evaluated yet
    "score"?: Number,
    // Code of the group
    "code": String,
}
```

2.6 Other design decisions

2.6.1 Web Application

As the platform's primary functionality is closely tied to coding activities, it is expected that users will predominantly employ personal computers, so there is no need to develop a mobile application, which would require a significant amount of additional work.
Instead, the system will be accessible through a web application, that is much easier to develop, maintain and being accessed by the users.

2.6.2 Single Page Application

The system will be developed as a Single Page Application (SPA), that is a web application that interacts with the user by dynamically rewriting the current web page with new data from the web server, instead of the default method of the browser loading entire new pages.
This approach will allow the system to be more responsive and to be more similar to a desktop application, since the user will not have to wait for the entire page to be reloaded every time he performs an action. Furthermore, it will allow the system to be more efficient, since the server will not have to send the entire page every time the user performs an action, but only the data that has changed, leaving the client to render the page.

2.6.3 Relational Database

We selected a relational database for our system design because it is effective at storing structured data, granting data integrity, and providing fast query performance. It can also be easily scaled to handle large amounts of data and support many concurrent users. The database allows us to store and retrieve information efficiently, while also ensuring that the data is accurate and consistent

2.6.4 RESTful API

We have chosen to implement a RESTful API for our system because it is a simple, lightweight, and flexible architecture that is easy to understand and use. It is also scalable and reliable, making it ideal for our application.

3. User Interface Design

Since the mockups of the user interface have already been presented in the RASD document, we will not repeat them here. Instead, we will present the UX diagram

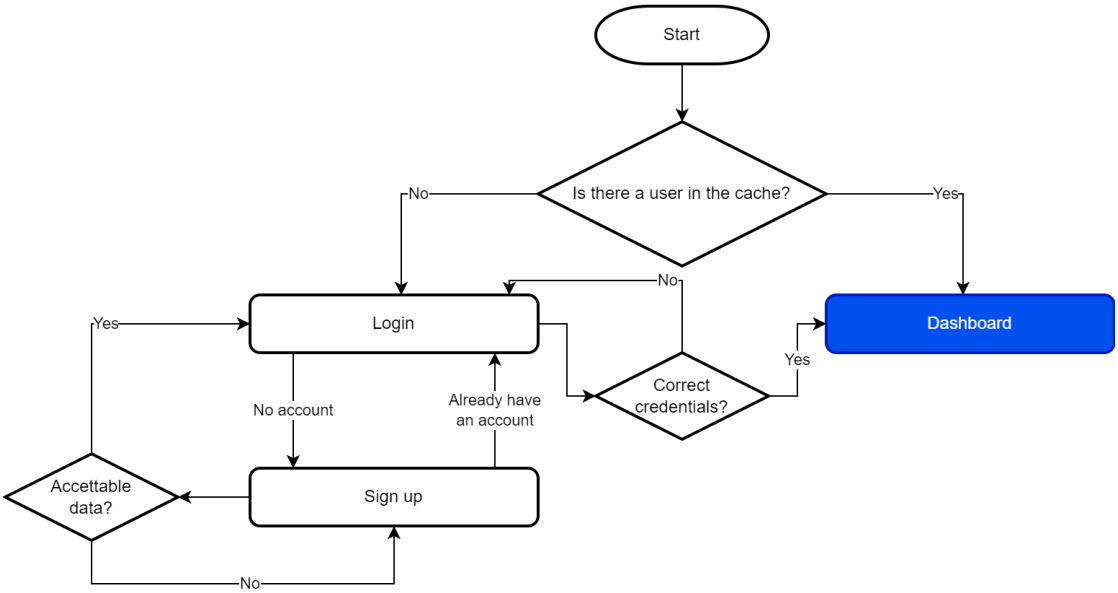

Figure 3.1: Undifferentiated: Entriy point i.e. the login page

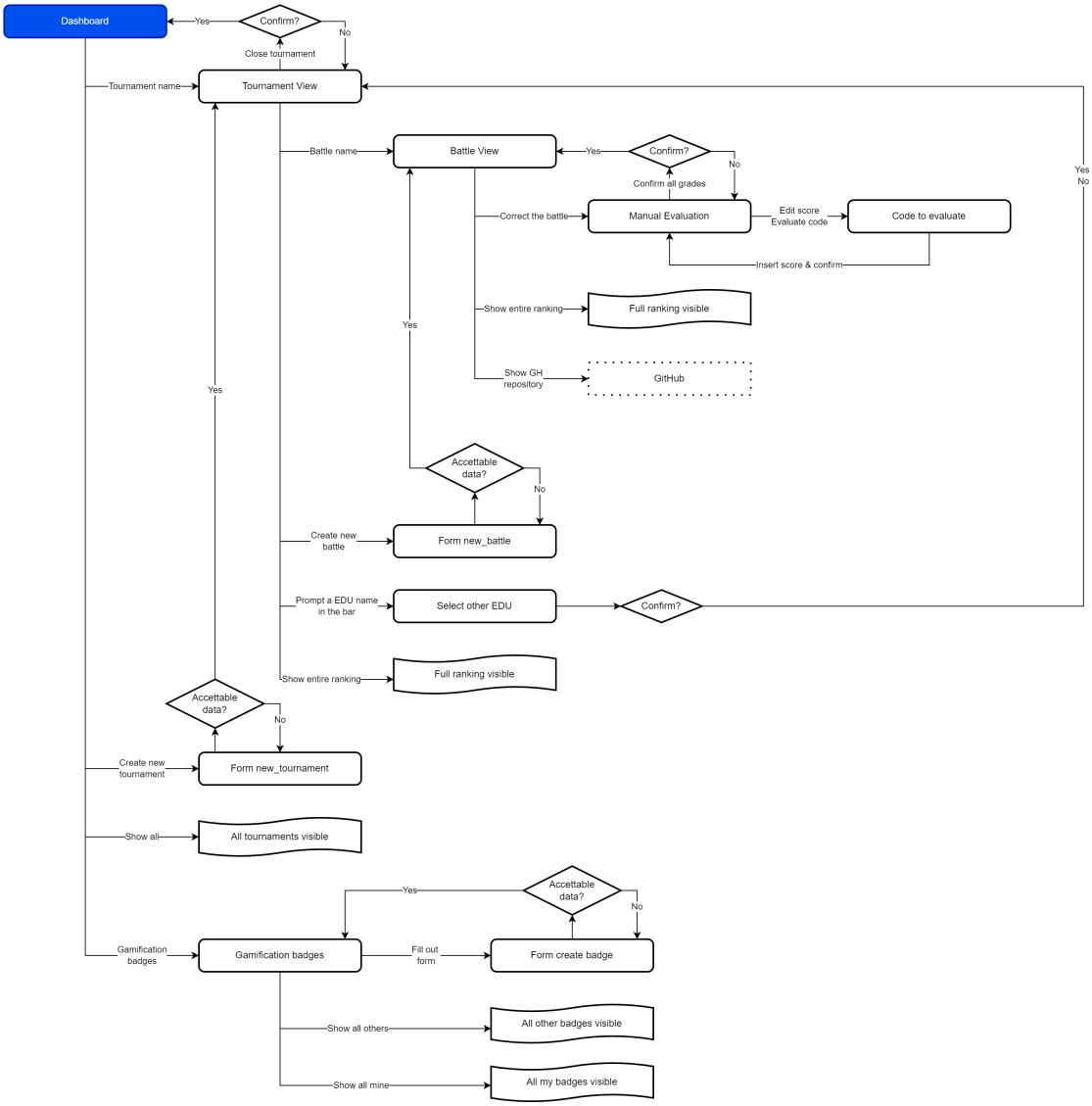

Figure 3.2: Flow graph for the EDU

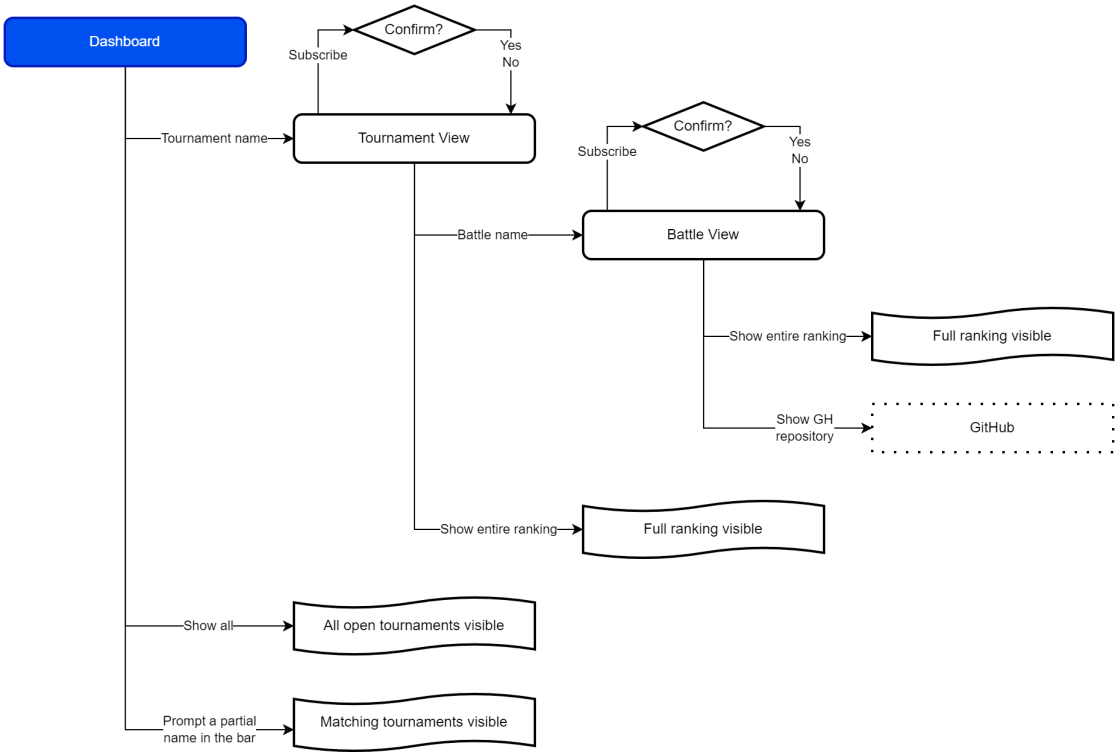

Figure 3.3: Flow graph for the STU

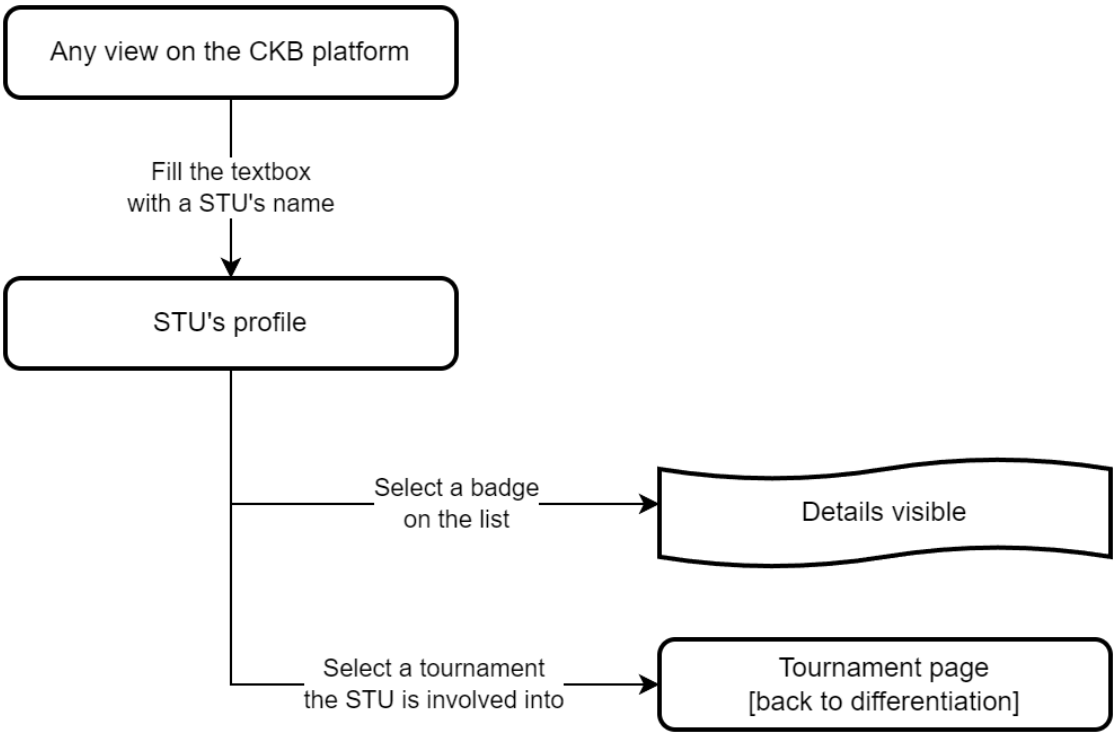

Figure 3.4: Undifferentiated: viewing a STU's profile

4. Implementation, Integration and Test Plan

The application is based on three logical layers (data, logic and presentation) that can be implemented in parallel.
