DD:

Design Document

POLITECNICO
MILANO 1863

AY 2023/2024

Alessandro Fornara Simone Colecchia

Prof. Matteo Rossi

Version 1.0

January 3rd , 2024

INDICE

1. INTRODUCTION

A. Purpose

CodeKataBattle is an innovative web platform designed to provide an immersive and visually engaging experience for both learners and educators in the realm of software development and programming.

The main purpose of the platform is to provide a friendly and smart environment in which users can improve their software development skills through competitive coding exercises known as "code kata".

- Intuitive User Experience (UX):

The platform is crafted to ensure an intuitive and user-friendly experience. Through a visually appealing interface, users will find it easy to navigate, explore coding challenges, and track their progress.

- Gamified Learning Environment:

CodeKataBattle transforms learning into engaging challenges. Design elements and interactive features are incorporated to motivate and reward users as they progress through coding exercises.

- Educator-Friendly Tools:

Recognizing the work of educators, the platform plays the role of facilitating teaching. From customizable assignments with automated evaluation to the creation of dynamic coding competitions, the design empowers educators to enhance their teaching methodologies.

B. Definitions, acronyms, abbreviations

B.1. Definitions

- User:
  It is a generic user of the platform, who wants to teach (Educator), learn, or improve coding (Student).

- Battle:
  It is an event with fixed starting and ending time, consisting of a fight among multiple teams, which have to solve a Code Kata and provide their solution.

- Tournament:
  It is a championship in which Students compete and it is composed of several sequential Battles.

- Public Tournament:
  It is a tournament that is visible and open for all the Students subscribed to CKB.

- Private Tournament:
  It is a tournament that is open only to a group of Students, who have been given the permission to visualize it and subscribe.

- Code Kata:
  A kata is an exercise in karate where you repeat a form many, many times, making little improvements in each. Code Kata is an attempt to bring this element of practice to software development. See more at https://codekata.com/.

- Tournament and Battle Final Rank:
  The final rank for a tournament or battle is the rank that is computed and finalized at the end of them. It will be definitive, and no changes will be applied to it.

- Team:
  It is a group of Students created to participate to a single Battle. Each team is composed by a certain number of Students between a minimum and a maximum value (fixed by the creator of the Battle).

- Keyword:
  It is a unique alphanumeric string which is automatically generated by the system to preserve privacy and security. It is used to protect access to Private Tournaments or Teams.

- Automated Workflow:
  A set of actions performed by an actor to automatically retrieve and push/send data whenever some changes are observed.

- Automatic Evaluation:
  It is an automatic computation of a team score performed by the system, based on the provided solution for a Code Kata and considering some criteria (set by the creator of the Battle).

- Manual Evaluation:
  It is a personal additional evaluation for a team score, which is performed manually by an Educator with respect to each member of the team and according to his discretion.

- GitHub:
  Hosting at https://github.com/, it is the most famous platform for software development and version control, allowing users to store, manage and share their code.

B.2. *Acronyms*

- CKB: CodeKataBattle, the name of the platform
- DMZ: De-Militarized Zone, secure network segment acting as a buffer zone between an organization's internal network and the external untrusted network (Internet).
- SPOF: Single Point Of Failure, is a part of a system that, if it fails, will stop the entire system from working.
- DBMS: DataBase Management System, software application that provides an interface and tools for creating, managing, and interacting with databases.
- DAO: Data Access Object, architectural pattern for persistency management. It is an interface used to communicate with the DBMS.

B.3. *Abbreviations*

- **[RVn]** – It points to the n-th runtime view
- **[RRLn]** – It points to the n-th requirement concerning registration and login functionalities.
- **[RTn]** – It points to the n-th requirement concerning tournament functionalities.
- **[RBn]** – It points to the n-th requirement concerning battles aspects.
- **[RGn]** – It points to the n-th requirement concerning GitHub.
- **[RNn]** – It points to the n-th requirement concerning notifications.

C. Revision history

- ✓ **V1.0 –** First version of the document. Revised on 3rd January 2024.

D. Reference documents

- "Systems and software engineering —Life cycle processes — Requirements engineering" - ISO/IEC/IEEE 29148:2018(E).
- "Assignment RDD AY 2023-2024".
- MongoDB documentation - https://www.mongodb.com/docs/
- Unified Modeling Language specifications - https://www.omg.org/spec/UML/

E. Document structure

- **Section 1:** Introduction
  This section offers a brief introduction to the document that is here presented, including all the definitions, acronyms and abbreviations that will be found reading it.

- **Section 2:** Architectural Design
  This section offers a more detailed description of the architecture of the system. The first part provides a high-level description of the system, together with a description of its components. Then, the deployment of the system is presented and described in detail. Finally, the main runtime views of the system are exposed and outlined.

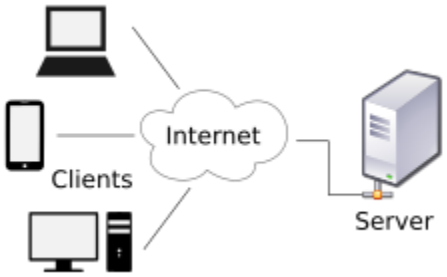
- **Section 3:** User Interface Design
  This section is useful for graphical designers and contains several concepts of the visual appearance of CKB platform, referring to the client-side experience.

- **Section 4:** Requirements Traceability
  This section acts as a bridge between the RASD and DD document, providing a complete mapping of the requirements (functional and non-functional) described in the RASD to the logical modules presented in this document.

- **Section 5:** Implementation, Integration and Test Plan
  The last section is addressed to the developer team and describes the procedures followed for implementing, testing, and integrating the components of CKB: there will be a complete report about how to implement and test them.

2. ARCHITECTURAL DESIGN

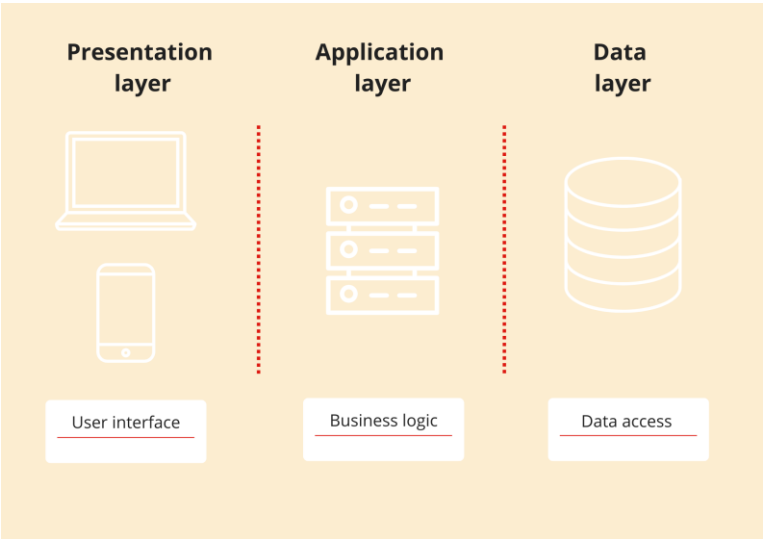
A. Overview

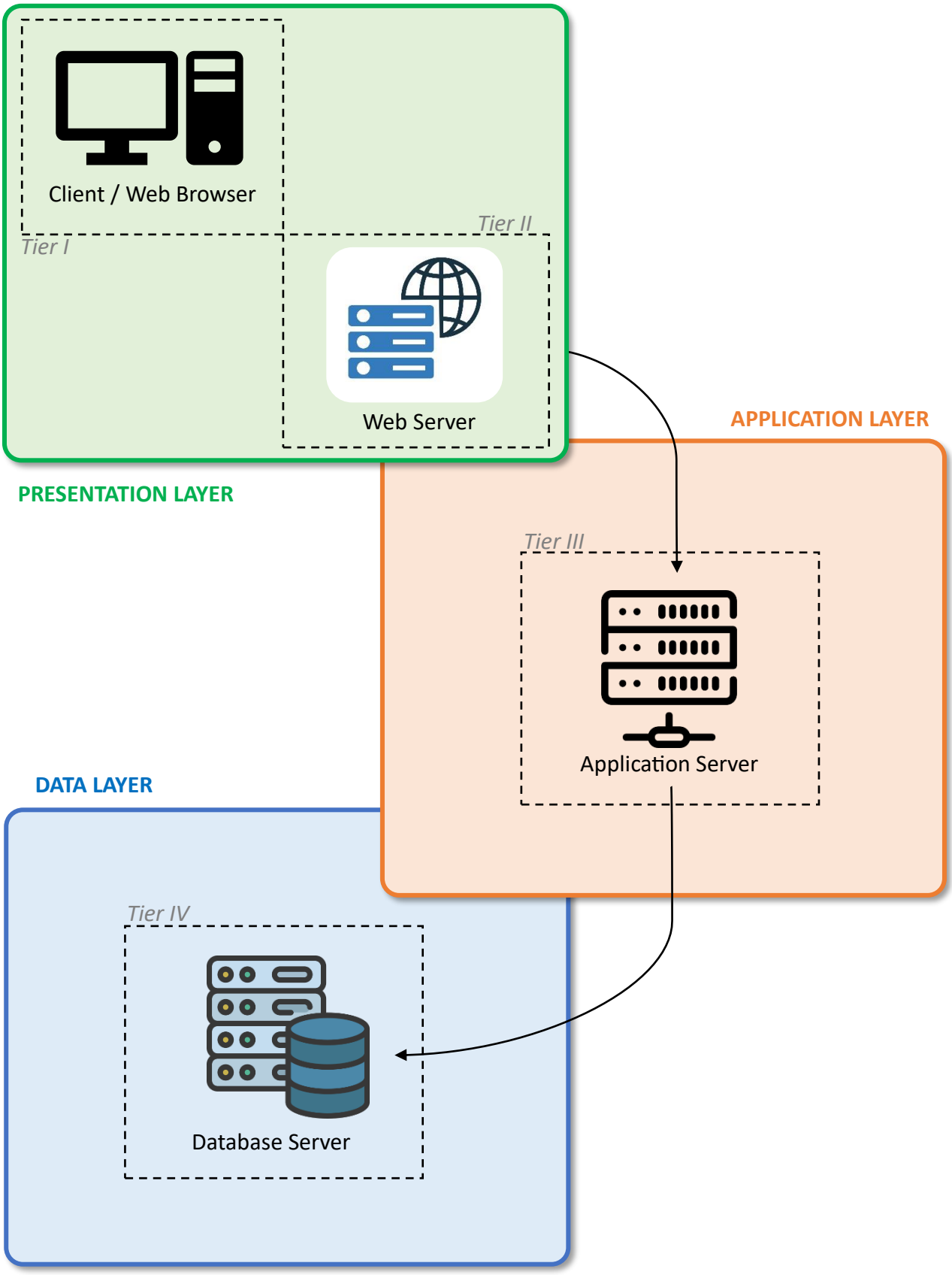


CodeKataBattle is a web application which follows the commonly known client-server paradigm. In particular, the client is *thin*, which means it doesn't manage the application business logic, but only the presentation layer. The server, instead, is *fat* and contains all the data management and business logic.

The system consists of multiple layers responsible for specific functions:

- **Presentation Layer**: focuses on the user interface and interaction.
- **Application Layer**: manages the business logic and data flow between the presentation and data layers.
- **Data Layer**: manages data persistence and database access.




