

Accademic
Year:
2023-2024



POLITECNICO MILANO 1863

Design Document CodeKataBattle

SOFTWARE ENGINEERING 2
PROFESSOR DI NITTO ELISABETTA

CONTI ALESSANDRO
DI PAOLA ANTONIO

CODICE PERSONA: 10710583
CODICE PERSONA: 10717589

MATRICOLA: 252665
MATRICOLA: 956038

Contents

1. INTRODUCTION	3
1.1. Overview	3
1.2. Purpose	3
1.3. Scope	3
1.4. Definitions, Acronyms, Abbreviations	4
1.4.1. Definitions	4
1.4.2. Acronyms	4
1.4.3. Abbreviations	4
1.5. Revision history	5
1.6. Reference documents	5
1.7. Documents Structure	5
2. ARCHITECTURAL DESIGN	6
2.1. Overview	6
2.1.1. High level view	6
2.1.2. Distributed view	7
2.2. Component view	9
2.3. Deployment view	13
2.4. Runtime view	15
2.5. Component Interface	29
2.6. Architectural styles and patterns	47
2.6.1. Architectural style	47
2.6.2. Patterns	48
2.7. Others design decision	48
2.7.1. Availability	48
2.7.2. Notification timed	49
2.7.3. Data storage	49
2.7.4. Security	49
3. USER INTERFACE DESIGN	50
4. REQUIREMENT TRACEABILITY	63
5. IMPLEMENTATION, INTEGRATION AND TEST PLANNING	72
5.1. Overview	72
5.2. Implementation plan	72
5.2.1. Features identification	73
5.3. Unit testing	74
5.4. Integration testing	74

5.5. System testing.....79

6. EFFORT SPENT.....81

7. REFERENCES.....82

1. INTRODUCTION

1.1. Overview

This Document is intended to provide a comprehensive overview of the CodeKataBattle project. This documentation will be a guide for the reader, enabling them to understand the decisions regarding the design of the application. In particular, the architectural structure and design choices will be detailed, allowing developers, stakeholders, and other team members to gain a clear understanding of how all the features previously outlined informally in the RASD will be implemented.

1.2. Purpose

The purpose of our project, named CodeKataBattle (abbreviated as CKB), is to provide a digital platform that enables students to develop and refine their skills in software development through a collaborative experience. The CKB platform allows students to practice in a programming language of their choice. The exercises offered on this platform are created by educators and require students to complete the code or parts of it, adding the missing components they deem appropriate. The code is then executed and subjected to specific tests created by educators to verify its correctness.

Exercises are organized into tournaments by educators, within which 'battles' are held to evaluate the performance of the students. This process culminates in the creation of performance rankings among all students involved in the battles and tournaments. In this way, CKB encourages friendly competition and the growth of students' skills in the field of software development.

1.3. Scope

The CKB application, as defined in the RASD document, is a web application focused to enhancing software development skills for users enrolled in the Students category. Educators, who have in-depth expertise in the topic, will manage tournaments and battles within the platform to facilitate the improvement of students' skills. Through detailed rankings of battles and tournaments, students can assess their level and monitor progress in enhancing their skills.

1.4. Definitions, Acronyms, Abbreviations

1.4.1. Definitions

- **Commit:** Commits are the core building block units of a Git project timeline. Commits can be thought of as snapshots or milestones along the timeline of a Git project. Commits are created with the `git commit` command to capture the state of a project at that point in time.
- **Fork:** A fork is a new repository that shares code and visibility settings with the original “upstream” repository.
- **Code kata:** Code kata contains the description of a battle and the software project on which the student will have to work, including also test cases and build automation scripts
- **Test Case:** A test case is a singular set of actions or instructions that the Educator wants to perform to verify a specific aspect of the project pushed by the students. If the test fails, the result might be a software defect that students might have not found.
- **Upload:** Upload in which the educator sends to the platform database the code kata for a specific battle.
- **Repository:** A Git repository is the .git/folder inside a project. This repository tracks all changes made to files in your project, building a history over time. Meaning, if you delete the .git/folder, then you delete your project’s history.
- **Branch:** In Git, a branch is a new/separate version of the main repository. Branches allows users to work on different parts of a project without impacting the main branch. That main branch is the one seen as the default one.
- **Push:** The `git push` command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo.
- **Pull:** The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration work flows.
- **Demilitarized Zone:** A demilitarized zone is a physical or logical subnet that contains and exposes services to an external network that is not considered secure, such as the Internet. The purpose of a DMZ is to protect an organization's LAN.

1.4.2. Acronyms

- CKB: CodeKataBattle.
- CK: CodeKata, Code Kata.
- GH: GitHub.
- GHA: GitHub Action.
- DMZ: Demilitarized Zone.
- DAO: Data Access Object.
- RASD: Requirements Analysis and Specification Document.
- DD: Design Document.

1.4.3. Abbreviations

- [Rn]: the n-th functional requirement.
- [SDn]: the n-th sequence diagram.
- [Fn]: the n-th feature.
- [IPn]: the n-th integration phase.
- Repo: Git repository.

1.5. Revision history

- Version 1 (2024/01/06).
- Version 2 (2024/02/04)
 - Updated sequence diagram based on the implementation (section 2.4.)
 - The component interface has been updated based on the actual implementation (section 2.5.)
 - Typing error correction (section 5.3.)

1.6. Reference documents

This document is strictly based on:

- The specification of the RASD and DD assignment of the Software Engineering 2 course, held by professor Matteo Rossi, Elisabetta Di Nitto and Matteo Camilli at the Politecnico di Milano, A.Y 2022/2023.
- Slides of Software Engineering 2 course on WeBeep.
- Official link of CodeKata: <http://codekata.com/>.

1.7. Documents Structure

The rest of the document is organized as follows:

- *Architectural Design* (Chapter 2) contains a detailed description of the system architecture, the definition of the main components and the relationships between them.
The last part of the section will describe the design choices, models and paradigms used in the implementation.
- *User Interface Design* (Chapter 3) contains a detailed description of the user interfaces, which are presented in the RASD in the form of an overview.
- *Requirement Traceability* (Chapter 4) shows the relations between the requirements from the RASD and the design choices of the DD and how they are satisfied by the latter.
- *Implementation, Integration and Test Planning* (Chapter 5) provides a road-mapping of the implementation and integration process of all components and explains how the integration will be tested.

2. ARCHITECTURAL DESIGN

2.1. Overview

In this section we will explain what components will compose our system, the interaction between them, and the replication mechanisms chosen to make the system distributed.

2.1.1. High level view

The CKB application will be implemented following the idea of a three-tier architecture as shown in Figure 1.

The three tiers are the follow:

1. **Presentation Tier:** This tier allows interaction between the user and the application through a user interface based on web pages. These pages dynamically adapt to user requests and application responses.
2. **Application Tier:** This tier handles user requests, retrieves and, if necessary, modifies information in the database.
3. **Data Tier:** This tier constitutes the main database. Its primary function is to store data securely and make it available when requested by the user.

The three-tiered distributed architecture allows efficient division of responsibilities, making it easier to scale and manage components.

The Presentation Tier handles the user interface, the Application Tier handles the application logic, and the Data Tier handles persistence and data access. This subdivision facilitates maintenance, scalability, and optimization of overall system performance.

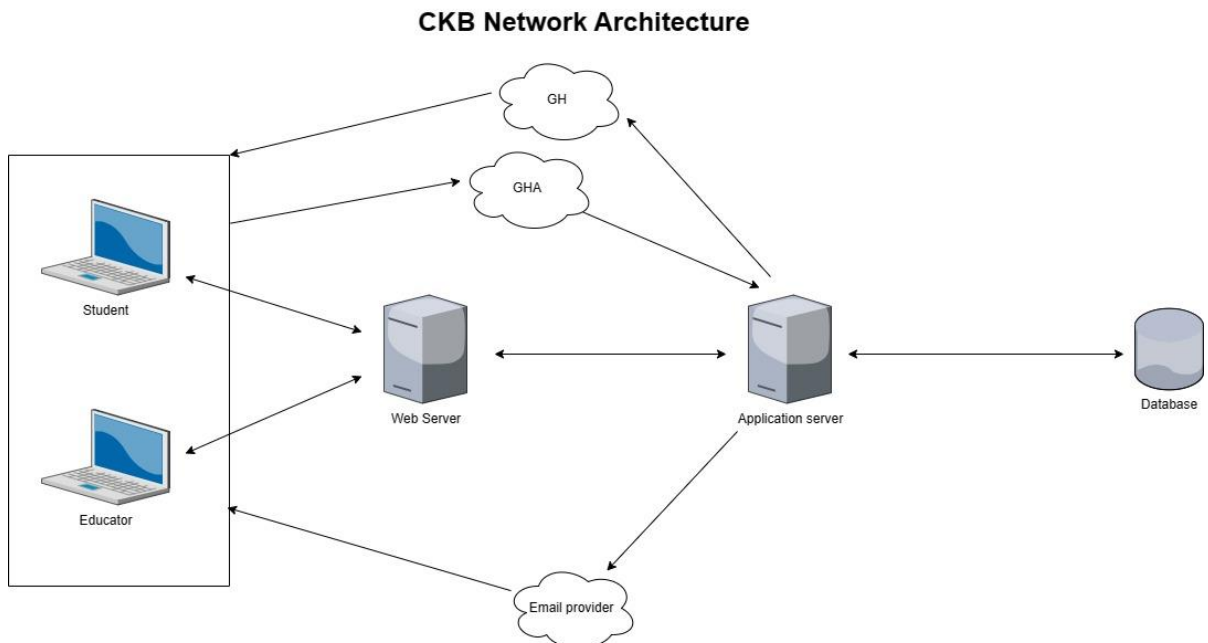


Figure 1 High Level View

2.1.2. Distributed view

The CKB application will be implemented following the idea of distributed system architecture as shown in Figure 2. This implementation choice allows different components to be deployed on separate machines, so as to improve fault tolerance and reduce the computational load on individual machines.

The Web Server will be the first component to be deployed; each machine will manage a specific page with which the user interacts. To route user requests to the corresponding server, a load balancer must be installed between the Web site and the Web Server.