The whole system can be tested in parallel following a bottom-up approach (see later), this is due to the fact that all the components can be developed independently and then integrated together, and so the testing can be done in an incremental way to evaluate the dependencies between the subcomponents.

4.1 Development plan

Since the front-end relies on APIs provided by the back-end, we will focus more on the back-end development, in order to provide the front-end developers with the APIs they need.

4.1.1 Front-end

Even if it relies on the back-end APIs, the front-end can be developed in parallel with the back-end, since the APIs are well defined and documented; it is sufficient to provide a mock-up of the json objects that will be returned by the APIs to correctly develop the front-end.

4.1.2 Back-end

As said before, the back-end development is the most important part of the project, since it provides the front-end with the APIs it needs.
The order of the development of the subcomponents is given by the dependencies between them, so the order is the following:

1. **DBMS**: it is the core of the data layer, so it is the first component to be developed. It includes the implementation of the e-r diagram given in Figure 2.3

2. **Query manager**: it is the component that provides the APIs to the logic layer that will be used to query the database. It is the second component to be developed

since it relies on the DBMS and it is the only component that can access the database.

3. **Auth manager, Notification manager**: they shall be implemetnted before the other subcomponents because they are used by the other subcomponents to authenticate the users and to send notifications to them. Since they are independent from each other, they can be developed in parallel.

4. **All other subcomponents**: they can be developed in parallel since they are sufficiently independent from each other. If, for any reason, a subcomponent needs to interact with another one, it can see the other one as a black box

4.2 Integration plan

Now it is shown the order in which the subcomponents will be integrated together to form the whole system.
Each component will be uint tested before being integrated with the other components, and after the integration the whole component will be tested.
The order of the integration is given by the dependencies between the subcomponents, so the order is shown by the following graphs (for the sake of simplicity, the graphs show only the dependencies between the subcomponents omitting the interfaces between them):

The first component to be integrated is the DBMS, since it is the core of the data layer.

Figure 4.1: Integration plan for the data layer

The second component to be integrated is the auth manager, since it makes all the system behave differently based on the user's permissions and so we need to ensure that the auth manager works correctly before integrating the other components (especially, to ensure that the query to the databases are done by an authenticated user at the right level).

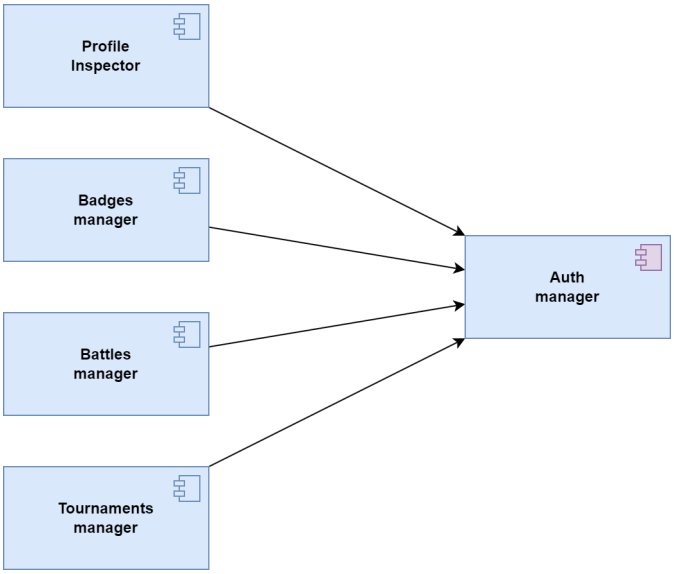

Figure 4.2: Integration plan for the auth manager

The third component to be integrated is the query manager, since it is the only component that can access the database and nearly all the other components rely on it to query the database.