Client / Web Browser
Tier I
Tier II
Web Server
PRESENTATION LAYER
APPLICATION LAYER
Tier III
Application Server
DATA LAYER
Tier IV
Database Server

The system will be designed with a 4-tier architecture (two for the presentation layer, one for the application layer and one for the data layer) with the aim to focus on the following goals:

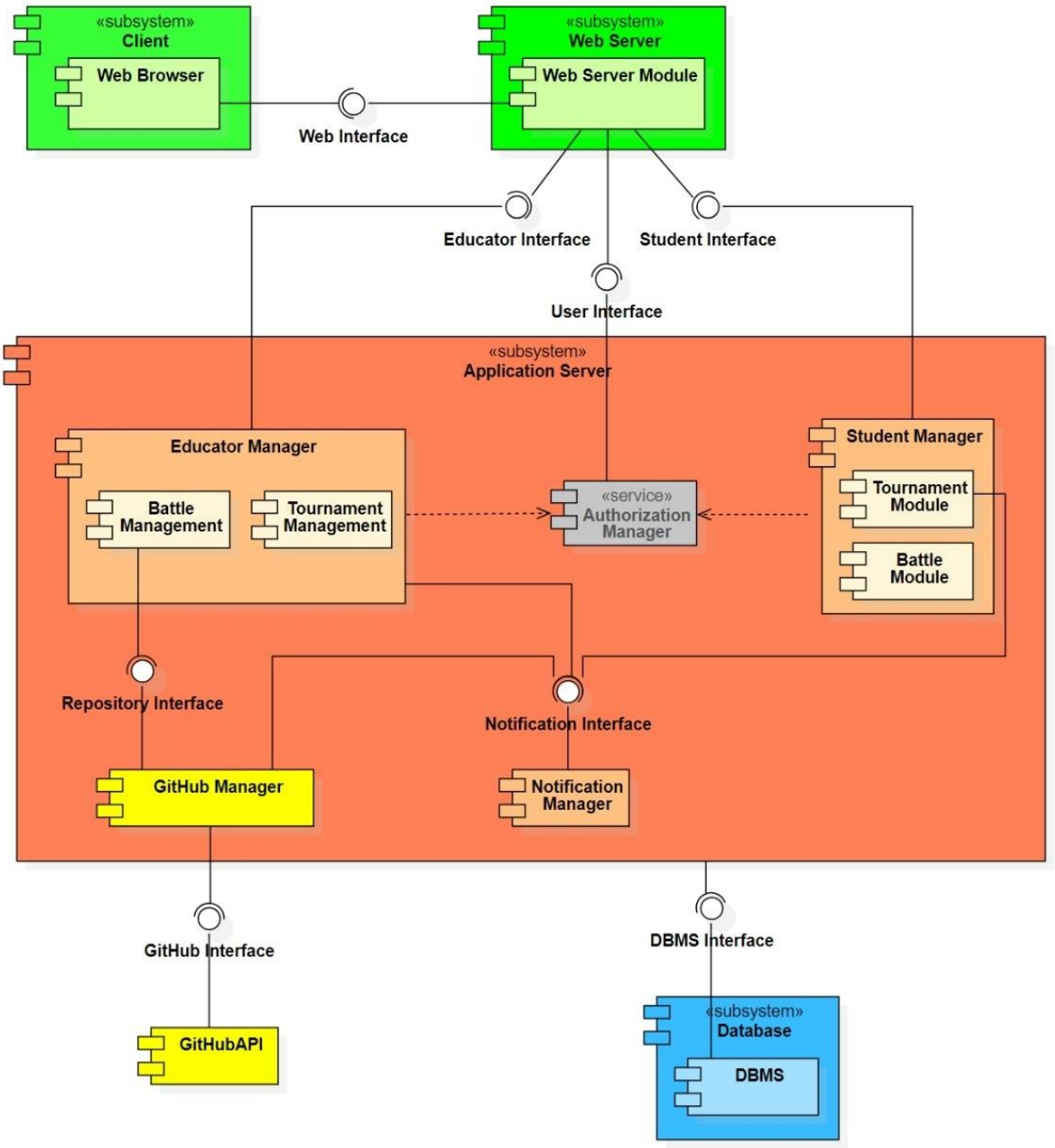
- *Adaptable Scalability.*
  With more tiers, it is easier to vertically scale the system, aligning with the growth of the user population of the application.

- *Flexible technology.*
  Every node can use his own specific technology, without influencing the others. Moreover, this kind of architecture simplifies the upgrade process, which is a valuable attribute considering the nature of the platform.

- *Simplified Maintainability.*
  The division of responsibilities simplifies maintenance. For example, changes at the presentation level can be handled independently of changes to the business logic or data level.

- *Performance Boost.*
  The distribution of functionalities across multiple layers enhances performance by allowing each layer to be optimized independently from the others.

B. Component view



- **Authorization Manager**

This component handles requests regarding authentication. This includes signing up and logging in. Moreover, the aim of this component is to manage authorizations to prevent users from forwarding requests they are not allowed to.

- **Educator Manager**

The Educator Manager serves as the central hub for handling requests from educators. It includes several key sub-components:

- Tournament Management
  This sub-component is dedicated to overseeing various aspects of tournaments. Educators can utilize this module to create new tournaments, manage tournament lifecycles (including closure), and perform other administrative tasks related to tournaments.

- Battle Management
  This sub-component handles requests related to battle management. This includes the creation of battles, the closure of consolidation stages, and managing various phases of battles within the system.

➢ **Student Manager**

The Student Manager is designed to handle requests specifically from students. It includes the following sub-components:

- Tournament Module
  Dedicated to tournament-related activities, this module manages requests such as tournament subscriptions etc..

- Battle Module
  The Battle Module is responsible for the handling of battle and team-related operations. Students can utilize this module for tasks like team creation, joining existing teams, and managing team-related activities.

➢ **Notification Manager**

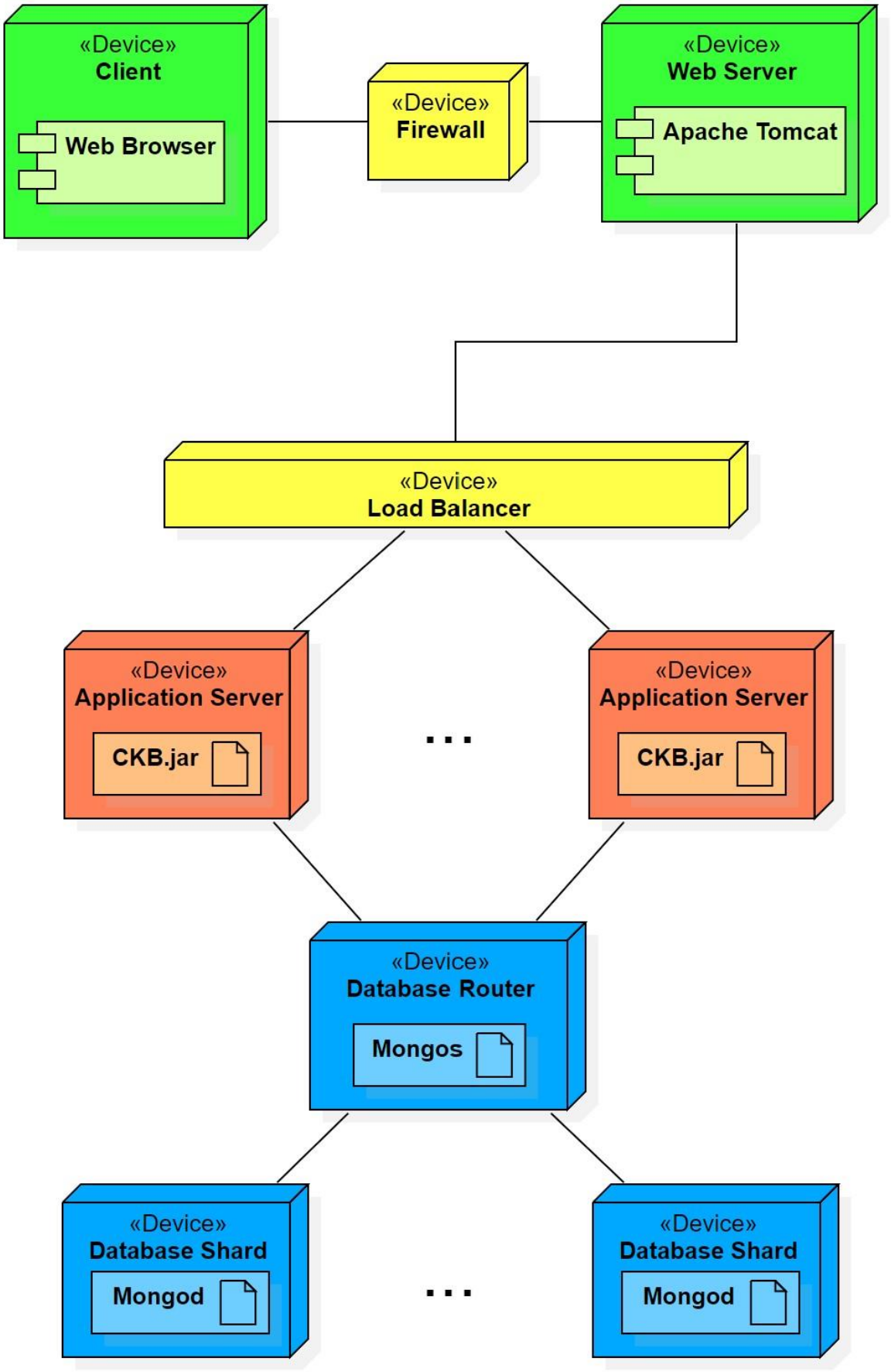
The Notification Manager serves as the centralized component responsible for dispatching notifications to users in response to various system events. Whether it's tournament updates, battle notifications, or other relevant information, this component ensures timely and effective communication with users.

➢ **GitHub Manager**

The GitHub Manager is aimed to handle the logic related to the communication with GitHub. it is responsible for:

- Orchestrating the automated creation of repositories on GitHub. It handles the logic and operations required to set up repositories as needed by Educators.
- Managing the reception and processing of pushed code from GitHub. When code is pushed to repositories, this component ensures the processing of code changes to fulfill point calculation.

C. Deployment view



C.1. Why monolithic approach?

This diagram provides a detailed view of the physical architecture of the system, depicting how the nodes interact and position themselves.

In the previous paragraph, the main components of the system have been described, highlighting the roles they have inside of the Application Server.

Going deeper, it is observable that the latter is not in charge of heavy computational tasks: in fact, most of the functions of the platform are almost completely carried out by the DBMS, since they are just simple data writing/reading operations.

Consequently, the communication-related timing, i.e. the time taken to perceive a request and deliver a response, prevails over the pure computational time, i.e. the time taken to fulfill that request.

This represents the main reason for choosing a monolithic approach for the deployment of the system: the idea is to deploy a single machine in charge of managing the entire logic of the application, processing requests coming from the Web Server and querying the Database Server to answer them.

C.2. Evolution and scaling

By exploiting the monolithic approach, some advantages are obtained.

- The cost of the initial deployment is limited, since the number of machines to be bought and installed is lowered: this could help either in delivering a prototype and in observing the growth of the user base.
- Vertical scalability (scale-up) is facilitated: the system can be easily strengthened vertically by upgrading the single machine of the Application Server. This could be a good compromise to follow the initial growth of the platform with reduced costs.

To fulfill the potential expansion of the platform, the microservices-approach can be reconsidered in the future. In this case, the application layer of the system would be split into different servers, each one managing a different area of the system logic.

The most reasonable deployment for CKB as a microservices-based system might consist in assigning each main component of the Application Server (Authorization Manager, Educator Manager, Student Manager, GitHub Manager and Notification Manager) to a different machine. The communication between different components would be achieved by implementing additional endpoints, according to the RESTful paradigm. In this way the system would be scaled out and optimized.

C.3. Tiers description

As previously mentioned in section 2.A, the system adopts a 4-tier architecture which are described below.

- Tier I. Client

The first tier is dedicated to the client, representing the user interface and the interaction with the application, which is carried out through the web browser. This tier serves as the entry point for users to access and interact with the platform. It includes the visual elements, user experience, and client-side logic that make up the front-end of the application.

- Firewall

It is placed between the Client (Tier I) and the Web Server (Tier II) and protects the system network from external threats coming from the Internet (such as DDoS attacks etc.). In this way, the system exploits a DMZ approach and ensures security and privacy to the users of the platform.

- Tier II. Web server

The second tier is represented by the Web Server, strategically handling incoming requests from clients and managing the dynamic aspects of the application. Together with the Firewall, it plays a key role in ensuring the security of the system: it manages the access to the platform, thus preventing users from exploiting functionalities they are not allowed to.

The diagram shows a single instance of the Web Server but, alongside with a possible evolution of the platform, multiple instances of it can be deployed in order to fulfill a bigger load of requests.

- Load balancer

It is placed between the Presentation Layer and the Application Layer and it has the duty of dispatching, forwarding and distributing the requests coming from the web server to the multiple instances of the Application Server.

Its primary purpose is to optimize resource utilization and improve system performance, but it is also responsible for guaranteeing the availability of the system (especially of the Application Layer): it enhances fault tolerance by automatically rerouting traffic away from failed or unhealthy servers.

- Tier III. Application server

The third tier is dedicated to the Application Server, the core of the system. It contains all the logic of the application, including its main functionalities and features, and realizes CKB experience.

As shown in the diagram, multiple instances of the entire platform business logic are deployed on different machines in order to supply a potentially high throughput of incoming requests. This is made possible thanks to the load balancer which routes them to the correct server, according to its specific strategy.

This kind of deployment, based on the horizontal scalability, is very suitable for following a rapid increase in the platform user base: when the load of requests of each server becomes unbearable, a new Application Server can be deployed with little effort to redistribute the total load of requests.

- Tier IV. Database server

The fourth and last tier of the system deployment is occupied by the Database Server, the core of data management. It oversees storing, managing and updating the data of the platform, thus serving in the most important role within the system.

In the case of CKB, as previously mentioned, most of the functions of the platform are simple read/write operations on data, which are carried out by the DBMS itself. For this reason, it is of paramount importance to have the possibility to perform a very large number of transactions at a time.

Another aspect to be considered is the potential amount of data as a consequence of the growth of the platform. The Database Server should be able to provide a considerable capacity to store all the information effortlessly.

Moreover, it is important to guarantee a high level of availability of data and to assure no losses of information, such that users can rely on their correctness and completeness.

Considering all these aspects, it goes without saying that the Database Server should employ a good partitioning and replication system. The best solution to achieve all the goals described above may be the adoption of the *sharding* technique.

It consists in dividing the database into smaller segments according to a predetermined criterion: they are called *shards,* each one representing an independent portion of the total data of the system. These shards can be distributed on the different nodes of the Database Server, each one communicating with a so called *router*: it receives requests from the Application Server and routes them to the shard that manages the related segment information.