The Application Server will also be deployed on different machines. Each of them will handle a certain functionality that will interact with the others via public method interfaces. An additional load balancer will be installed between the Web Server and the Application and will take care of distributing the computational load equally among the machines handling the required service.

The components we are going to add to ensure the security of the system and to distribute the computational load received are:

- *Firewall*: a network security device that monitors incoming and outgoing traffic through a predefined set of security rules, deciding whether to allow or block certain events. The firewall then acts as a filter for all user requests, allowing only those authorized according to specific access permissions to pass through.
- *Load Balancing*: a system that fairly distributes the workload it receives among various hardware resources in order to optimize system performance. Load balancing aims to ensure fair allocation of user requests across different machines, thus helping to improve service availability so that it is more efficient.

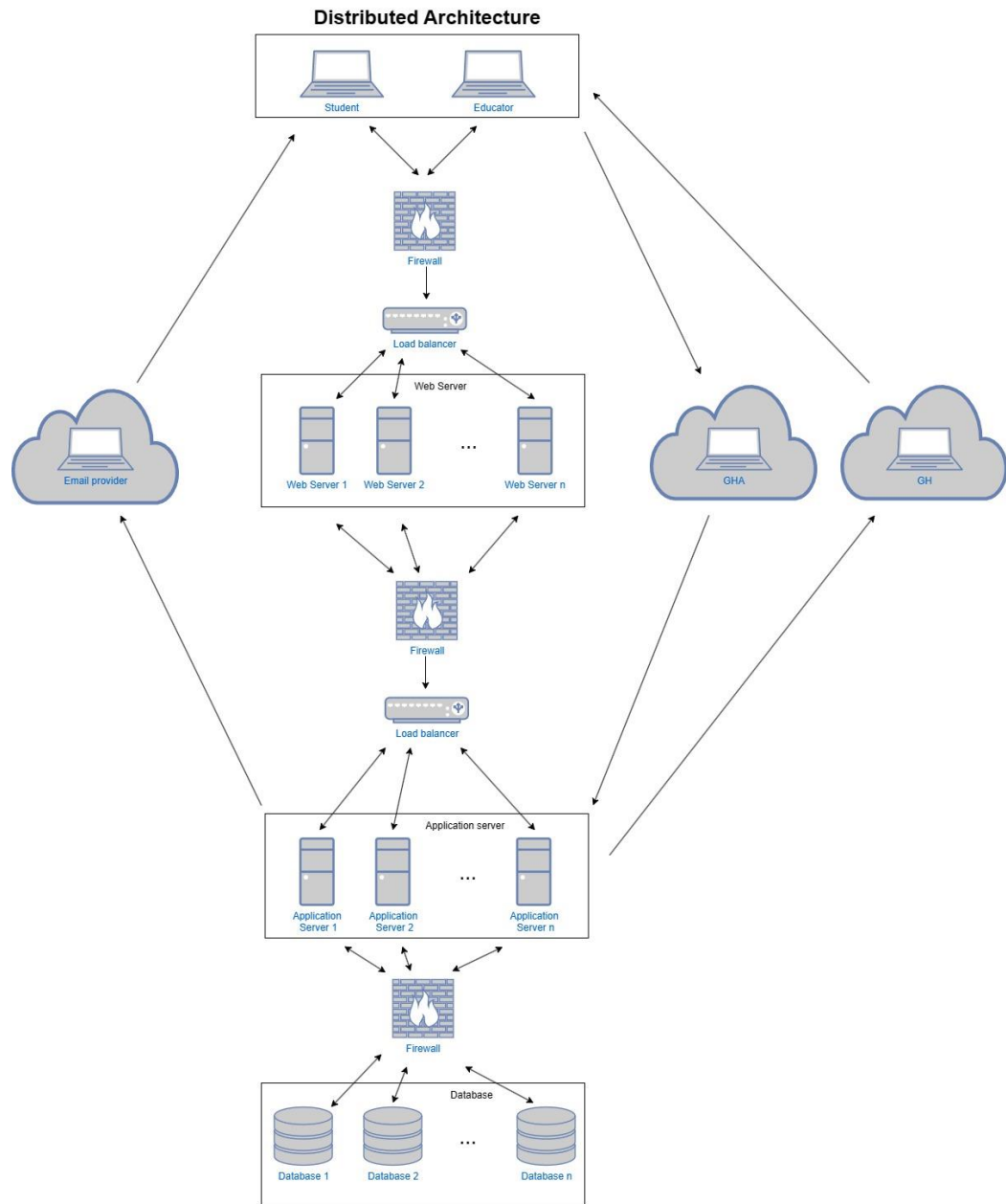


Figure 2 Distributed View

2.2. Component view

This section will show the component view of the system representing the internal architecture of the CKB application.

The diagram we see in Figure 3 allows us to identify the three architectural levels of the application, as described above.

1. The presentation tier, represented by our *Web Site*, deals with dynamic interaction with the user. Whenever a change to the web page occurs, this layer of the architecture communicates with the *Web Server*. This interaction ensures a timely and accurate response to user requests, helping to provide a dynamic web experience aligned with user interactions. In this process, the presentation tier acts as a key interface between the user and the system, ensuring that changes made to the page are reflected consistently and immediately on the website that the user views.
2. The application tier, jointly represented by the *Web Server* and the *Application Server*, handles requests from users and interacts with the database containing the application data.

When a user sends a notification to the server, the *Web Server*, via the web page displayed to the user, forwards the request to the *Application Server* in the appropriate format. The *Application Server* is the key component that, upon receiving the request in the correct format, forwards it to the database to retrieve and/or modify the data within. This process constitutes an effective communication flow that allows user requests to be fulfilled efficiently and consistently, while ensuring proper access and manipulation of data in the database.

3. The data tier is represented by the *database*, which is responsible for storing and making accessible the data essential to the proper functioning of the application.

The *database* ensures that the information stored in it can be accessed by the *Application Server* whenever it is necessary to retrieve and/or modify data.

All the components that will compose the system will be explained in detail below.

- Web Site

This component takes care of user interaction.

- Web Server

All of these components handle the pages that user sees and can interface with.

- Home Page

On this page, the user is given a choice between two options: registration or access to the application. Selecting one of these options automatically causes the corresponding page to change.

- Login Page

This component handles the login page.

- Sign In Page

This component handles the sign in page.

- Personal Home Page

This component is responsible for managing the home page.

This page shows the specific home page of each specific user.

- Tournament Page

This component handles the tournament page.

- Battle Page

This component handles the battle page.

- Application Server

This component takes care of the interaction between the user and the database.

- Servlet Login Manager

This component handles the login process of a user. It checks whether the user is registered in the database and, if so, redirects the web page to its designated home page.

- Servlet Sign In Manager

This component handles the registration process of a user. It checks whether the user is not registered in the database and, if so, saves the user.

- Servlet Manage Tournament

This module represents the tournament management service. The service comprises all components dedicated to displaying tournament information, creation and closure by educators, addition of contributors, registration by interested students, and searching by a user.

- Servlet Log Out

This component handles the logout of the user who had logged in.

- Servlet Modify Personal Information
This component is responsible for modifying the user's personal data in the database.
- Servlet Manage Battle
This module represents the battle management service. The service comprises all the components dedicated to displaying battle information, creation by the educators managing the tournament, and registration by interested students.
- Servlet Manage Team
This module constitutes the team management service. The service comprises all components dedicated to team creation, acceptance of invitations from members, and registration for the corresponding battle.
- User DAO
This component handles all interactions with the database that require access to user information.
- Tournament DAO
This component handles all interactions with the database that require access to tournament information.
- Battle DAO
This component handles all interactions with the database that require access to battle information.
- Team DAO
This component handles all interactions with the database that require access to team information.
- Evaluation
This component is responsible for grading the work of a team of students according to the requests of the educator who created the battle.
- Start Battle
This component handles the initial stages of a battle. It is responsible for creating a repository on GH for each enrolled team and notifying the enrolled teams of the start of the battle.
- GH Organizer
This component handles the interaction with GitHub, both in terms of the application, i.e., creating the repositories with the GitHub Actions needed for the evaluations provided by the Evaluation component, and receiving information from GitHub, such as the results of executing code from a group of students.

- Send Notification

This component is responsible for interacting with the email provider in order to send notifications to interested users.

The notifications it needs to send are created by the following components: *Servlet Create Tournament*, *Servlet Create Battle*, *Servlet Close Tournament* and *Servlet Join Battle*.

- Database

This component is responsible for saving the data and making it accessible to each user request.