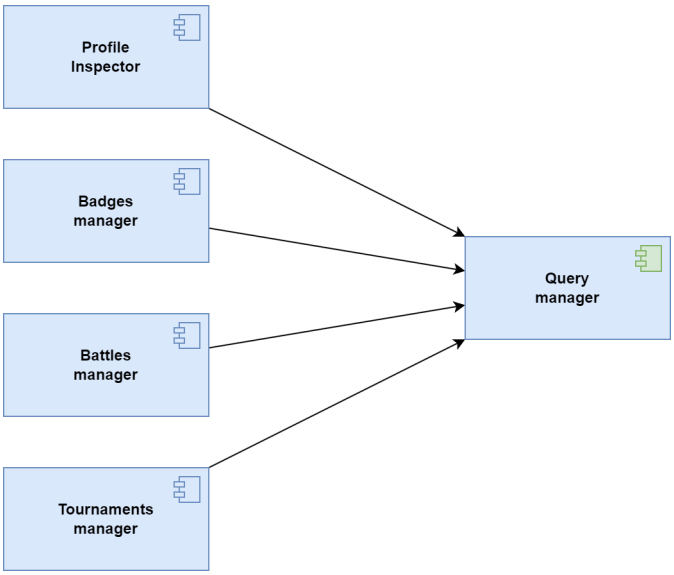

Figure 4.3: Integration plan for the query manager

Another component that needs to be integrated is the notification manager, since it is used by the other components to send notifications to the users.

It can be integrated in parallel with the other Query Manager, since it does not need to access the database, the notifications to be sent are provided by the other components.

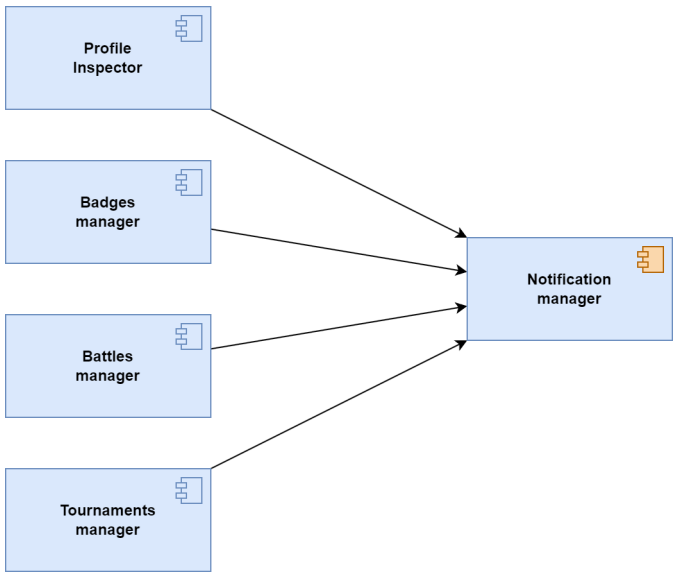

Figure 4.4: Integration plan for the notification manager

Finally, once all the unit tests are passed and the components are integrated together, the presentation layer can be integrated with the other components and a full testing phase can be performed on the whole system.

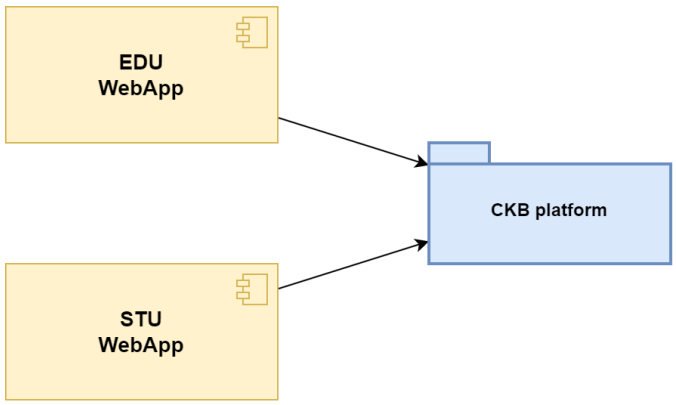

Figure 4.5: Integration plan for the presentation layer

The aim of the testing phase is to verify the functional and non-functional requirements and must take place in a testing environment that is as close as possible to the production environment.

To ensure the minimum probability of failure of the released software, two kinds of testing will be performed:

- **functional testing**: it will be performed to verify that the system meets the functional requirements, in particular, the ones specified on the RASD document.
- **performance testing**: it will be performed to verify that the system meets the non-functional requirements, in particular, wether the software remains functional with increase demand and various environment conditions.

5. Effort Spent

Team

Topic	Time
Division of work	1h

Table 5.1: Effort Spent during team meetings

Tommaso Pasini

Topic	Time

Table 5.2: Effort Spent by Tommaso Pasini

Elia Pontiggia

Topic	Time
Architectural design	4h
User Interface Design	2h
Implementation, integration and test plan	3h

Table 5.3: Effort Spent by Elia Pontiggia

Michelangelo Stasi

Topic	**Time**
Runtime Views	3h30min

Table 5.4: Effort Spent by Michelangelo Stasi