MongoDB is the most famous and used DBMS employing sharding as its partitioning (and replicating) system. Its sharding system is composed of two main characters:

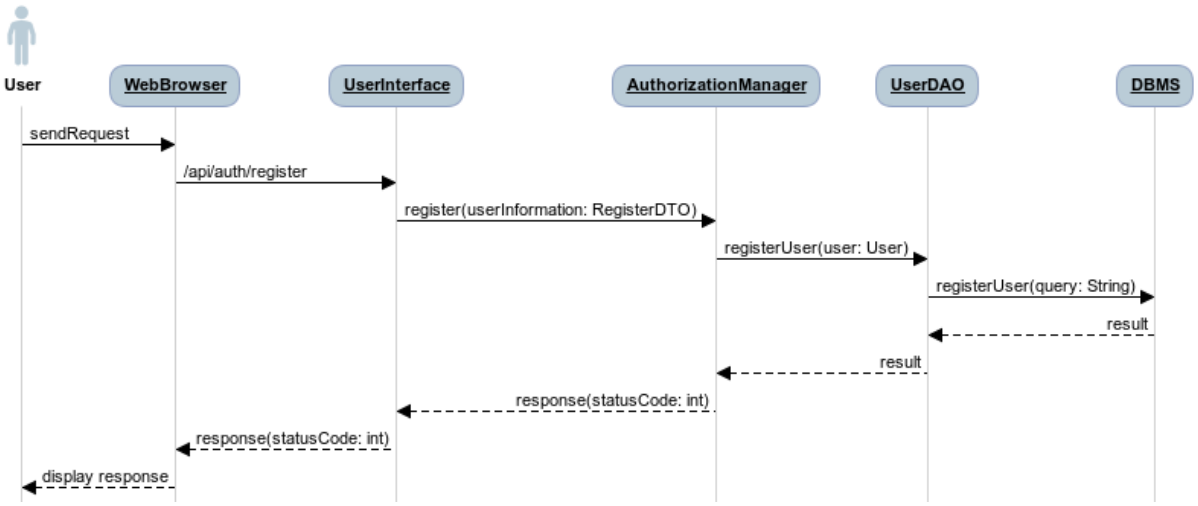
mongos, the routing service that makes the communication between the Application Server and the shards possible, and *mongod*, the daemon process of MongoDB that handles each operation on data.

With the same purpose of avoiding bottleneck formation, *mongos* can be deployed on different nodes enhancing performance and increasing fault tolerance.

D. Runtime view

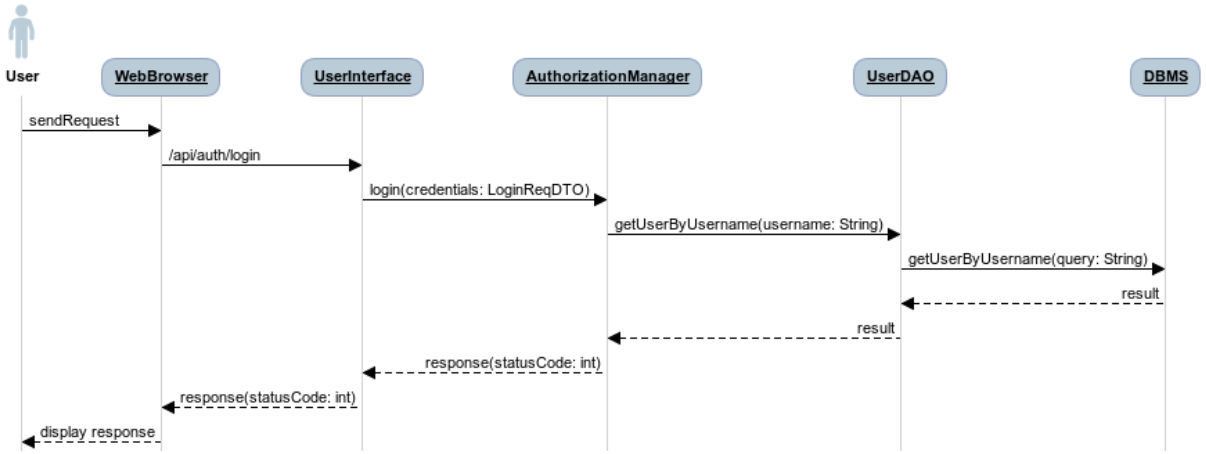
In this section, all the runtime views corresponding to the use cases in the RASD are represented and described.

[RV1] – Register



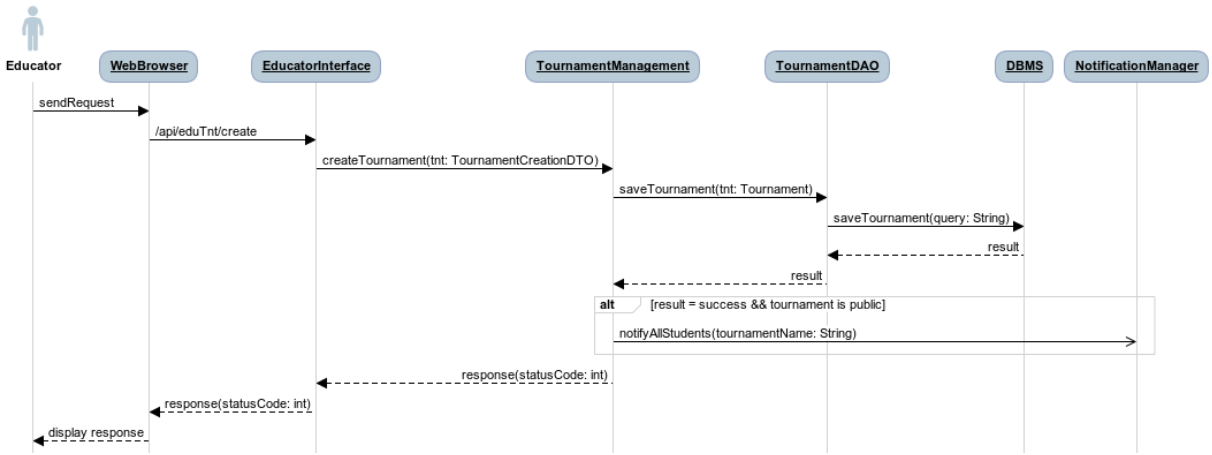
Initially, the user enters their information via a dedicated form and uses the Web Browser to send a request to the server. This request is received by the User Interface and managed through the *register()* method within the Authorization Manager component. This method will attempt to store the new user's data in the database through the UserDAO. If the operation is successful, the user will receive a response with a 200 status code. Otherwise, if the provided email or username already exists in the database, the user will receive a response with a 422 status code.

[RV2] – Login



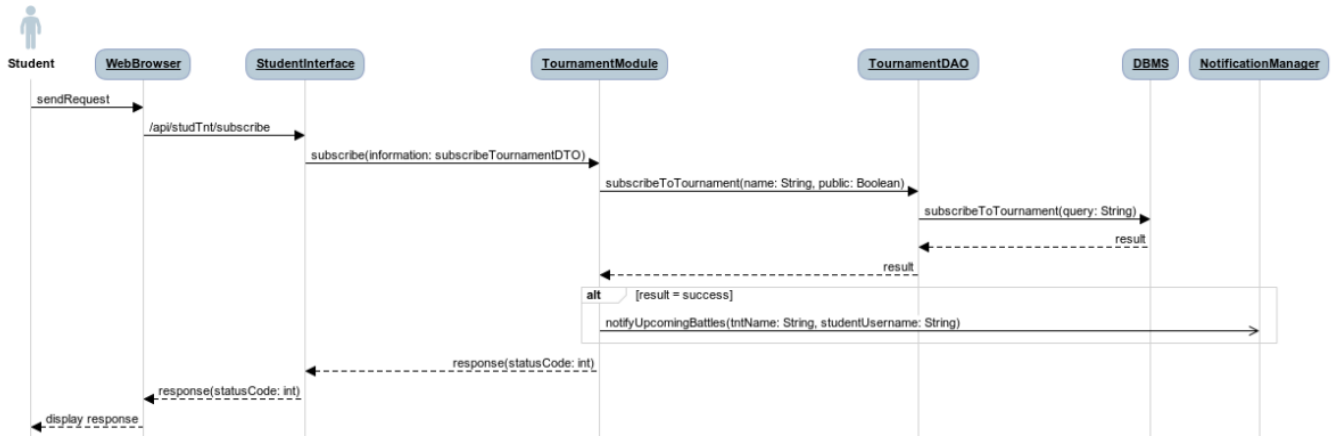
Initially, the user enters their credentials via a dedicated form and uses the Web Browser to send a request to the server. This request is received by the User Interface and managed through the *login()* method within the Authorization Manager component. This method will check the correctness of credentials by checking the database through the UserDAO. If the operation is successful, the user will receive a response with a 200 status code. Otherwise, if the credentials are incorrect, the user will receive a response with a 422 status code.

[RV3] – Create Public/Private Tournament



Initially, the user requests to create a tournament via a dedicated form and uses the Web Browser to send a request to the server. This request is received by the Educator Interface and managed through the *createTournament()* method within the Tournament Management component. This method checks the correctness of the information sent by the Educator (for example if the registration deadline is valid) and then calls the *saveTournament()* method contained in the TournamentDAO which will attempt to save the tournament in the database. If the operation is successful and the tournament is public, the Tournament Management will make an asynchronous call to the Notification Manager and the user will receive a response with a 200 status code. Otherwise, if the name selected for the tournament is not available or the registration deadline is not valid, the user will receive a response with a 422 status code.

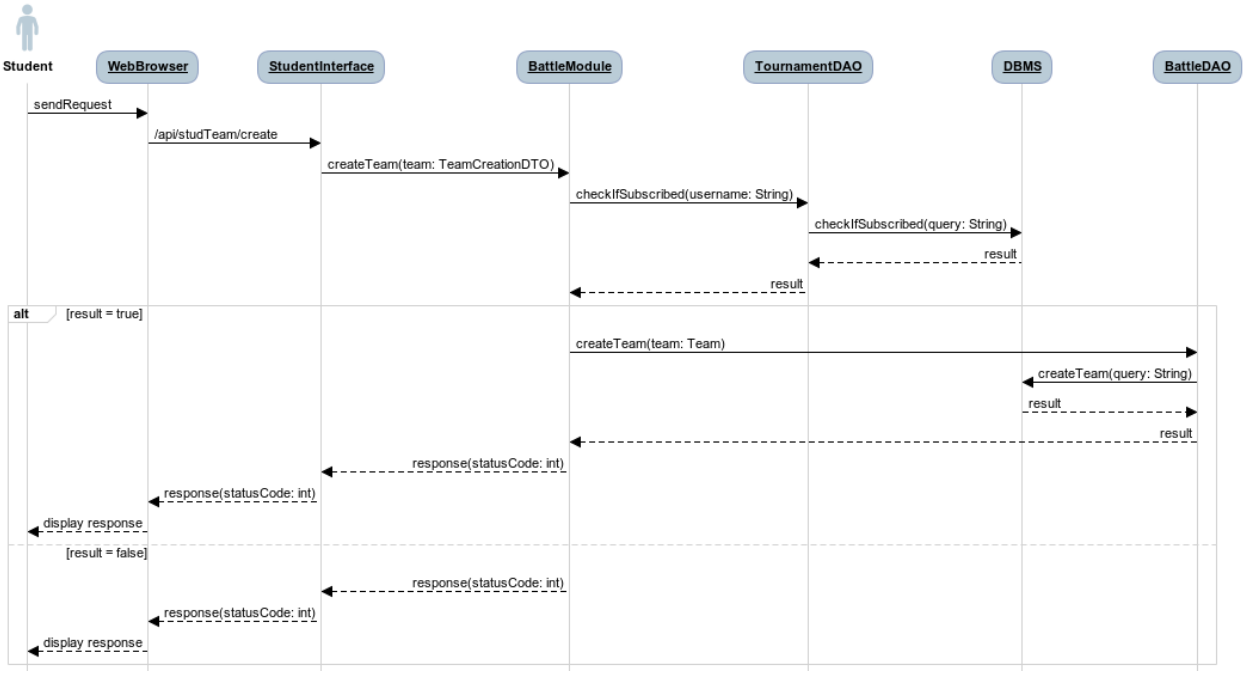
[RV4] – Join Public/Private Tournament



Initially, the user requests to join a tournament via a dedicated form and uses the WebBrowser to send a request to the server. This request is received by the Student Interface and managed through the *subscribe()* method within the Tournament Module component. This method calls the *subscribeTournament()* method contained in the TournamentDAO which will attempt to add the username of the student to the list of subscribed students of the specified tournament in the database. If the operation is successful, the Tournament Management will make an asynchronous call to the Notification Manager to notify the student about upcoming battles and the user will receive a response with a 200 status code. Otherwise, he/she will receive a response with a 422 status code if one of these conditions is found to be true:

- The tournament name doesn't exist in the database (if PUBLIC tournament)
- The keyword doesn't exist in the database (if PRIVATE tournament)
- The tournament registration deadline has expired
- User is already subscribed to the tournament