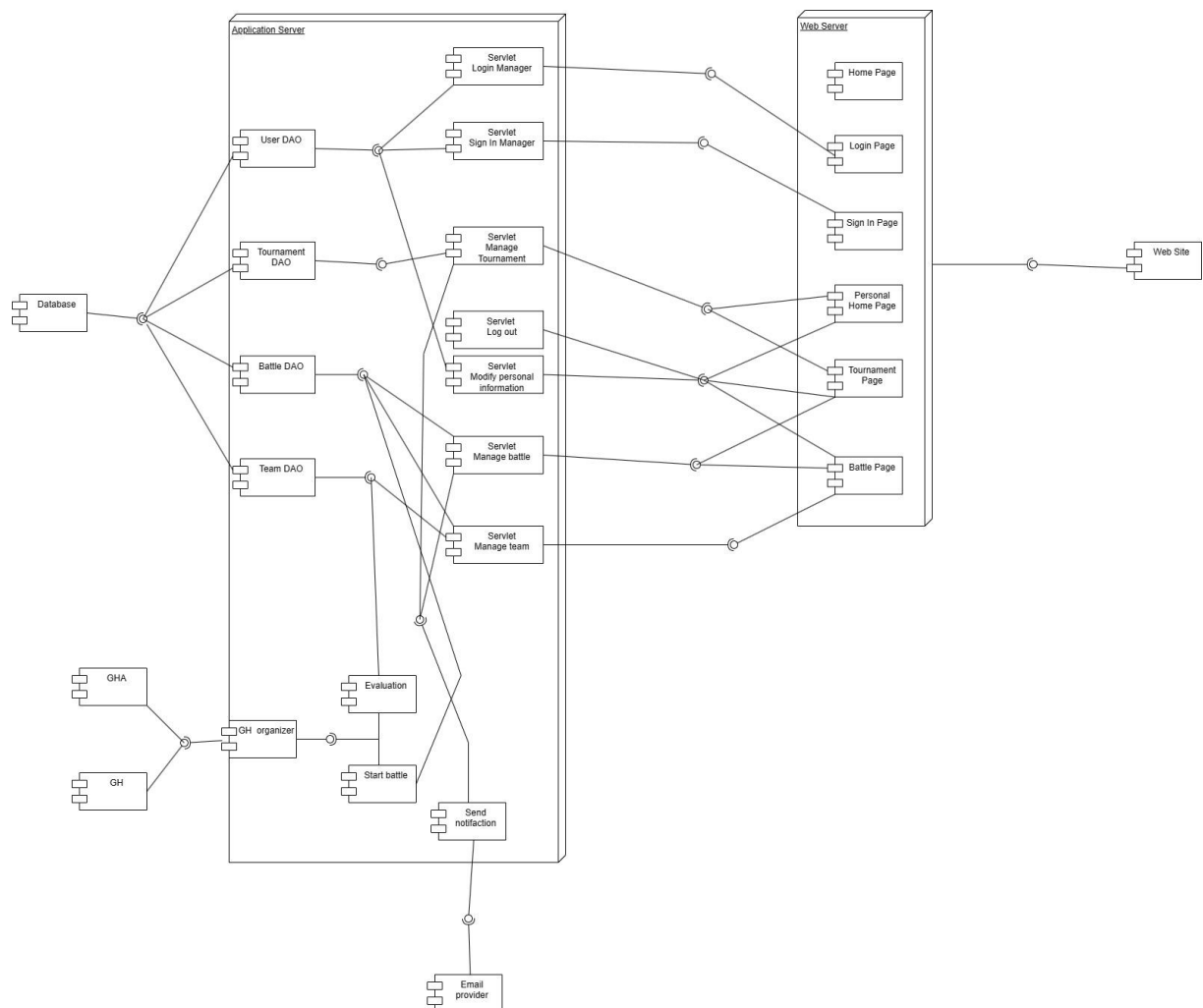


Figure 3 Component View

2.3. Deployment view

In this chapter is described the deployment view for CKB. This view describes the execution environment of the system and the geographical distribution of the hardware components that executes the software composing the application.

Following this, all the components displayed in the Figure 4 will be explained in detail.

- *PC Student* and *PC Educator* represent a computer used by a Student and a Educator. The two devices must have a web browser capable of interpreting the Javascript language. This browser is indispensable for browsing the Internet, enabling access to the CKB web page and fluid interaction with the application.
- *Firewall* is a device that monitors packets entering the system, if a packet is potentially dangerous it is not forwarded. These components are placed before load balancers, with the goal of carefully filtering all packets entering the network. This configuration ensures that only packets deemed safe are allowed to enter the application's network environment. Through this implementation, a DMZ is established in which security is maximized through strict access control, thus helping to protect the LAN's application from potential external threats.
- *Load Balancer* is a device which performs the load balancing. This component deals with dividing a group of requests among several machines that specialize in their execution. This approach aims to optimize the overall performance and increase the scalability of the system. Through the intelligent distribution of requests, the component helps ensure efficient use of available resources, greatly improving the responsiveness of the system and its ability to handle increasing workloads in a flexible manner.
- *Web Server* is the component that receives all user requests and passes them to the *Application Server* to execute and return a response. This component is the one responsible for changing the pages or content that the user sees on the screen.
- *Application Server* is the component that receives all user request, processes a response and sends it to the *Web Server* to display. This component is the only one that can process the data, read it and modify it on the database.
- *Database Server* is the component that saves and accessibility the data.

We have opted to use Amazon Web Services' RDS service as the main database. By implementing this choice, we will not have to evaluate the scalability and reliability of the data storage system, since these aspects are fully handled by the service provided by Amazon Web Services.

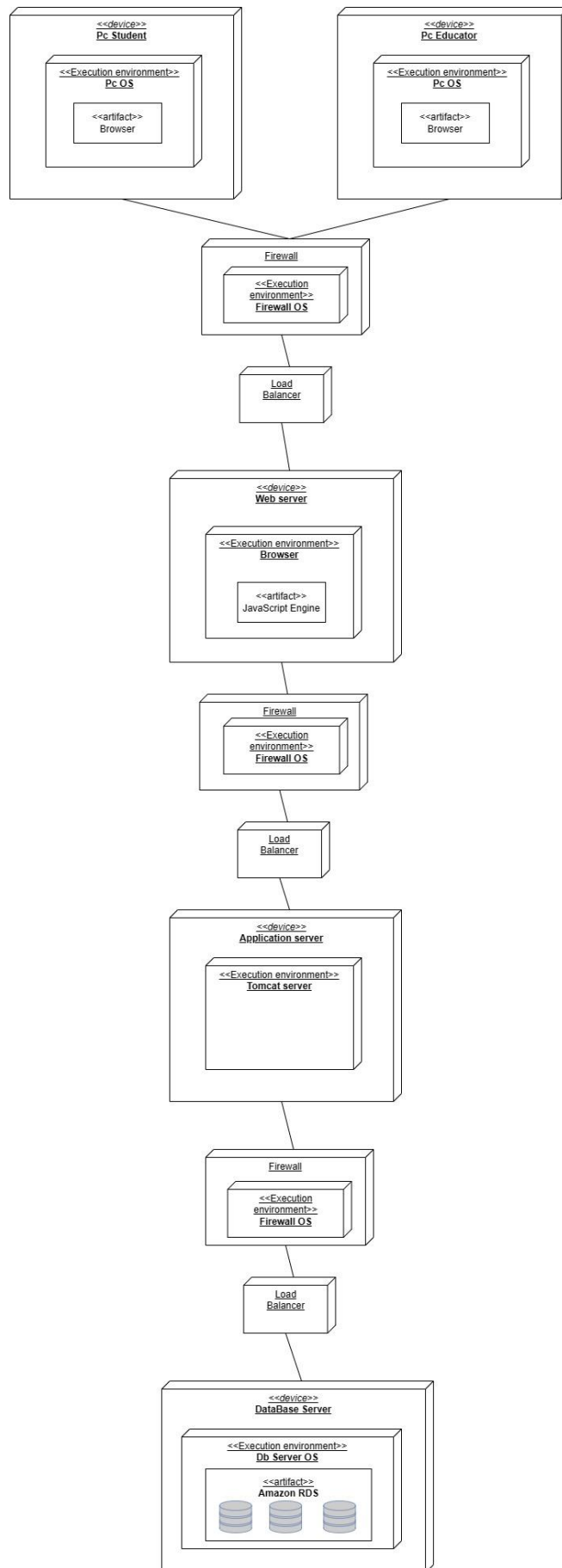


Figure 4 Deployment View

2.4. Runtime view

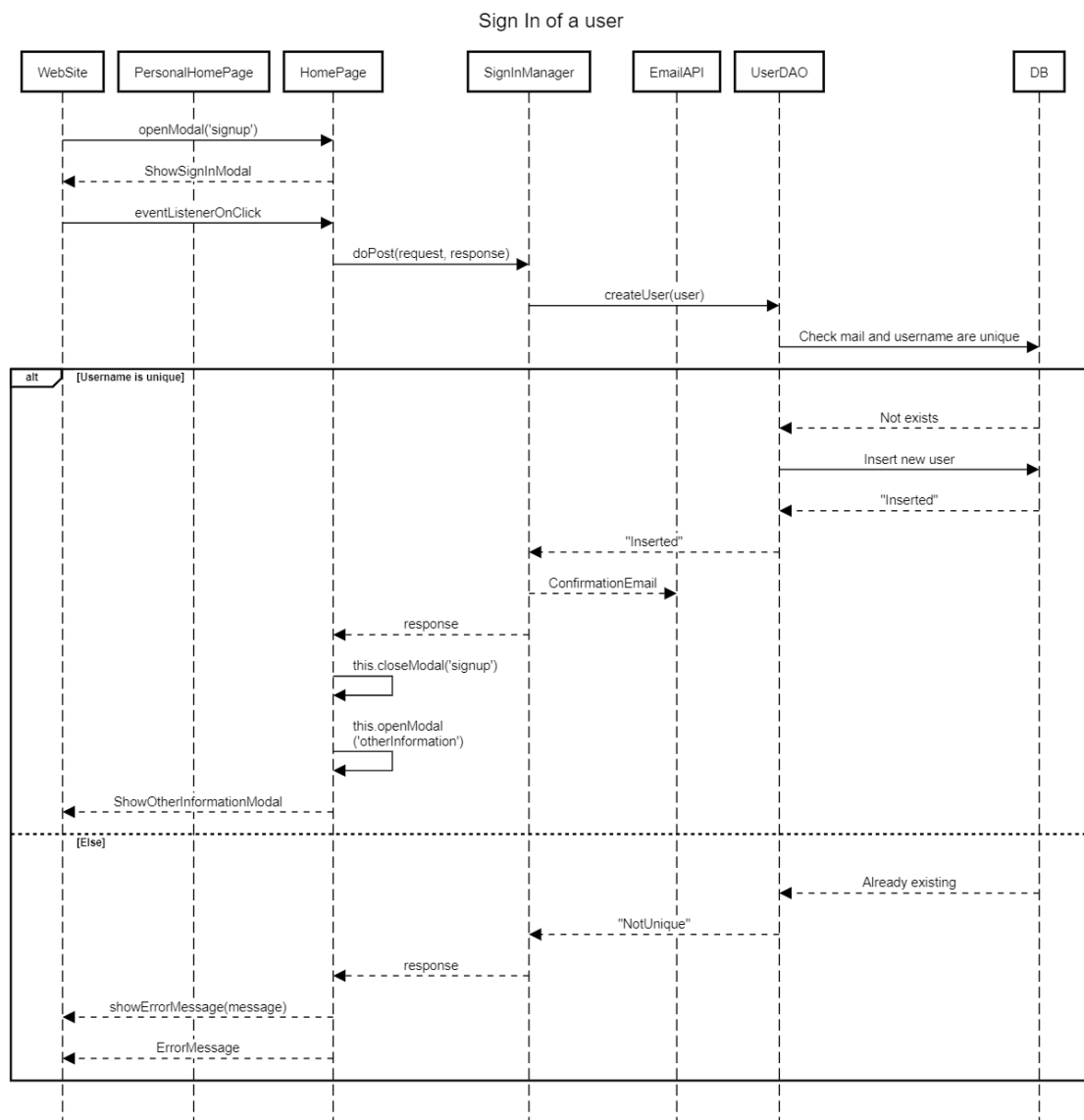
This section will present all the runtime views associated with the Use Cases found in the RASD. The runtime views show the interactions between the various components of the CKB system.

[SD1]. Sign in of a new user

This sequence diagram represents the interaction that occurs when a user wants to register to the CKB application.

When a user wishes to register in the application, they are shown a page to fill in with some personal data. After entering all the required information and pressing the *Submit* button, the data is transmitted to the server responsible for managing the saving and a confirmation email will be sent to them for the successful creation of the account.

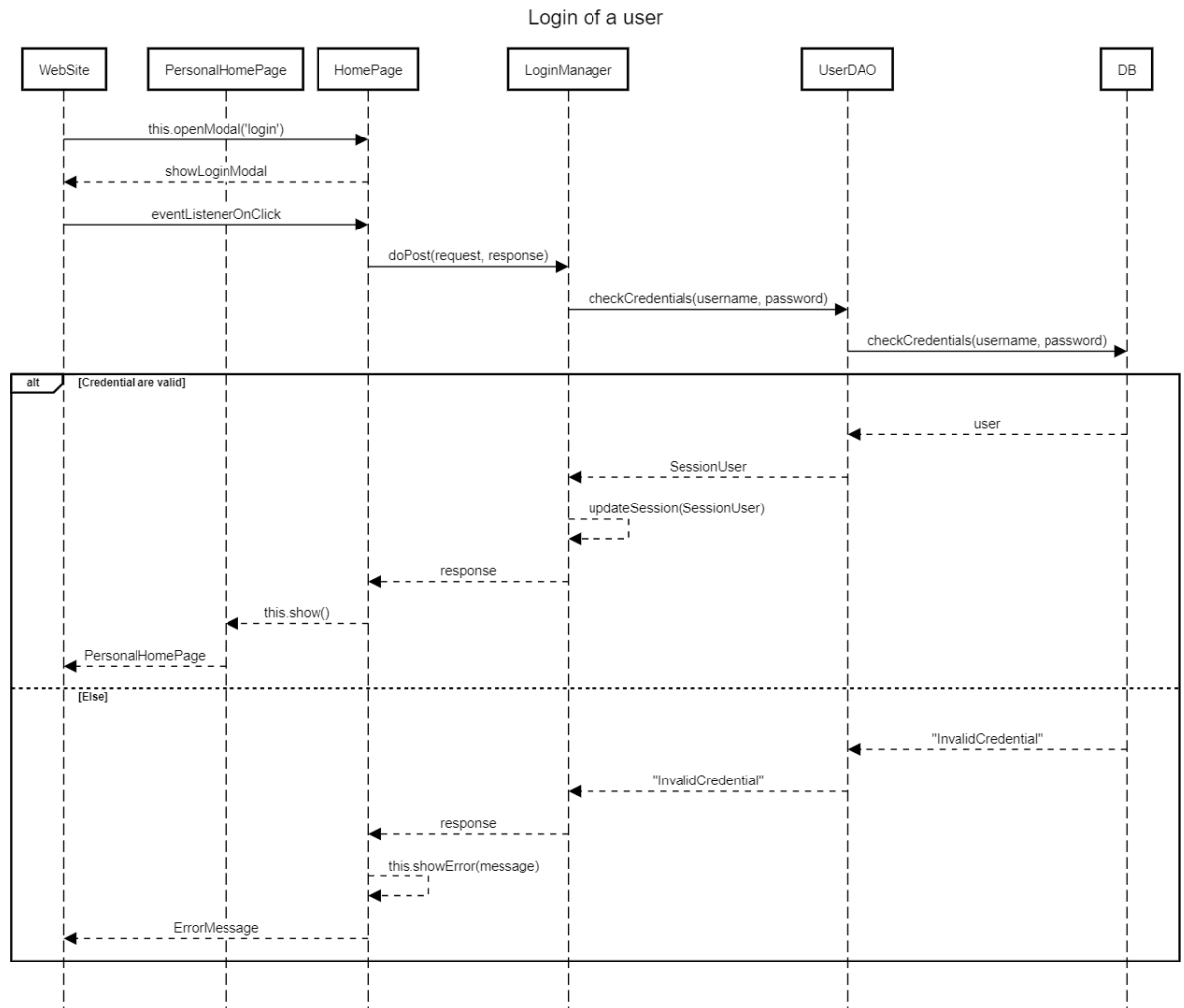
In case an error occurs, for example, if the user has already created an account with the same email or username, an error page is displayed. In this scenario, saving the data to the database and sending the confirmation email are not performed.



[SD2]. Login of a user

This sequence diagram represents the interaction that occurs when a user wants to access the CKB application.

When a user wishes to access the application, they enter their login credentials, which are then sent to the server for verification. If the verification is successful, the user is granted access permission and the application will show their customized Home Page. In case the credentials are incorrect, the user will display an error message prompting thier to provide the correct credentials.