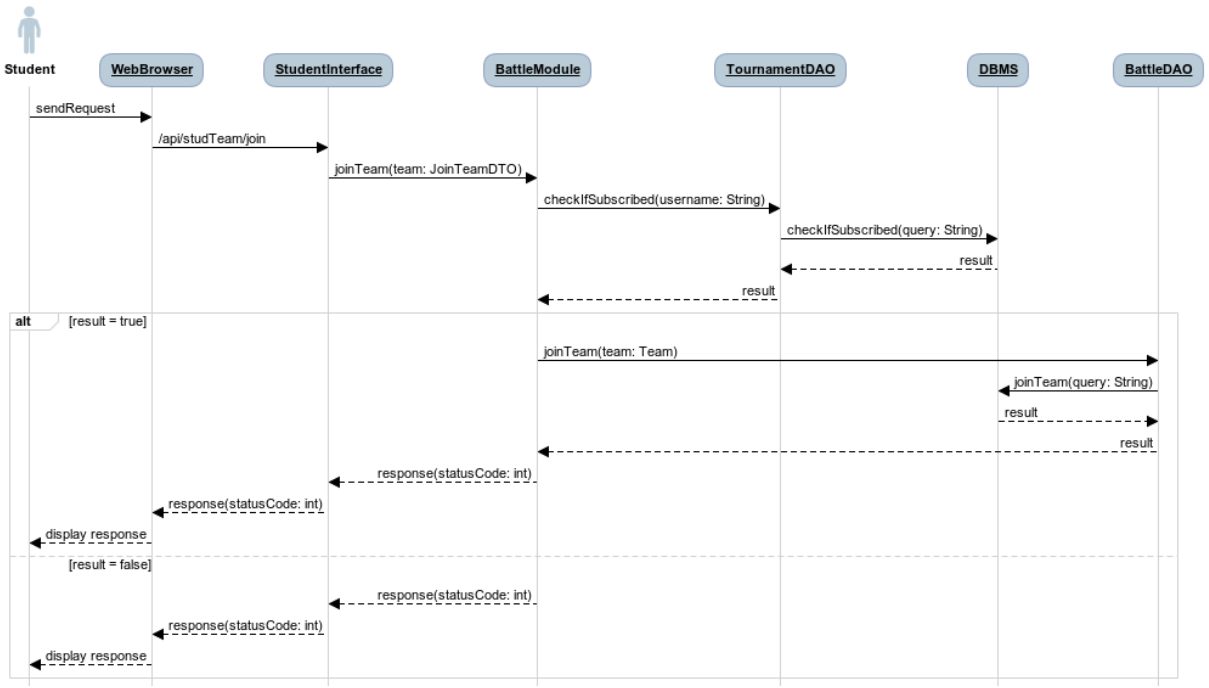
[RV5] – Create Team



Initially, the user requests to create a team via a dedicated form and uses the Web Browser to send a request to the server. This request is received by the Student Interface and managed through the *createTeam()* method within the Battle Module component. This method calls the *checkIfSubscribed()* method contained in the TournamentDAO which will check if the student is subscribed to the tournament that contains the battle he/she is trying to create a team for. If the result is true, then the *createTeam()* method contained in the BattleDAO is called. This method will attempt to add the team to the list of teams subscribed to the battle. If the operation is successful, the user will receive a response with a 200 status code. Otherwise, he/she will receive a response with a 422 status code if one of these conditions is found to be true:

- User is not subscribed to the tournament the battle belongs to
- The battle registration deadline has expired
- User is already a member of another team for that battle
- Team name not available for that battle

[RV6] – Join Team

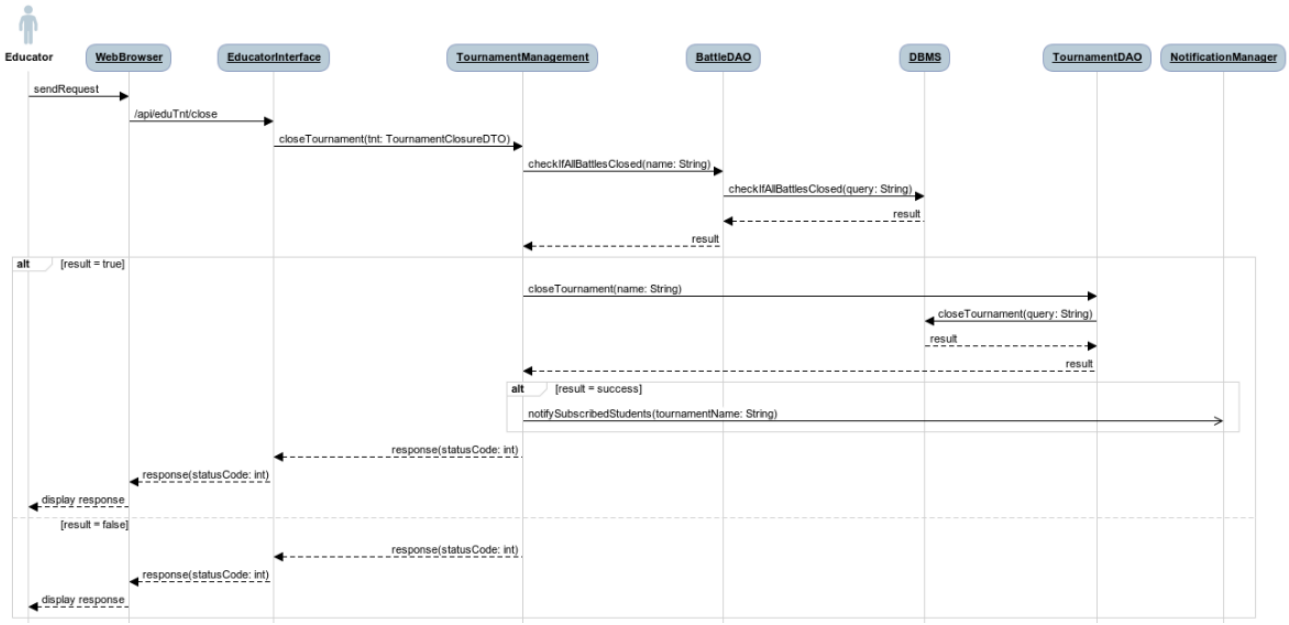


Initially, the user requests to join a team via a dedicated form and uses the Web Browser to send a request to the server. This request is received by the Student Interface and managed through the *joinTeam()* method within the Battle Module component. This method calls the *checkIfSubscribed()* method contained in the TournamentDAO which will check if the student is subscribed to the tournament that contains the battle he/she is trying to join a team for. If the result is true, then the *joinTeam()* method contained in the BattleDAO is called. This method will attempt to add the student to a team subscribed to the battle. If the operation is successful, the

user will receive a response with a 200 status code. Otherwise, he/she will receive a response with a 422 status code if one of these conditions is found to be true:

- User is not subscribed to the tournament the battle belongs to
- The battle registration deadline has expired
- User is already a member of another team for that battle
- Team is full
- The provided keyword doesn't exist in the database

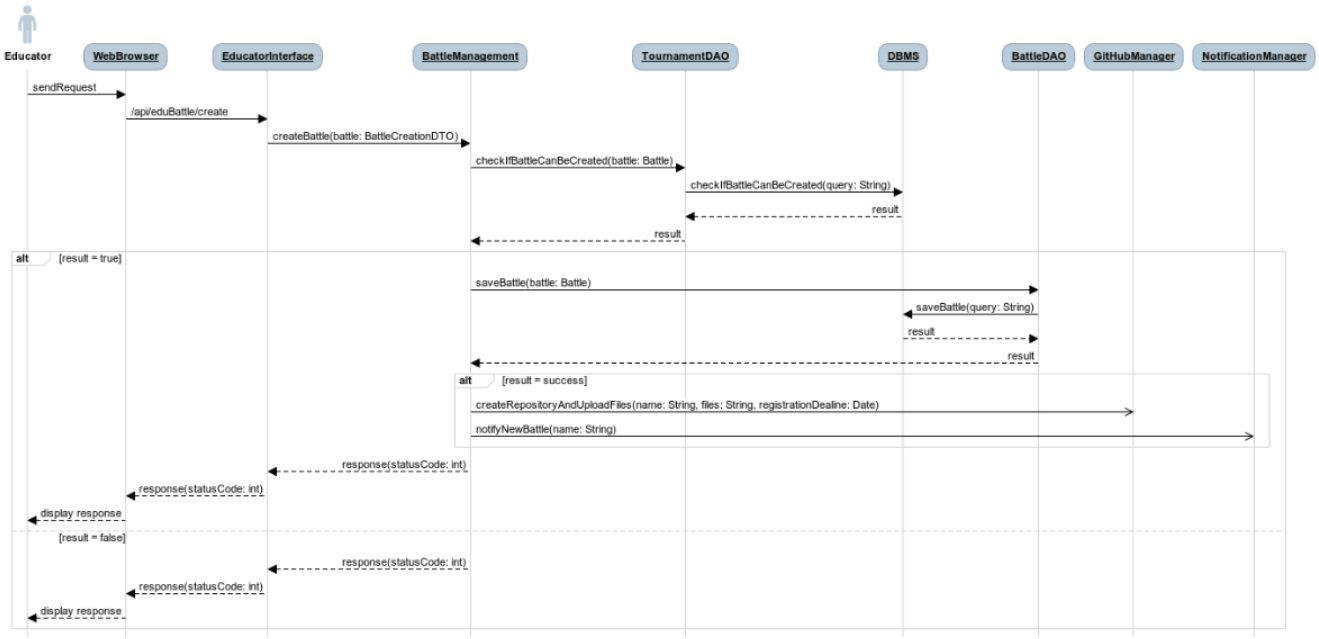
[RV7] – Close Tournament



Initially, the user requests to close a tournament via a dedicated form and uses the Web Browser to send a request to the server. This request is received by the Educator Interface and managed through the *closeTournament()* method within the Tournament Management component. This method calls the *checkIfAllBattlesClosed()* method contained in the BattleDAO which will check if all battles in the tournament are closed. If the result is true, then the *closeTournament()* method contained in the TournamentDAO is called. If the operation is successful, the Tournament Management will make an asynchronous call to the Notification Manager to notify the subscribed students and the user will receive a response with a 200 status code. Otherwise, he/she will receive a response with a 422 status code if one of these conditions is found to be true:

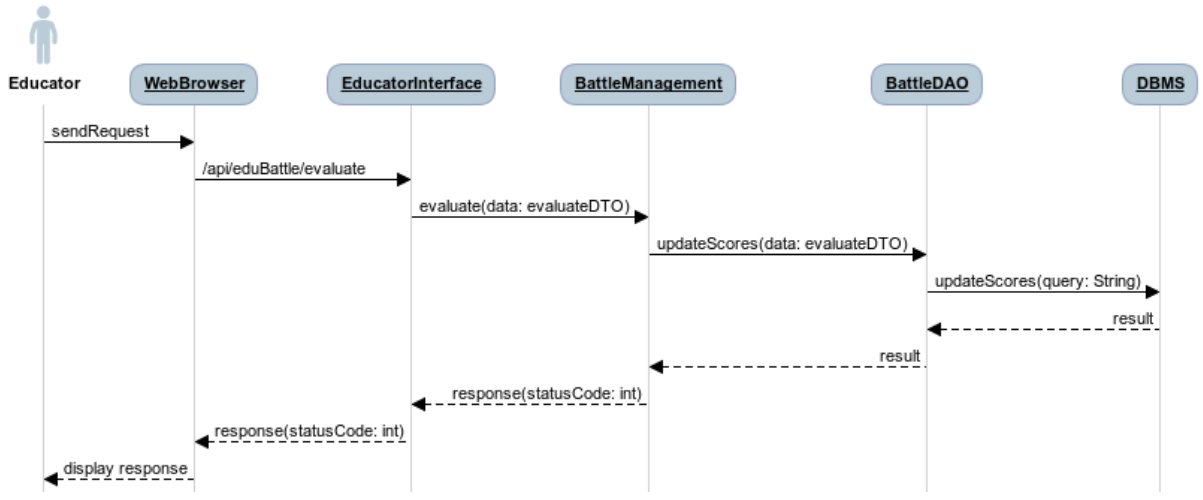
- User is not the admin of the tournament
- There is at least one *Ongoing* battle
- The tournament is already closed or doesn't exist in the database