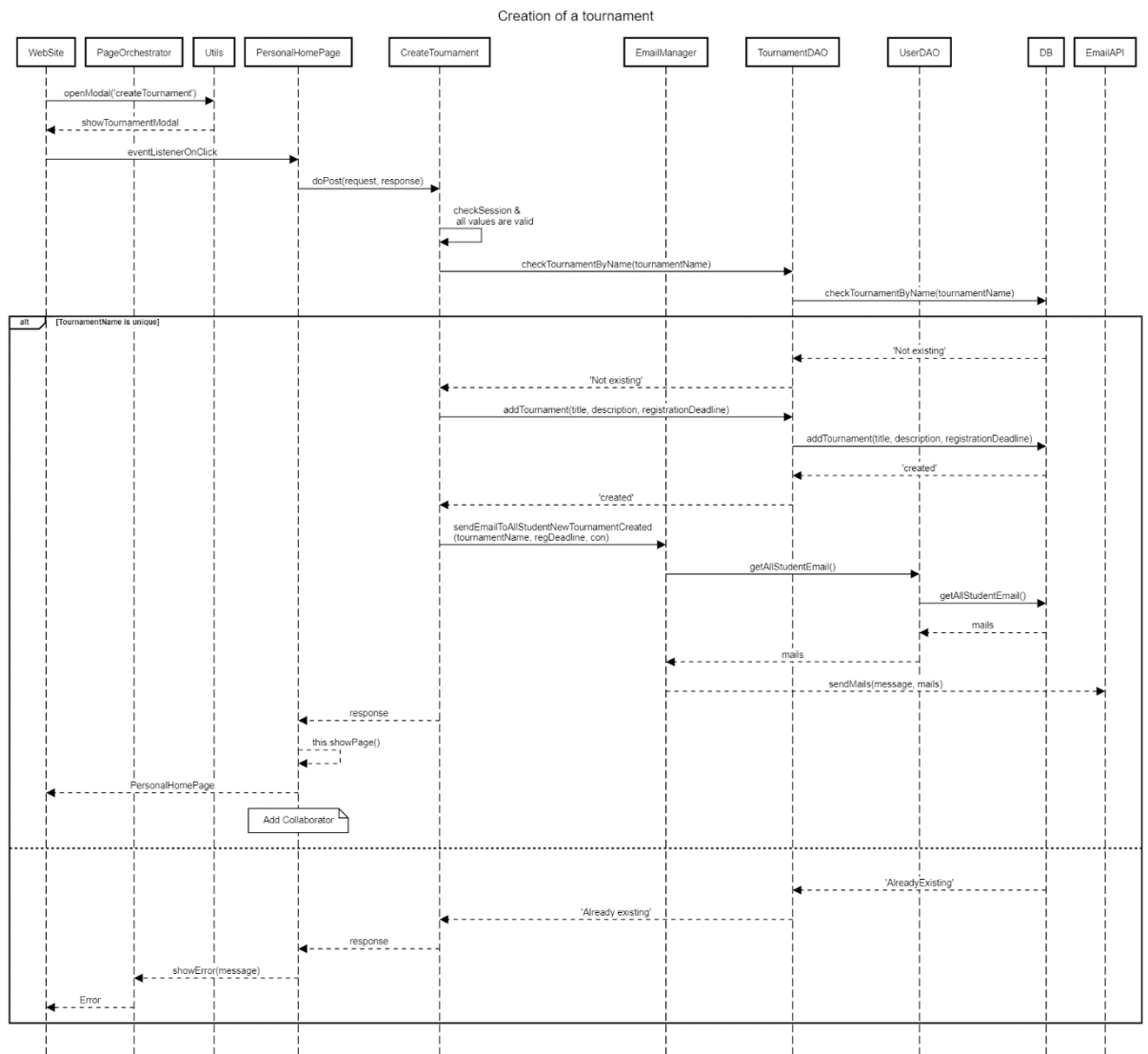


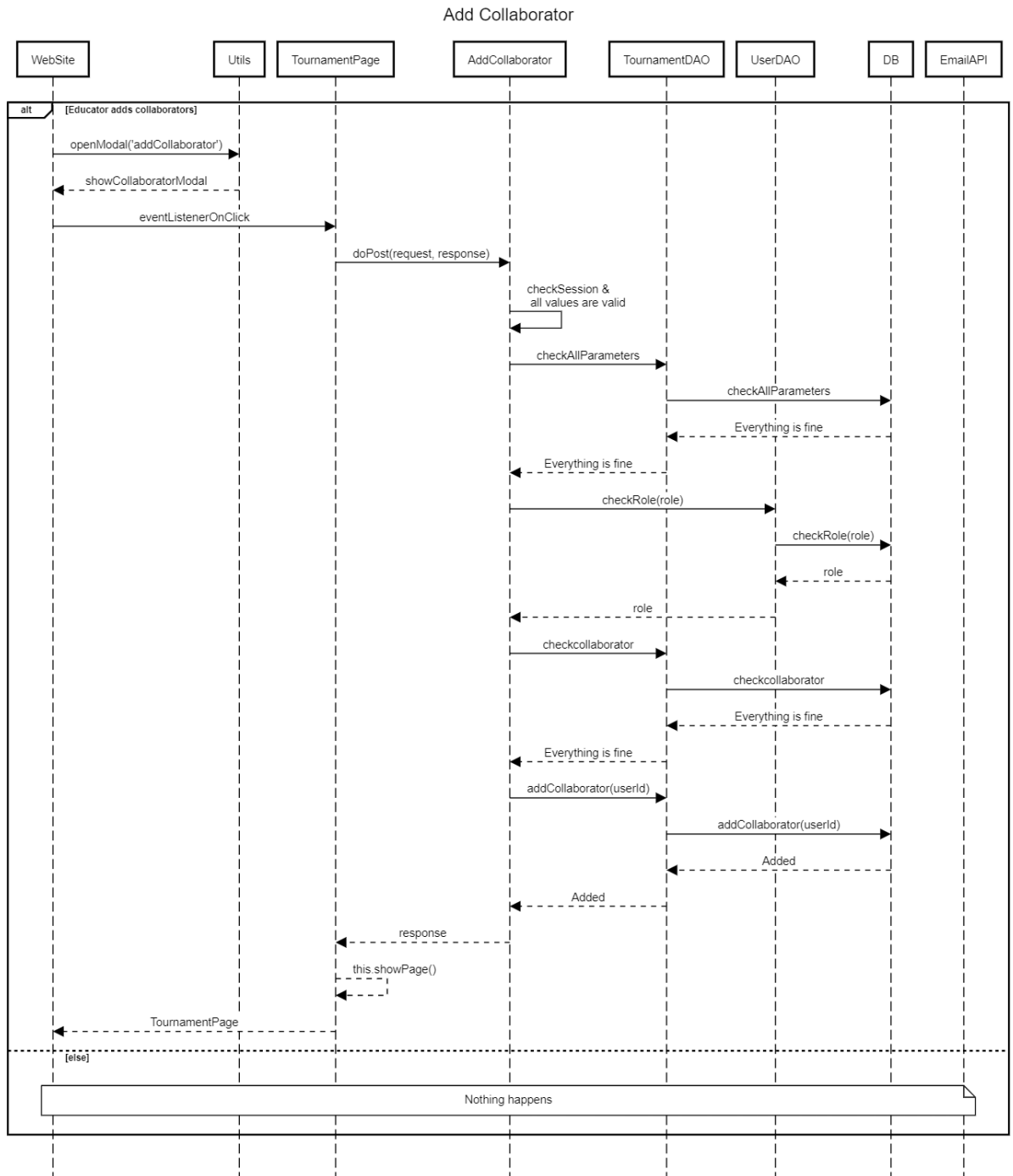
[SD3]. Creation of a tournament

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

When an Educator wishes to create a tournament, they fill in the required fields and send the request to the server, which saves it and sends an email notification to all students enrolled in the application to inform them of the creation of the new tournament. Next, the application shows the educator the page dedicated to the newly created tournament.

If the educator would like to add other educators as collaborators in managing the tournament, they can easily do so by clicking on the *Add Collaborator* button on the newly created tournament page and selecting the collaborators they would like to include.

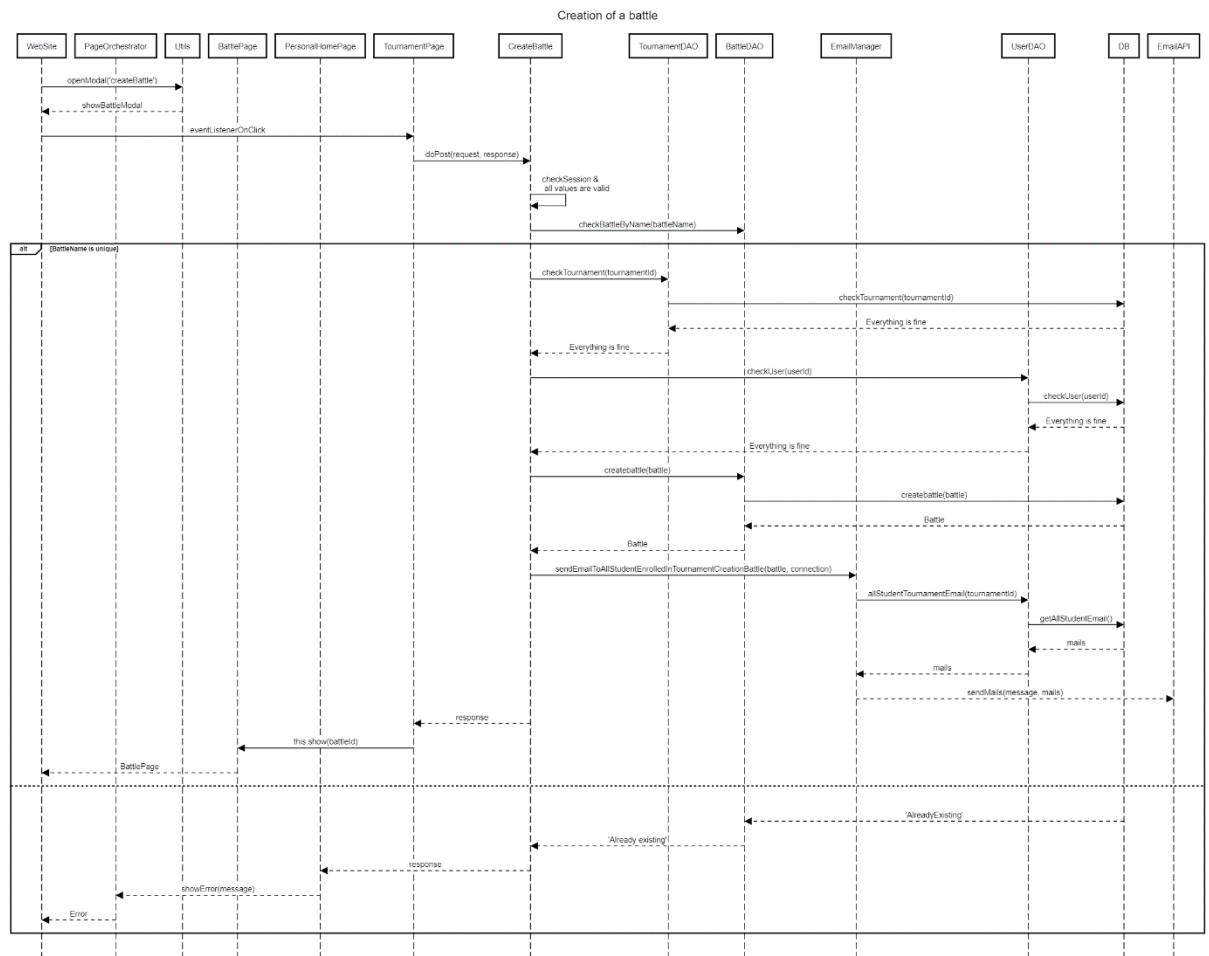




[SD4]. Creation of a battle

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

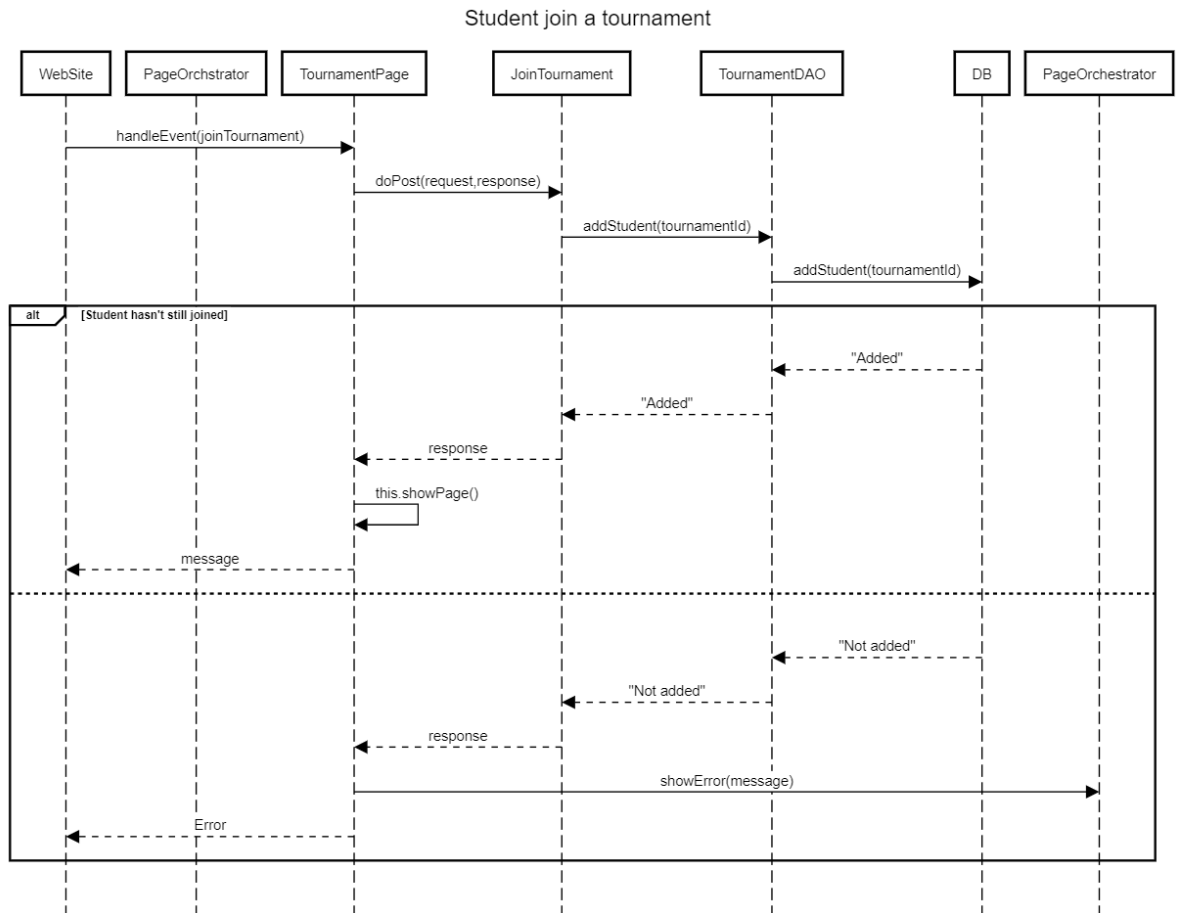
When an educator wishes to create a battle within a specific tournament, they fill in the required fields and sends the request to the server, which saves it and sends a notification via email to all students enrolled in the tournament to inform them of the creation of a new battle. Next, the application shows the educator the page dedicated to the newly created battle.