[RV8] – Create Battle



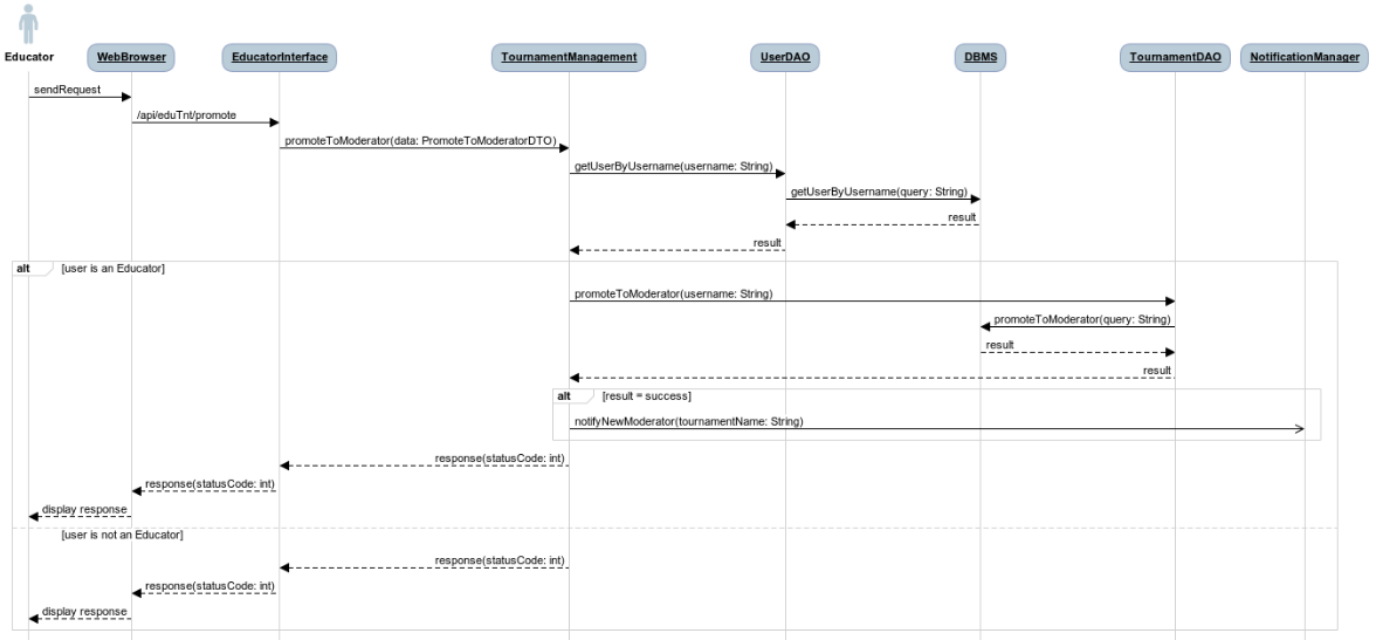
Initially, the user requests to create a battle via a dedicated form and uses the Web Browser to send a request to the server. This request is received by the Educator Interface and managed through the *createBattle()* method within the Tournament Management component. This method checks the correctness of the information sent by the Educator and then calls the *checkifBattleCanBeCreated()* method contained in the TournamentDAO which will check if the tournament is not in the *closed* phase and if the user has the right to create the battle. If the result is true, the *saveBattle()* method contained in the BattleDAO will be called. This method will attempt to save the battle in the database. If the operation is successful, Battle Management will make an asynchronous call to the Notification Manager and to the GitHub Manager to schedule the creation of the battle repository. The user will receive a response with a 200 status code. Otherwise, if one or more parameters are invalid or the tournament is closed, the user will receive a response with a 422 status code.

[RV9] – Evaluate



Initially, the user requests to evaluate a team via a dedicated form and uses the Web Browser to send a request to the server. This request is received by the Educator Interface and managed through the *evaluate()* method which will check if the user is an admin or moderator of the tournament through the TournamentDAO (omitted in the diagram) within the Battle Management component. This method will attempt to update the score of each member of the team through the BattleDAO. If the operation is successful, the user will receive a response with a 200 status code. Otherwise, if the battle is closed or the user is not an admin or educator of the tournament the battle belongs to, the user will receive a response with a 422 status code.

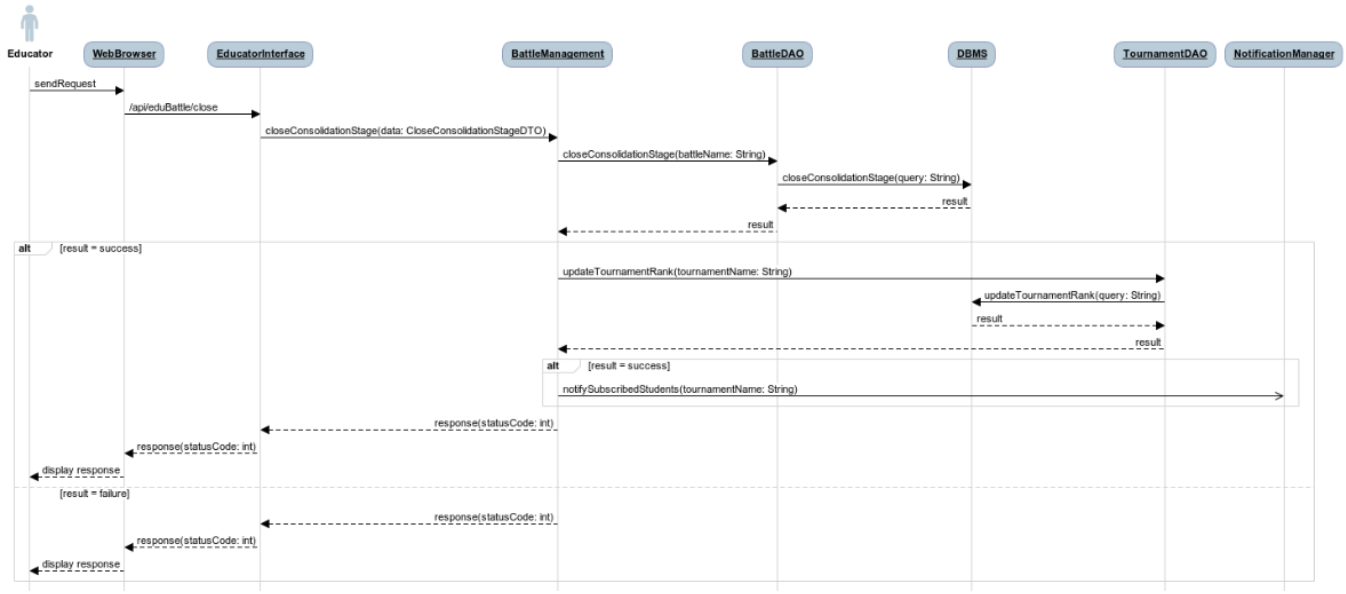
[RV10] – Promote To Moderator



Initially, the user requests to promote a user to Moderator of a tournament via a dedicated form providing a username, and uses the Web Browser to send a request to the server. This request is received by the Educator Interface and managed through the *promoteToModerator()* method within the Tournament Management component. This method calls the *getUserByUsername()* method contained in the UserDAO to check if the username provided corresponds to an account denoted as Educator in the database. If the result is true, then the *promoteToModerator()* method contained in the TournamentDAO is called, which will attempt to update the database. If the operation is successful, the Tournament Management will make an asynchronous call to the Notification Manager to notify the new moderator and the user receives a response with a 200 status code. Otherwise, he/she will receive a response with a 422 status code if one of these conditions is found to be true:

- The provided username corresponds to a user who is already Admin or Moderator of the tournament
- The provided username corresponds to a user who is a Student

[RV11] – Close Consolidation Stage



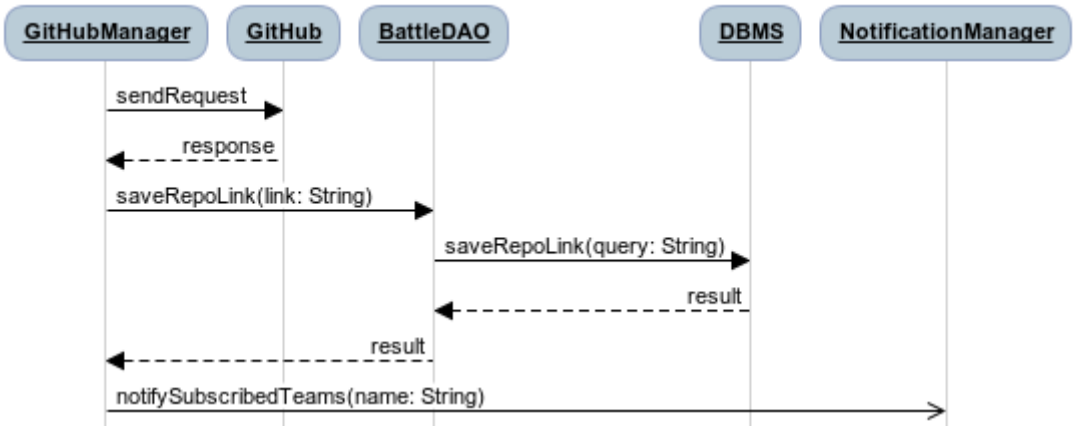
Initially, the user requests to close the consolidation stage via a dedicated form and uses the Web Browser to send a request to the server. This request is received by the Educator Interface and managed through the *closeConsolidationStage()* method within the Battle Management component. If the operation is successful, the method *updateTournamentRank()* contained in the TournamentDAO will update the tournament rank using the battle final rank. If the operation is successful, Tournament Management will make an asynchronous call to the Notification Manager to notify the students subscribed to the tournament and the user will receive a response with a 200 status code. Otherwise, he/she will receive a response with a 422 status code if one of these conditions is found to be true:

- User is not Admin of the tournament
- The consolidation Stage of that battle is already closed

[RV12] – See Info

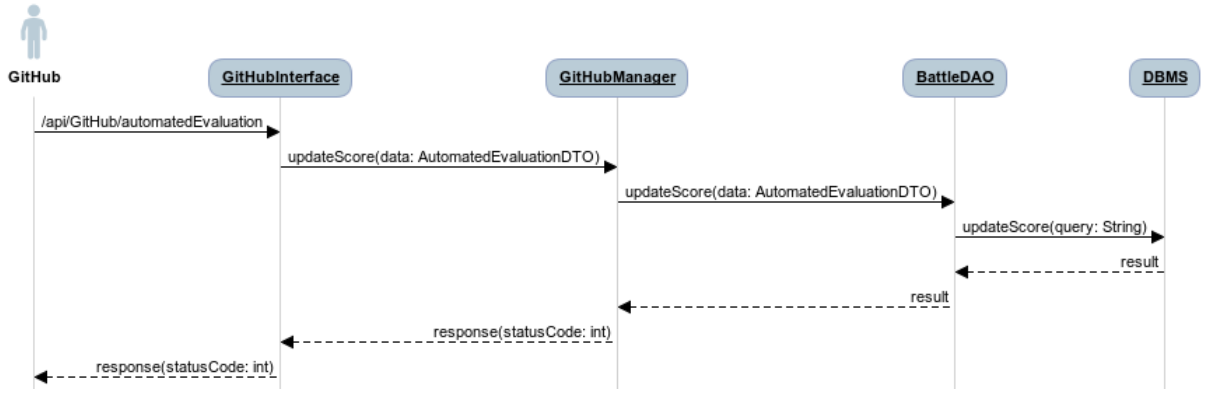
The diagram for the 8th use case (See Info) in the RASD has been omitted. The reason for this is that the unfolding of the use case is very dependent on the type of data required, and thus the diagram may become unreadable. However, it is important to specify that requests of this type will be redirected to the component that deals with the specific user who sent it, so the Student Manager will deal with Student requests and the Educator Manager with Educator requests. Then they will all be forwarded to the DAO that interfaces with the type of data requested and sent back to the client with a 200 status code.

[RV13] – Battle Starts



Initially, GitHub Manager sends a request to the GitHub API to create the repository and upload the files provided at the creation of the battle. GitHub will respond with the link of the repository which will be saved in the database and then sent to all subscribed teams through the Notification Manager with an asynchronous call. The GitHub Manager will also check if all teams meet the minimum number of members required to participate, eliminating those which don't.

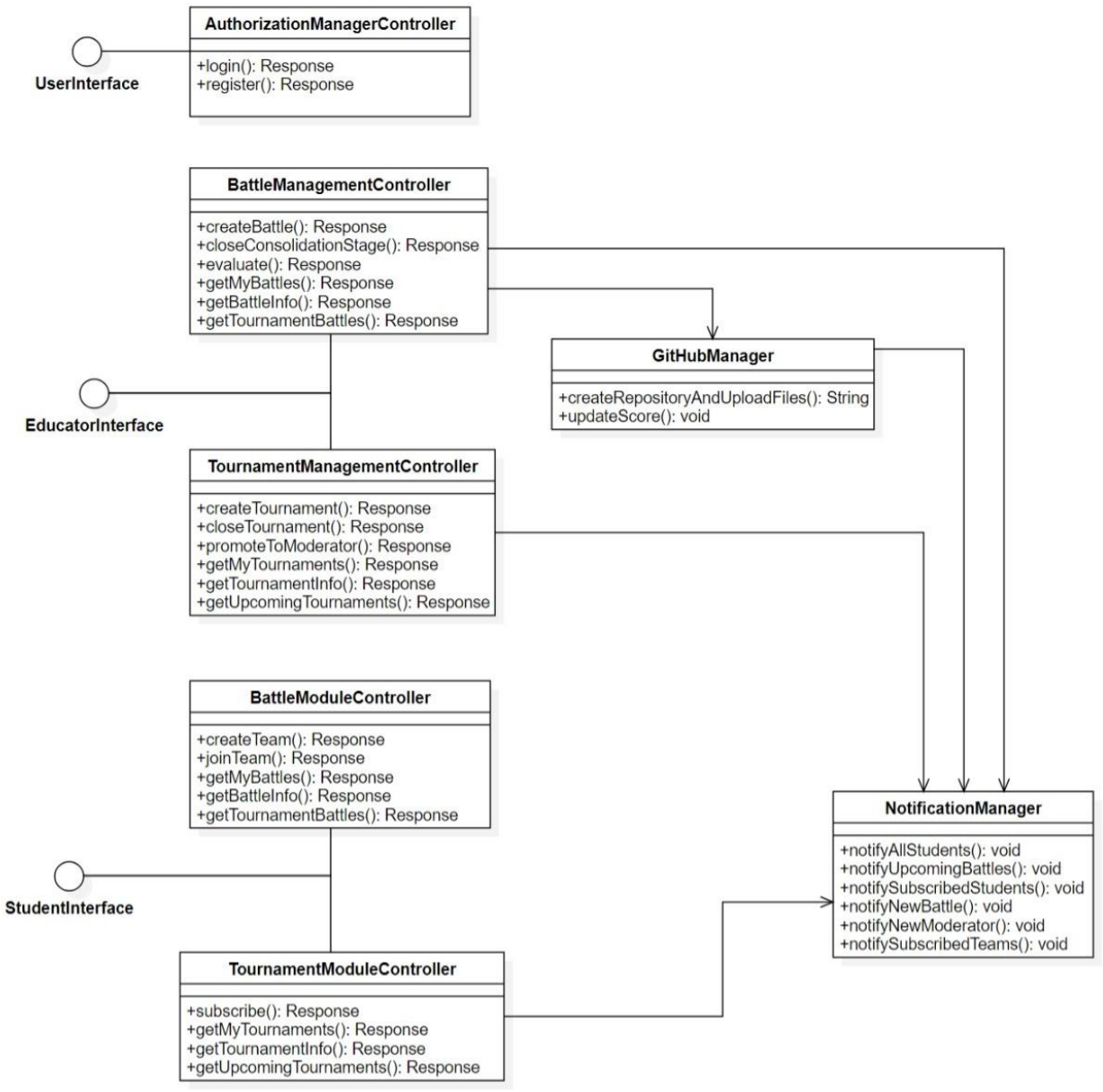
[RV14] – Push Code on GitHub



Initially, through GitHub Actions, GitHub sends a request to the server when a new push is detected. This request is received by the GitHub Interface and managed through the *updateScore()* method withing the GitHub Manager component. This method will perform the automated evaluation and update the score of the team corresponding to the repository in which the push was committed.