[SD5]. Student joins a tournament

This sequence diagram represents the interaction that occurs when a Student wants to join in a tournament in the CKB application

When a student wants to participate in a tournament, they access the page of the tournament of interest and click the *Join Student* button. Once the application displays an "Added" message, the student is officially enrolled in the tournament.

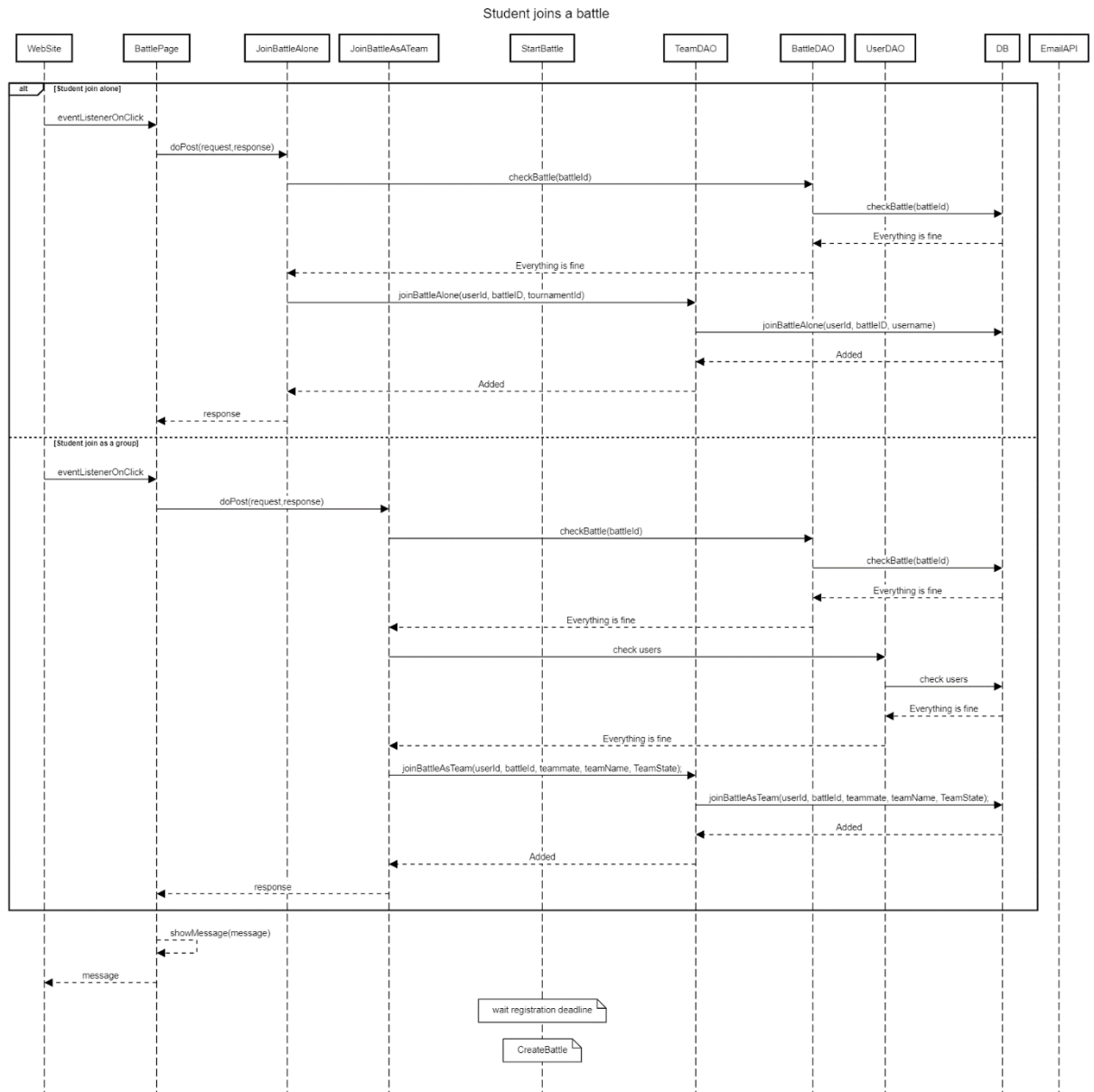


[SD6]. Student joins a battle

This sequence diagram represents the interaction that occurs when a Student wants to join in a battle in the CKB application

When a student wishes to participate in a battle, they access the page of the battle they are interested in. Within this page, there are two buttons: the first allows individual entry, if the battle allows it, while the second allows entry as a team. If the student opts for the second option, they will have to select the other students who will form their team.

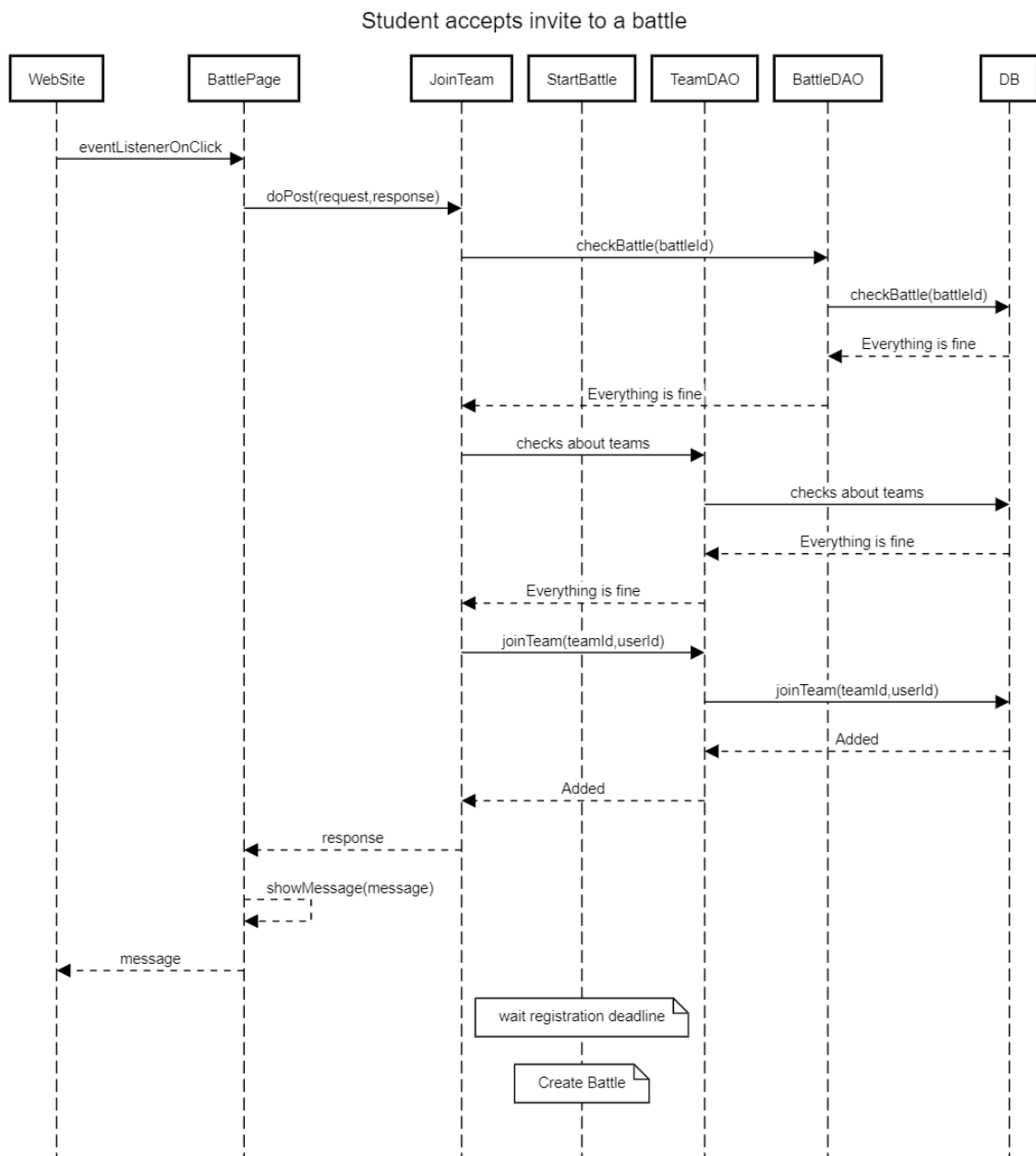
After confirming the team members, the application will save the team in the database.