E. Component interfaces

This figure illustrates the primary interfaces necessary for the platform's functionality and the corresponding controllers implementing them.



User Interface

This interface provides access to generic user-related functionalities like login and registration. It's exclusively managed by the AuthorizationManagerController, responsible for handling login, registration, and user information requests through methods returning serialized objects or strings as responses.

Educator Interface

Designed for Educators, this interface grants access to functionalities like tournament and battle creation. It's realized through two controllers:

- **TournamentManagementController**

  Manages all tournament-related operations. It offers methods that allow educators to create and close tournaments, promote other educators to Moderators, and retrieve information about tournaments where they hold Admin or Moderator roles.

- **BattleManagementController**

  Handles all battle-related operations. This controller provides methods for creating battles, closing consolidation stages, and adding personal evaluations for teams.

Student Interface

Designed for students, this interface allows access to features such as subscribing to tournaments or battles. It's realized by two controllers:

- **TournamentModuleController**

  Manages all tournament-related functionalities, including methods for tournament subscription and retrieving tournament information.

- **BattleModuleController**

  Handles battle-related operations, providing methods for battle subscription and obtaining battle information.

Additionally, some service classes have been included in this diagram to show their necessity and linkage to specific controllers.

- **GitHubManager**

  Manages the creation of repositories on GitHub and updates team points in a battle.

- **NotificationManager**

  Handles email notifications triggered by events such as the creation and closure of public tournaments, as well as student enrollment in tournaments or battles.

F. Selected architectural styles and patterns

F.1. 4-tiered architecture

The system has been designed in accordance with the 4-tier architecture, therefore it is divided into 4 different tiers which are described in detail in section 2.C.
This architectural choice is aimed to confer flexibility to the system and to facilitate scalability, maintainability and improve performance. For further details on the advantages of this paradigm, see section 2.A.

F.2. RESTful architecture

The RESTful architecture adopted by CKB establishes a framework for structuring and accessing system resources, facilitating interactions between clients (e.g., web browsers) and the server via a client-server model. This architecture prioritizes *stateless communication*, where the server maintains no information about the state of client, enhancing scalability and reliability.
Moreover, servers have the ability to temporarily extend or customize client functionality by transferring software code to the client (*code on demand*), reducing the computational load on the servers. For instance, when filling out a registration form on a website, the browser instantly highlights any mistakes made, such as an incorrect email. Additionally, through *client-side scripting*, all page updates and requests occur on the client-side, improving user experience by allowing more interactivity.

F.3. Model-View-Controller

Model–view–controller (MVC) is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements. These elements are the internal representations of information (the model), the interface (the view) that presents information to and accepts it from the user, and the controller software linking the two.

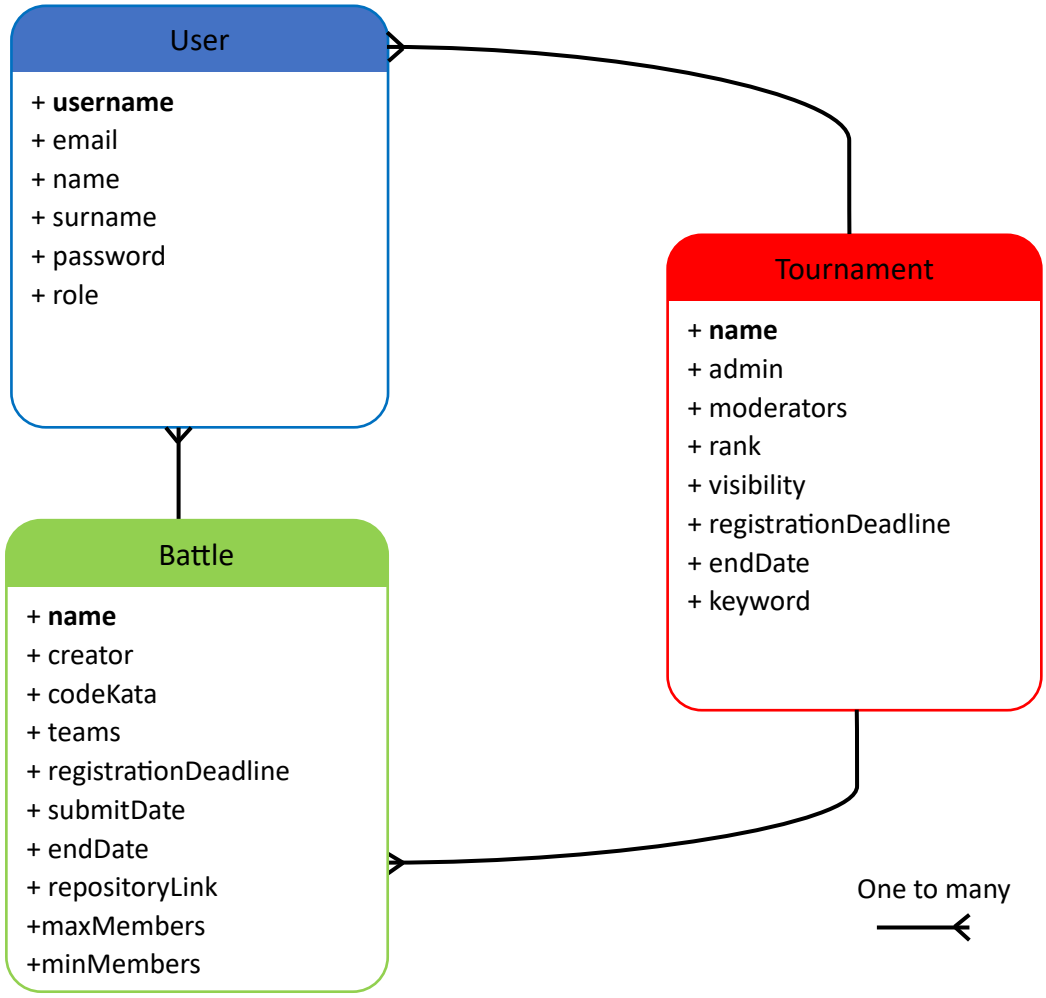
F.4. Thin client and fat server

In this architectural paradigm, the emphasis lies in distributing the workload and responsibilities between the client and the server.
This architectural style gives significant computational load and operational responsibilities to the server, which contains the application's business logic, while the client focuses on rendering user interfaces and forwarding user requests to the server. This means that the client doesn’t require high computational power which makes the application more accessible to various devices.

G. Other design decisions

G.1. NoSQL

In section 2.C.3, it is made clear that CKB needs to be equipped with a highly scalable DBMS, capable of managing a considerable amount of data without lacking in performance, availability and reliability.
For what concerns the logic point of view, CKB logical structure is composed of three main entities represented in the data model below, together with their attributes.



It is observable that the relations among them are few and simple.
Therefore, complying with the NoSQL paradigm may be suitable for organizing CKB data model: it doesn't conform to the relational model of SQL, and consequently it forgoes entity relationships to obtain the desired DBMS properties listed above.

Within NoSQL family, the most appropriate DBMS could be MongoDB: it is best suited for handling objects (*documents*) with a complex structure of attributes and perform atomic operations on them in a very powerful and flexible way.

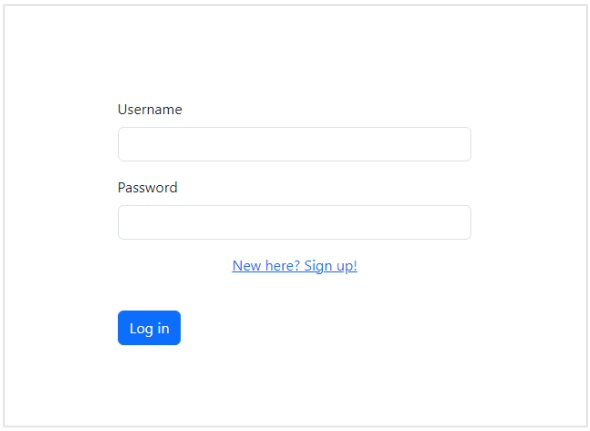
G.2. OAuth2.0

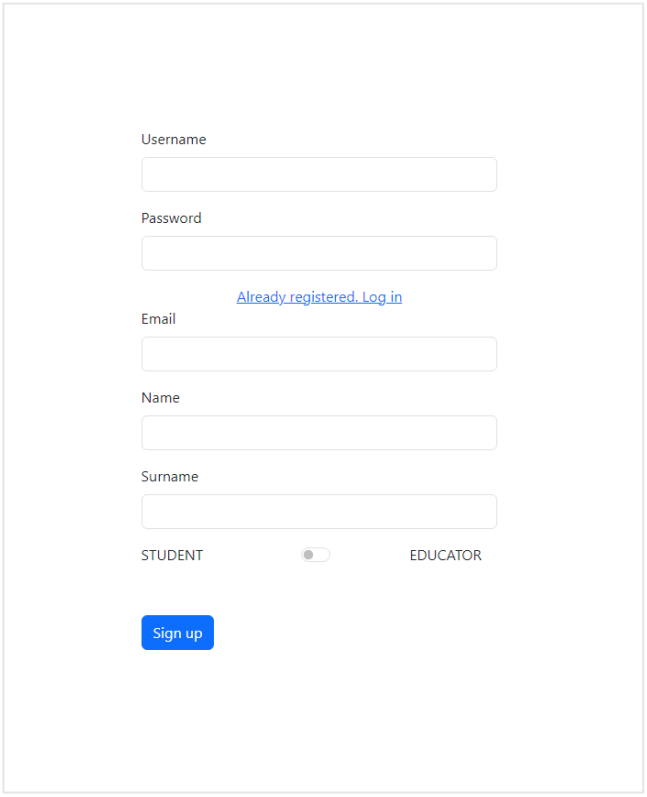
OAuth 2.0 is the industry-standard protocol for authorization. It focuses on client developer simplicity while providing specific authorization flows for web applications, desktop applications, mobile phones, and living room devices.
In the context of CKB, OAuth 2.0 is adopted to guarantee that each user type gains access solely to functionalities designed for their role. This ensures that an educator, for example, can access and utilize features specifically meant for Educators, maintaining strict control over who accesses what within the platform.

3. USER INTERFACE DESIGN

The aim of this section is to show the design concepts of the main pages of CKB platform, also describing the flow from one to another according to user inputs.