[SD7]. Student accepts an invite for a battle

This sequence diagram represents the interaction that occurs when a Student wants to accept an invite for a battle in the CKB application

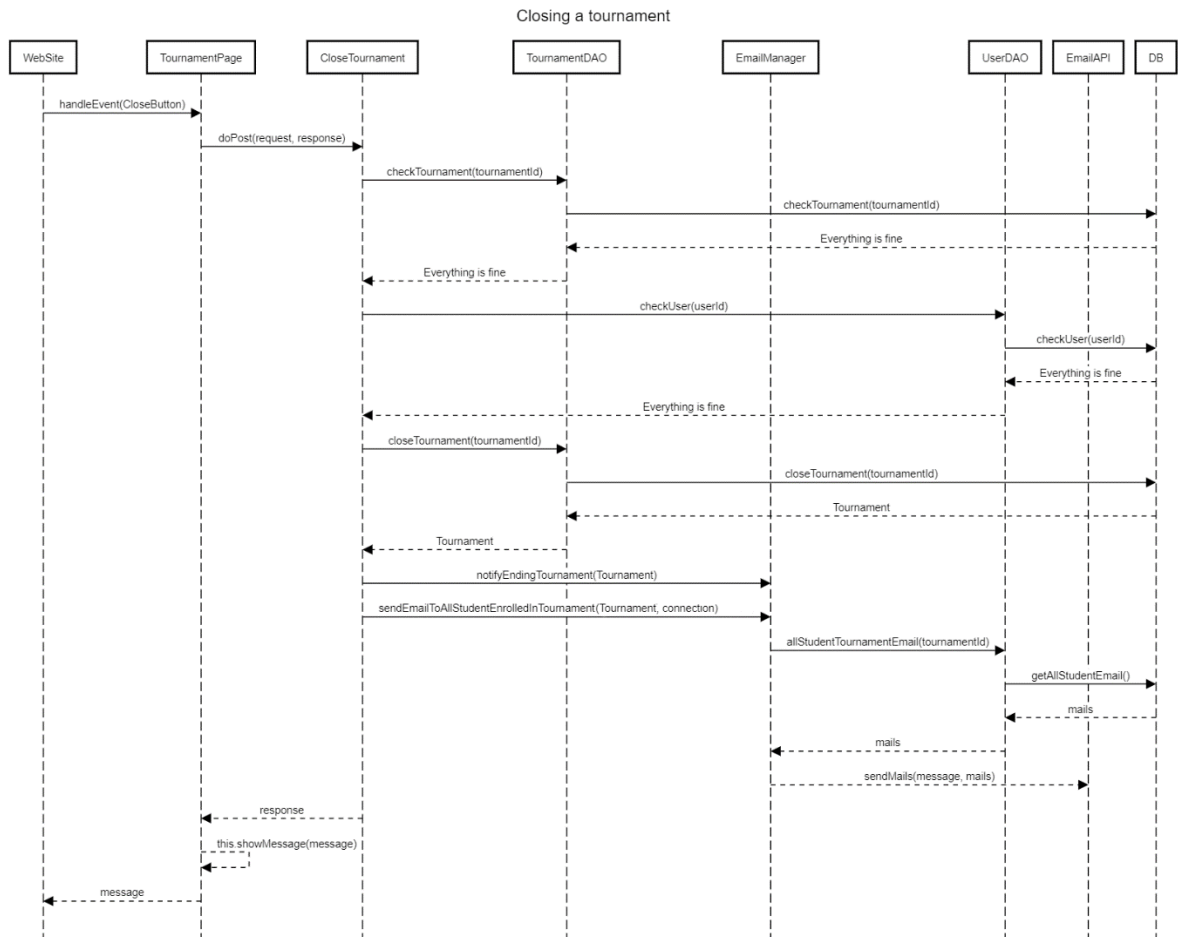
When a student wishes to accept an invitation to participate as a team together with a student who has invited them, they should go to the specific battle page and click the *Join Team* button that will show them all the teams that have invited them and they will select the one they prefer.



[SD8]. Closing a tournament

This sequence diagram represents the interaction that occurs when a Educator wants to close a tournament in the CKB application

When an educator wants to close a tournament they manage, they should go to the specific tournament page and click the *Close Tournament* button. By doing so, the application will save the status change for the tournament and send the students enrolled in it a notification regarding its closure.

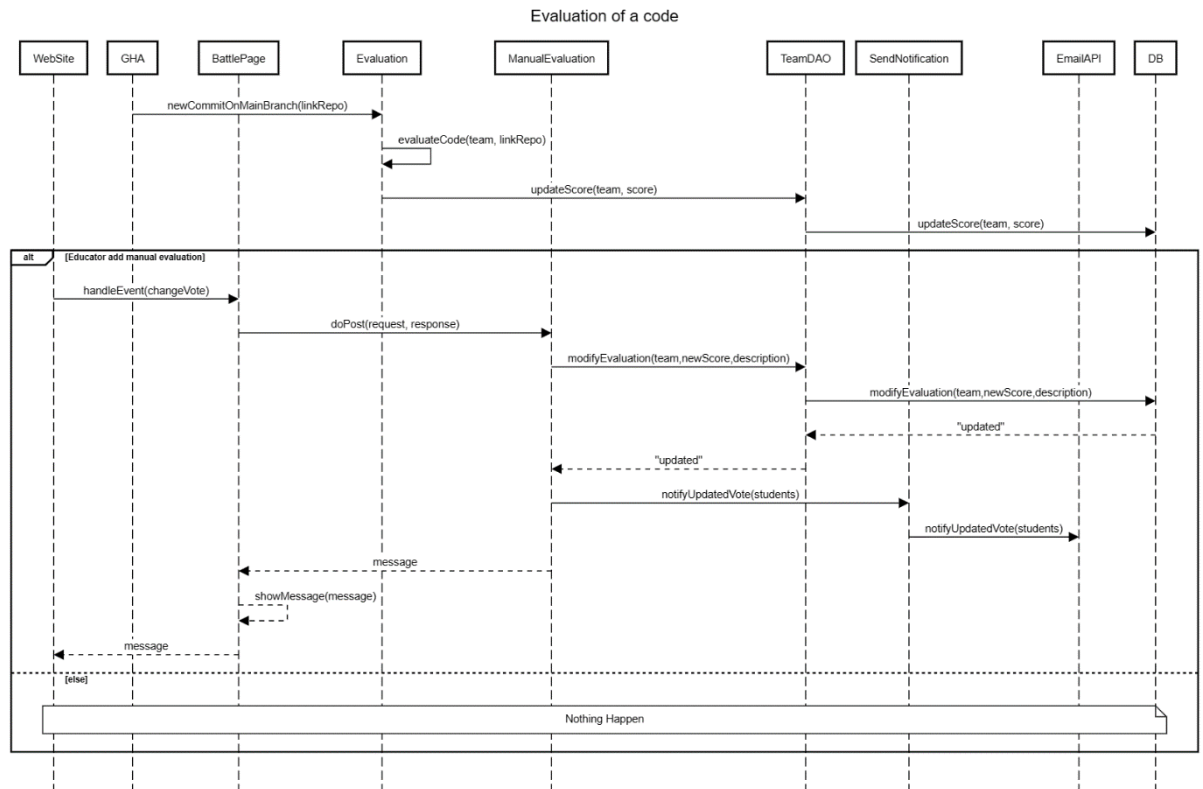


[SD9]. Evaluation of a code

This sequence diagram illustrates the interaction that occurs during an evaluation, whether automatic or manual, of a commit submission made by a team in the CKB application.

Whenever GHA sends a notification to the application regarding a commit made by a team to the main branch of their GH repo relating to a specific battle that is still in progress, the application will automatically perform an evaluation of the submission and assign a grade to it and store it in the database.

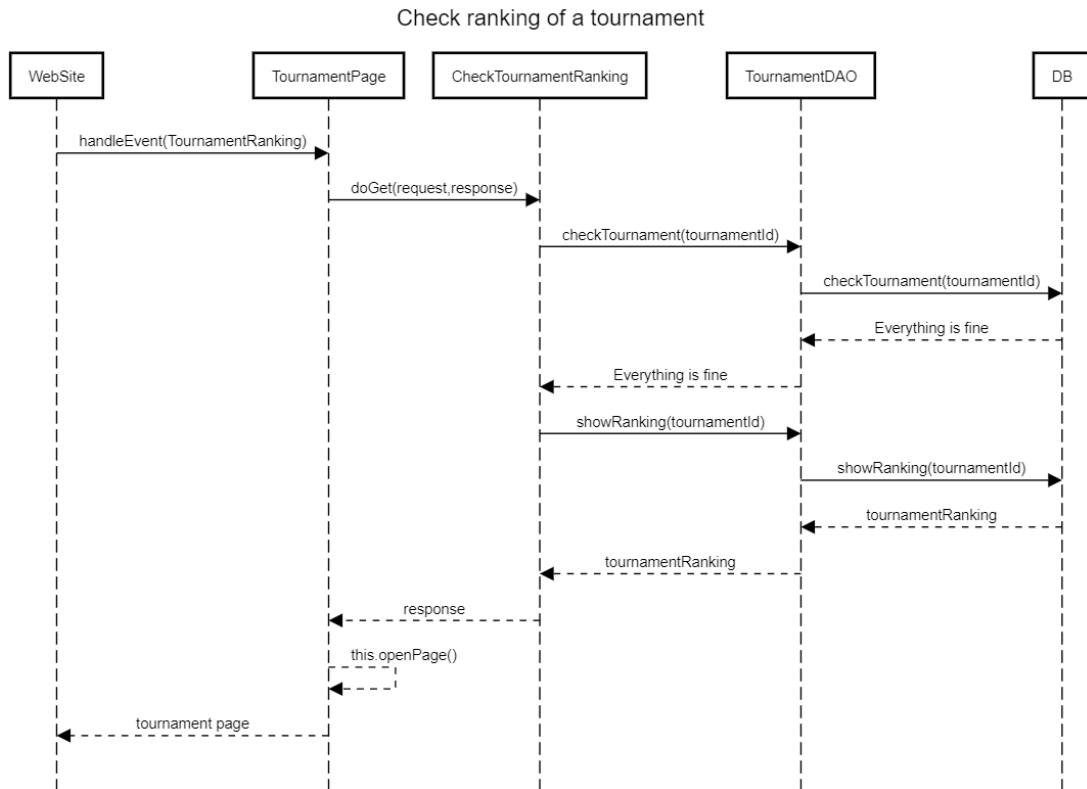
Should an educator wish to change the grade of a team entered in a battle, they may do so by going to the specific battle page and clicking on the *Manual Evaluation* button, this will display a form to allow the educator to select the team to be evaluated manually. Once the evaluation is complete, the application will save the grade and send a notification to the team, informing them that the educator has manually made changes to the grade.



[SD10]. Check ranking of a tournament

This sequence diagram represents the interaction that occurs when a User wants to check the ranking of a specific tournament in the CKB application.