Login and Registration Interface



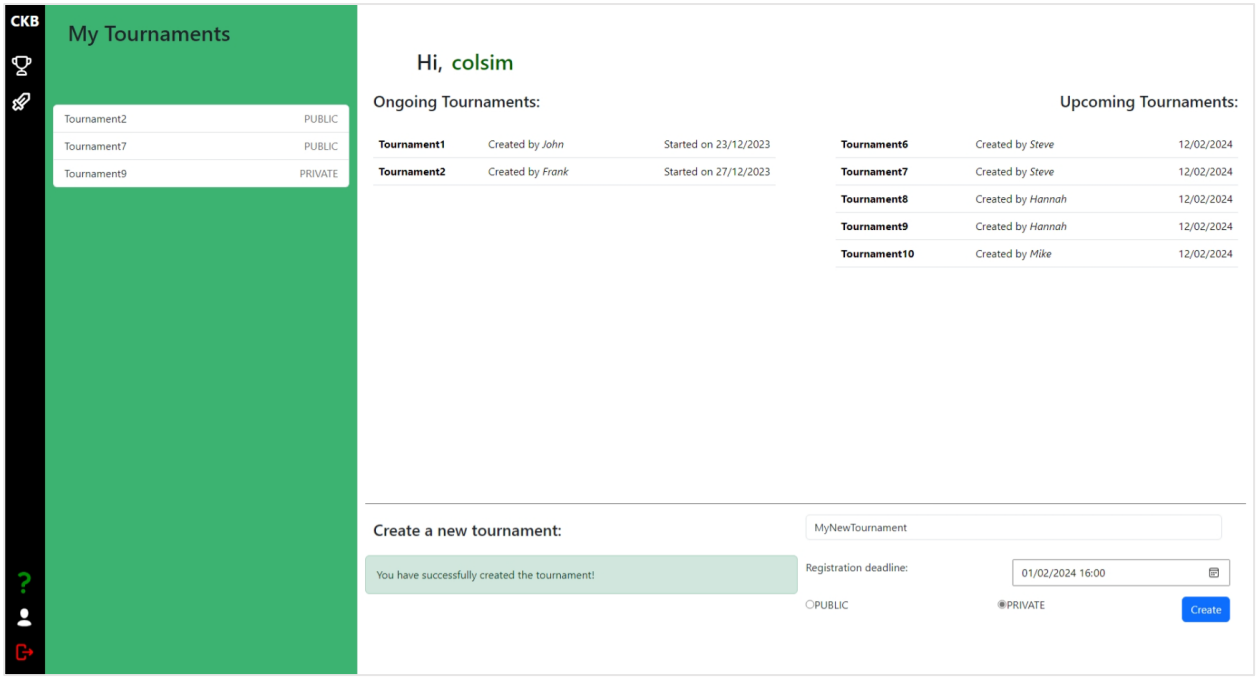


The pictures show the concepts for Login and Registration forms. It is possible to switch from one to another by clicking on the link in the middle of the interface.

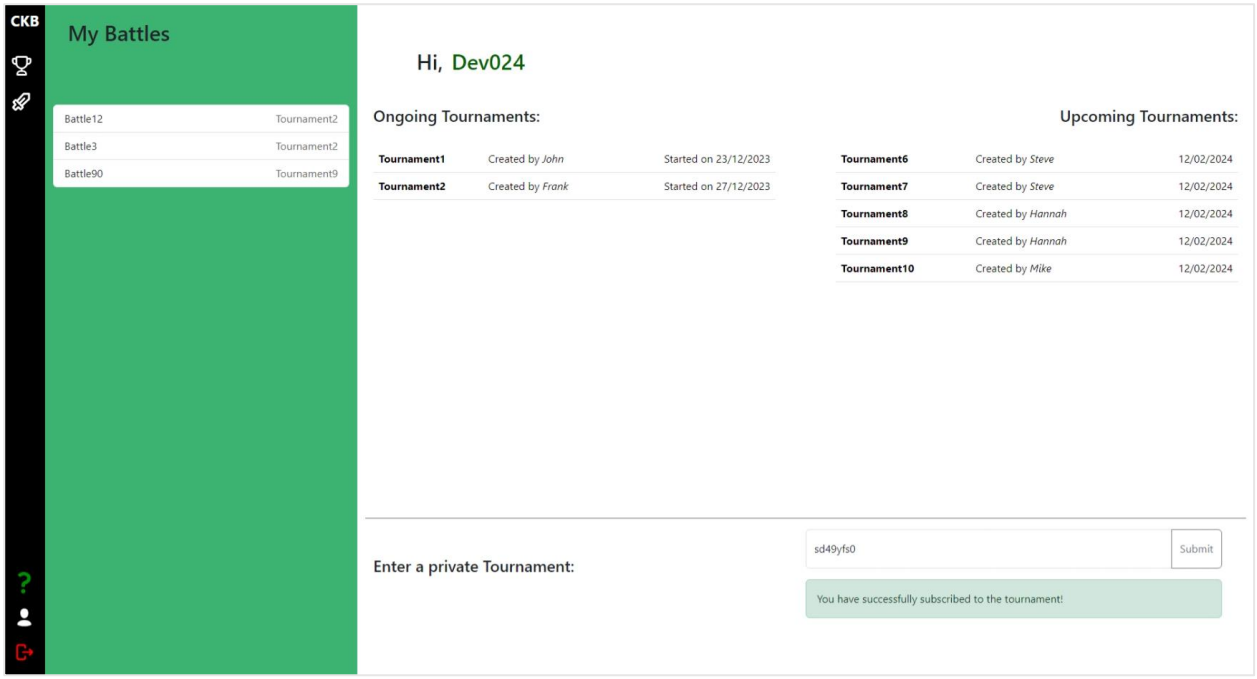
Homepage and MyTournaments/MyBattles interfaces

The pictures show the concepts for the Homepage of CKB platform, together with MyTournaments/MyBattles interfaces that can be opened and closed by clicking on the corresponding icons on the left bar.

As shown by the illustrations, there are two different kinds of Homepage, according to the role of the user.



Educator version



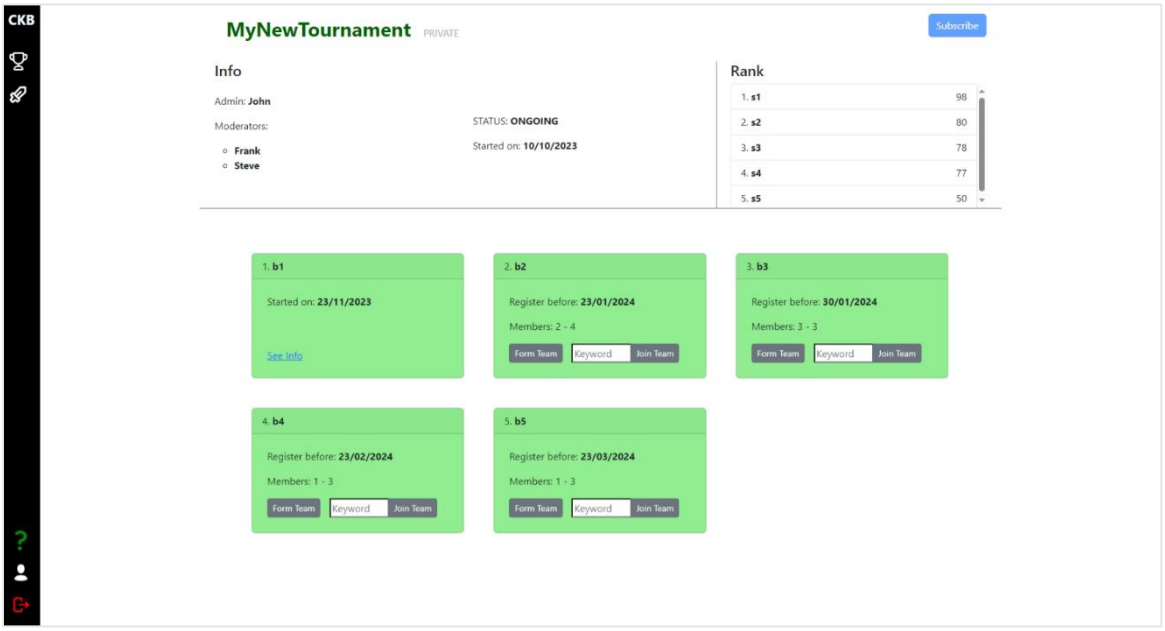
Student version

Both versions of the page allow the user to consult the list of Ongoing and Upcoming PUBLIC Tournaments. Instead, they differ from one another in the bottom section of the page, in which they can exploit two different functionalities: an Educator can fill the form to create a new Tournament, while a Student can join a private tournament by inserting its secret keyword.

Tournament Info Page

The picture below shows the concept for a Tournament Info Page.

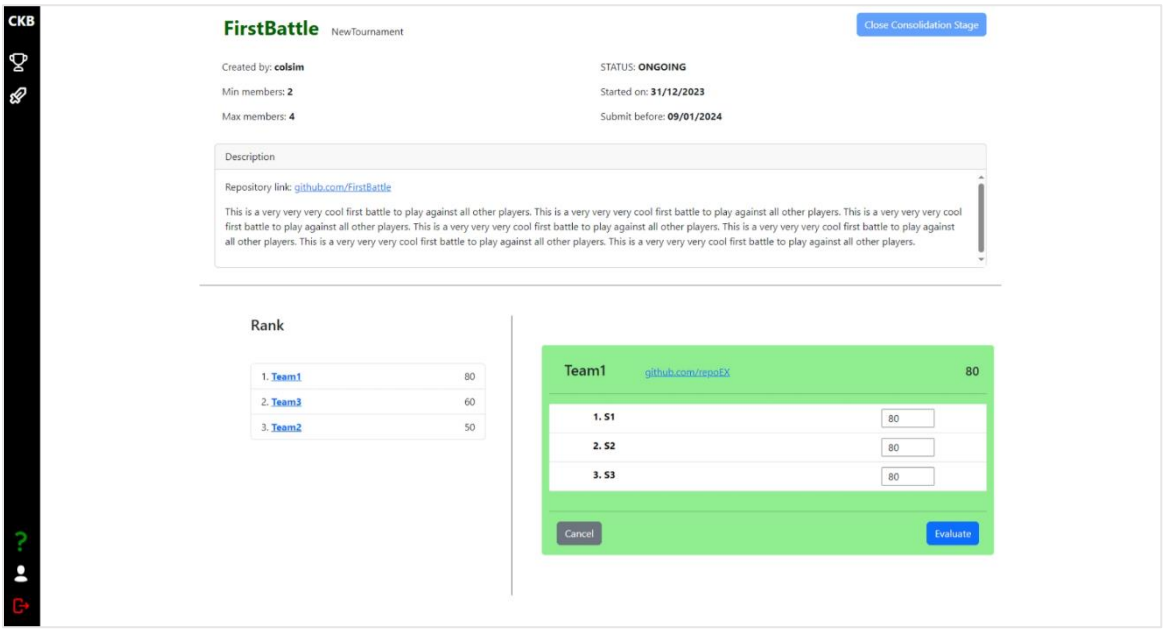
Even for this page there are two versions to allow the differentiation of functions between Educators and Students. This picture represents the Student version of the page of a private tournament, displaying its main information, its rank, and a view of all its battles, with the possibility to form/join teams.



Battle Info Page

The picture below shows a concept for a Battle Info Page.

Even for this page there are two versions to allow the differentiation of functions between Educators and Students. This picture represents the Educator version of the page, displaying the main information about it and giving him/her the possibility to exploit the 'Manual Evaluation' function.



4. REQUIREMENTS TRACEABILITY

This section shows the traceability between requirements and components described in component diagram. It highlights which components are responsible for the realization of each requirement.

A. Functional requirements

Requirement	Component
[RRL1] - The system should provide a “Registration” functionality.	Authorization Manager
[RRL2] - The system should provide a “Login” functionality.	Authorization Manager
[RT0] - The platform should allow users to see information regarding all ongoing and upcoming public tournaments, including the current rank.	Educators: Tournament Management Students: Tournament Module
[RT1] - The platform should provide the “Create Tournament” functionality.	Tournament Management
[RT2] - The platform should provide the “Close Tournament” functionality.	Tournament Management
[RT3] - The platform should provide the “Promote to Moderator” functionality.	Tournament Management
[RT4] - For each tournament, the platform should provide the principal information and the “Subscribe” functionality.	Tournament Module
[RB0] - The platform should allow users involved in a battle to see its current rank.	Educators: Battle Management Students: Battle Module
[RB1] - The platform should be able to perform an automated evaluation of provided code based on functional aspects, timeliness and quality level of sources.	GitHub Manager
[RB2] - The platform should provide the “Create Battle” functionality.	Battle Management
[RB3] - During the consolidation stage, the platform should provide the “Evaluate” functionality.	Battle Management

[RB4] - For each battle, the platform should provide the “Close Consolidation Stage” functionality.	Battle Management
[RB5] - For each battle, the platform should provide the “Create Team” functionality.	Battle Module
[RB6] - For each battle, the platform should provide the “Join Team” functionality.	Battle Module
[RG1] - The system should be able to request GitHub to create a repository and receive back the corresponding link.	GitHub Manager
[RG2] - The system should be able to get information regarding new pushes of code on GitHub.	GitHub Manager
[RN1] - The platform should notify all students when a new tournament is created.	Tournament Management Notification Manager
[RN2] - When a student subscribes to a tournament, the platform should send a notification about all upcoming battles.	Tournament Module Notification Manager
[RN3] - When a battle within a tournament is created, the platform should notify all students subscribed to that tournament.	Battle Management Notification Manager
[RN4] - When a battle starts, the platform should notify all members of each team by sending the link to the repository on GitHub.	GitHub Manager Notification Manager
[RN5] - When the consolidation period of a battle is completed, the platform should notify all the participating students about the battle final rank.	Battle Management Notification Manager
[RN6] - When a tournament is closed, the platform should notify all participating students about the tournament final rank.	Tournament Management Notification Manager
[RN7] - When an Educator gets promoted to Moderator, the platform should notify him/her.	Tournament Management Notification Manager

B. Non-functional requirements

For what concerns system attributes, the design choices presented in this paper illustrate how non-functional features have been accomplished.

In particular:

- **Reliability and consistency**
  The use of the sharding technique, designed specifically to achieve consistency and partition tolerance, together with the replication methods employed, play a key role in ensuring reliability and consistency of data. Multiple replicas of data, provided and managed by the sharding system itself, prevent the risk of data loss, thus contributing to the correctness and truthfulness of information.

- **Availability and Maintainability**
  The system deploys multiple instances of the application server and includes several shards thus avoiding the formation of SPOF in its two most important layers.
  This facilitates the increase of availability, i.e. the failure of a node does not totally compromise the access to all the database, and enhances the maintainability of the system, i.e. a single node can be replaced or upgraded without interrupting the flux of data.

- **Security**
  The adoption of OAuth 2.0, described in section 2G.2, guarantees the compliance with the different roles of the platform (Educator, Student) and strengthens the privacy of sensitive data.
  Moreover, the integration of an encryption protocol may help masking passwords and keywords, securing the access to private areas of the platform.

- **Portability and hardware limitations**
  The system is designed in accordance with the "*thin client*" paradigm, which ensures the lightness of client-side computational load. Moreover, it is realized as a web platform, which by nature implies its portability on several kinds of machines. Therefore, the system is easily accessible to almost whoever has a PC.

5. IMPLEMENTATION, INTEGRATION AND TEST PLAN

As previously described the application is divided as follows:

- Client (Web Browser)
- Web Server
- Application Server
- Database
- External Services (GitHub API)

The platform will be implemented, integrated and tested using a **bottom-up** strategy, which means that developers start from individual components and build up to the larger system. This approach is characterized by breaking down a large problem or project into smaller and more manageable tasks, starting with the details and working toward the bigger picture. For CKB, this consists in starting from the database and work up to the bigger components that receive and handle requests.

A. Component priority

All components mentioned in section 2.B. are essential to fulfill the functional requirements described in the RASD. However, prioritization is necessary among them for implementation:

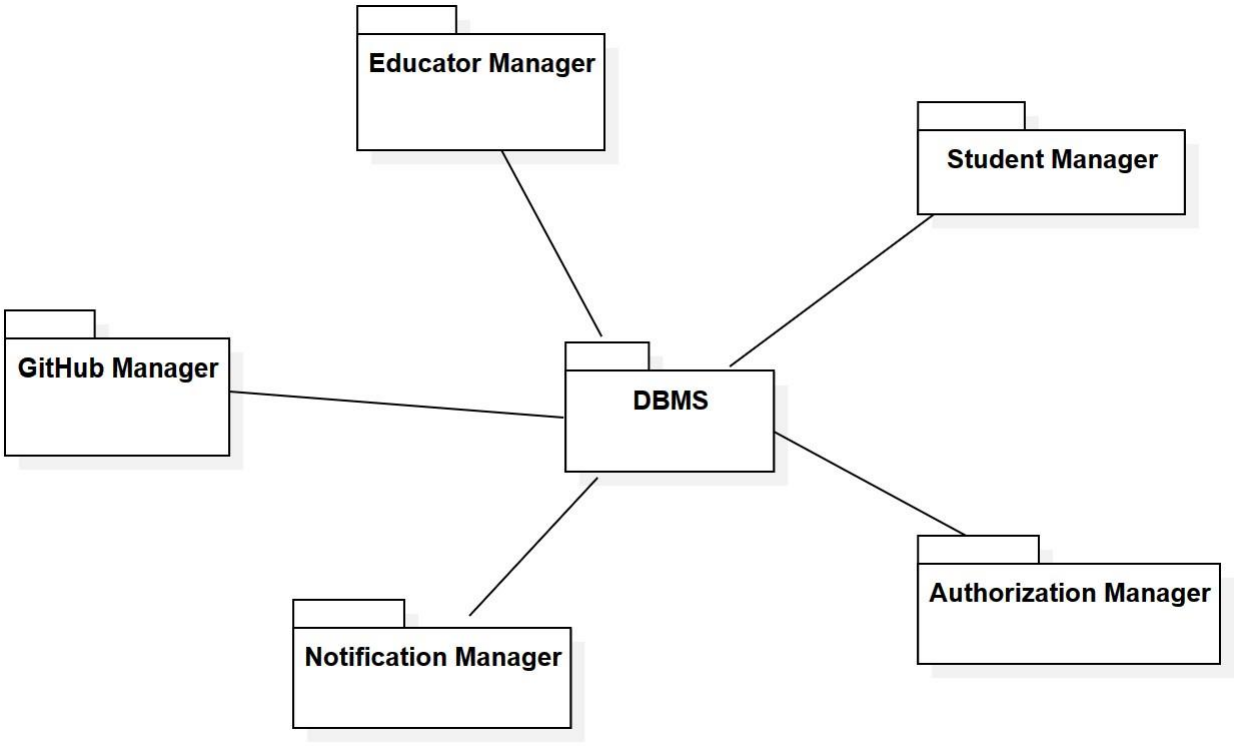
Component	Priority
Student Manager	High
Educator Manager	High
GitHub Manager	Very High
Authorization Manager	Medium
Notification Manager	Low

This prioritization ensures a structured approach to implementing components, focusing on critical dependencies and core functionalities before features that enhance user experience.

B. Build and test plan

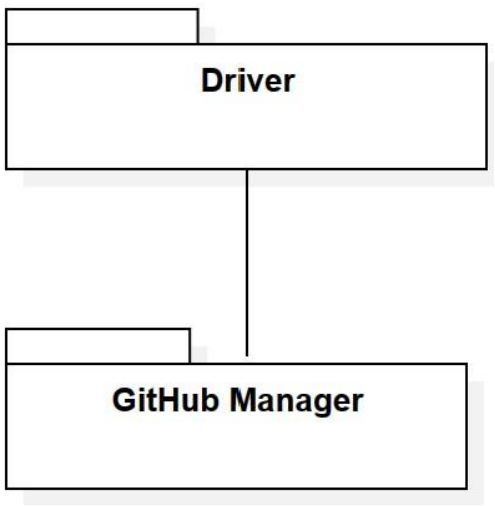
The first step is to create the DBMS since it plays a key role for the CKB platform. Each operation should be optimized to guarantee efficiency and consistency in the database. Then it should be tested before the integration with the other components of the system.

The next step is the development and integration of components that exploit the database functionalities.

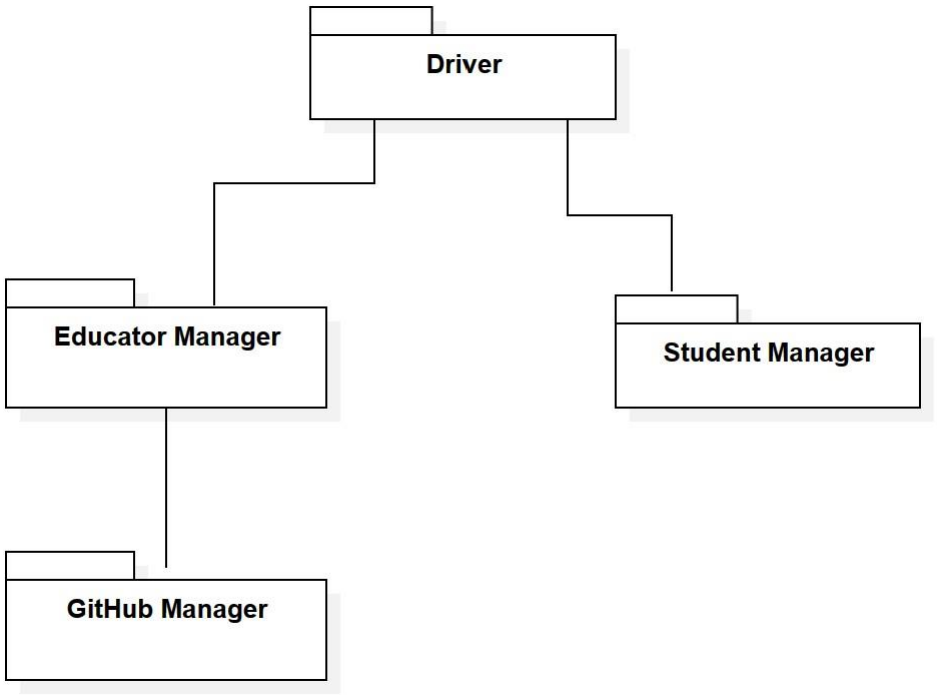


As shown in chapter 2, some components need to interact with each other to guarantee the functionality of CKB. Therefore, all components should undergo **integration testing** using drivers to simulate the behavior of upper-level components that are not yet ready for integration, according with the bottom-up strategy:

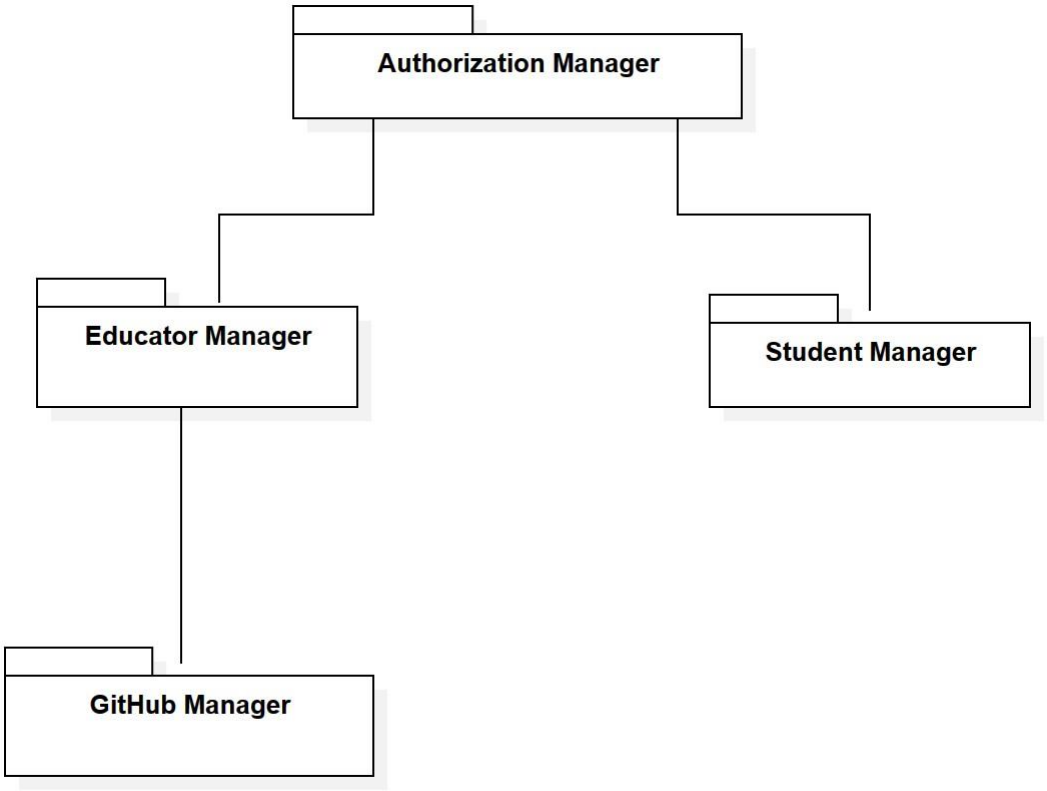
1. GitHub Manager:
   This component is responsible for automated repository creation on GitHub and code evaluation, making it vital to the platform's functionality. Its features are crucial, so developers should spend some time testing to make sure that it works correctly. Developers should verify seamless communication between this component and the API, integrating **static analysis tools** for automated code evaluation. Once released, the Educator Manager will utilize the GitHub Manager to schedule the creation of repositories when creating a battle.



2. Educator Manager and Student Manager:
   After having tested and validated the GitHub Manager component individually, it's time to implement these managers, integrating the previously developed component. Developers should test all functionalities to make sure everything works as intended.



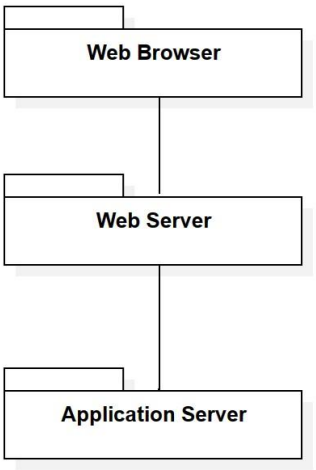
3. Authorization Manager:
   Now it’s time to implement the security of the platform. This component includes functionalities such as login and registration, but it also serves as a protective shield for other components. Its job is to ensure that users can’t get access to functionalities they are



not supposed to, for example, a Student must not be allowed to use functionalities reserved for Educators and to see information that must be available only to them. Therefore, it should be developed, integrated and tested very carefully to ensure the security of the platform.

4. Notification Manager:
   Although this component interfaces with nearly all others to notify users about a specific event, its lower priority allows for its implementation, testing, and integration to occur at a later stage. This is because the functionalities it provides are not critical for the operation of other components.

5. Web Server Module and Browser
   Finally, the application server should be integrated with the web server module and the web browser to achieve client-server communication.



After the web application has been completed, it should undergo **system testing**. The CKB platform should be tested by following the use cases defined in the RASD to ensure that all requirements are met and everything works as it's supposed to.

Developers should also consider **performance testing** to detect bottlenecks affecting response time, utilization, throughput, detect inefficient algorithms, detect hardware/network issues and identify optimization possibilities.

6. EFFORT SPENT

Sections	Fornara	Colecchia	Together	TOTAL
Section 1	0	1	1	2
Section 2	10	5	5	20
Section 3	0	4	1	5
Section 4	0	1	1	2
Section 5	4	0	1	5
Revision&Restyling	0,5	2,5	2	5
TOTAL	14,5	13,5	11	39