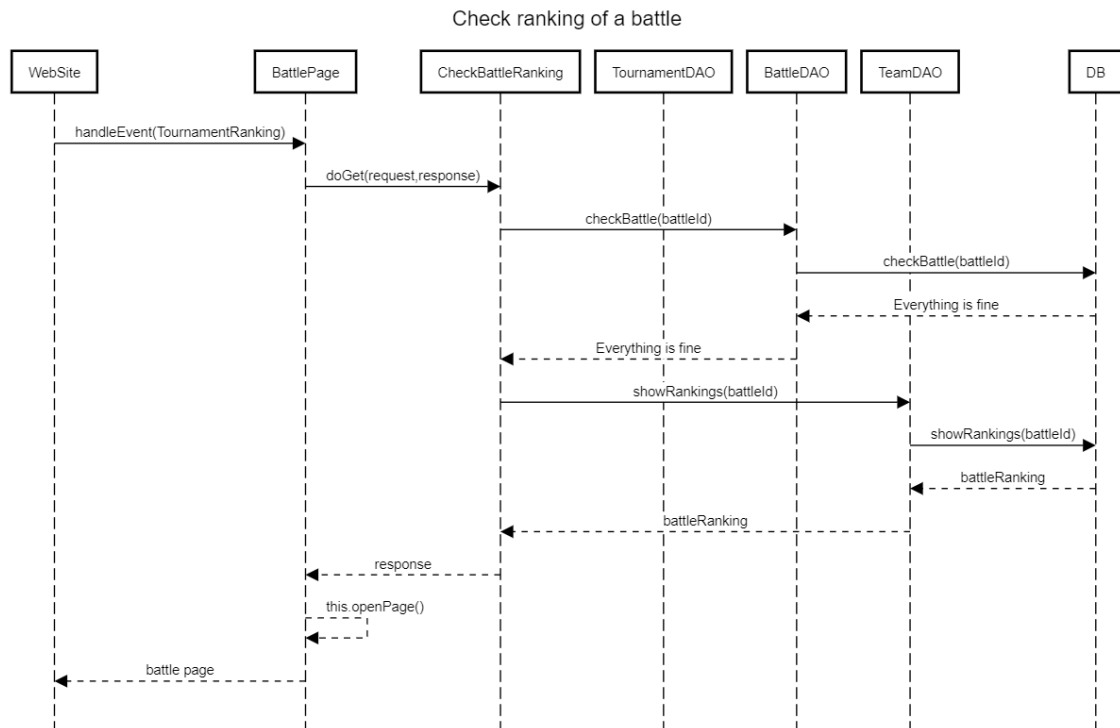
Each time a user enters the page of a specific tournament, it will show, in addition to the details of the tournament, the updated standings relating to the tournament's progress.



[SD11]. Check ranking of a battle

This sequence diagram represents the interaction that occurs when a User wants to check the ranking of a specific battle in the CKB application.

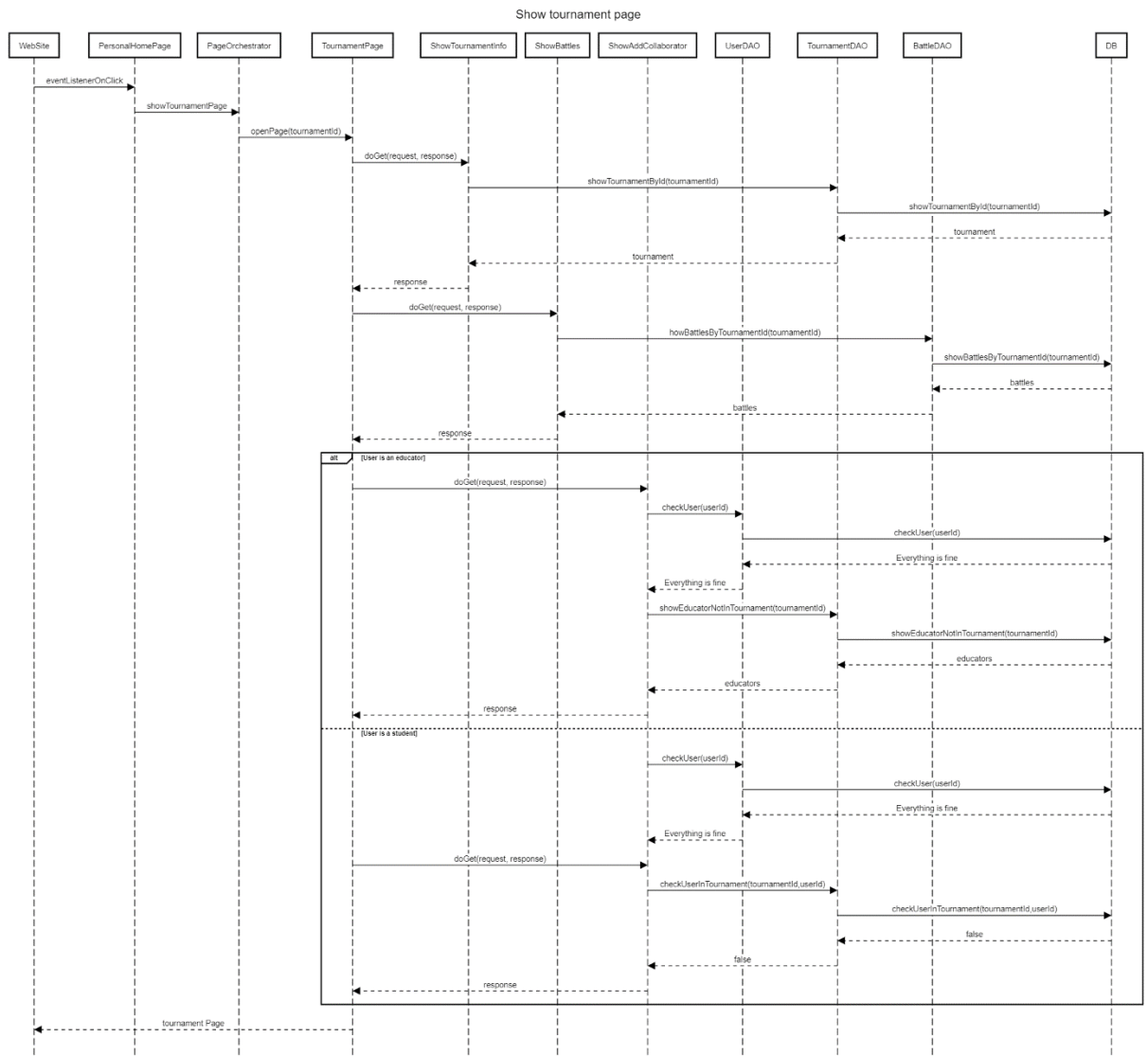
Each time a user enters the page of a specific battle, it will show, in addition to the details of the battle, the updated standings relating to the battle's progress.



[SD12]. Show tournament page

This sequence diagram represents the interaction that occurs when a User wants to show the specific tournament page in the CKB application.

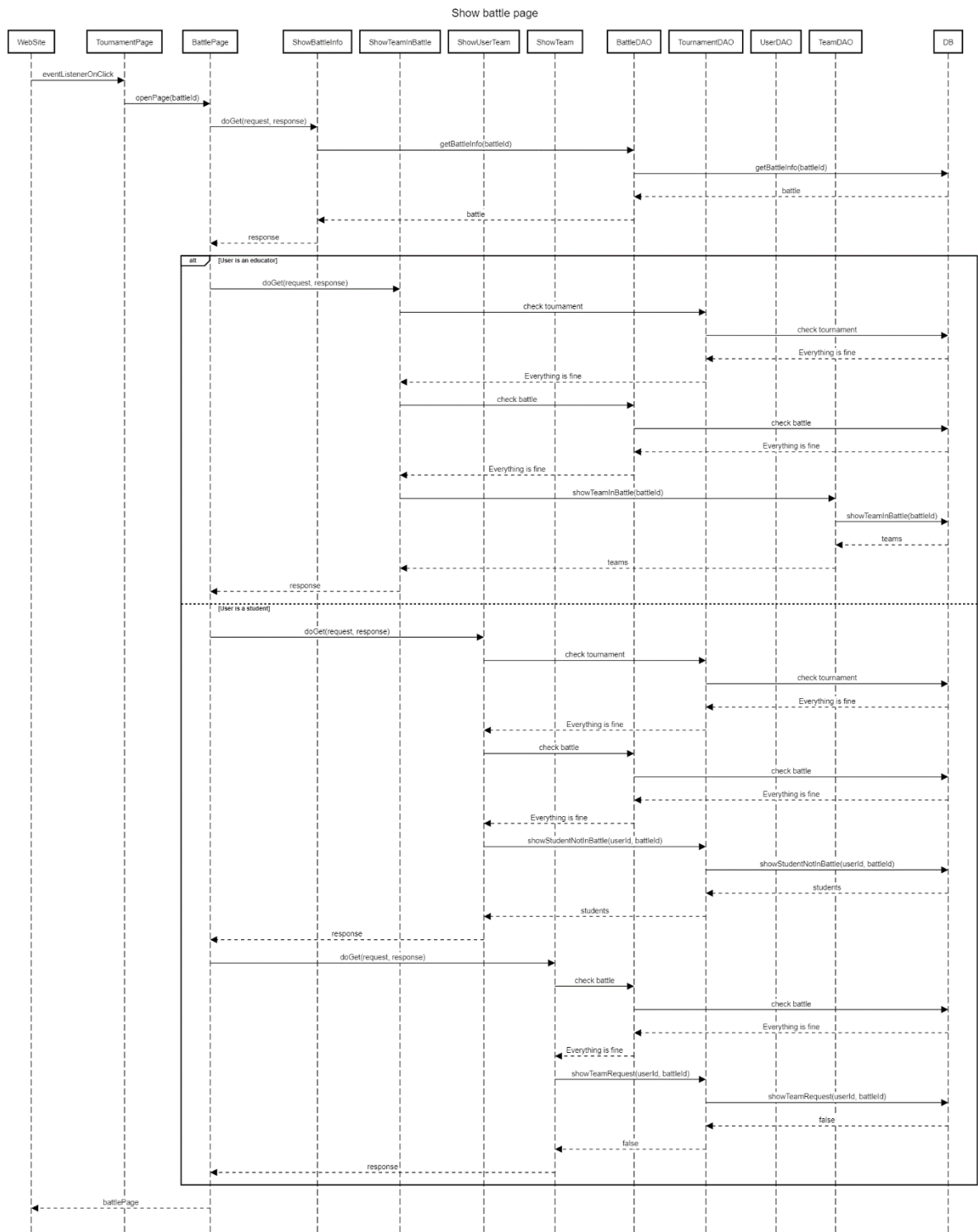
Whenever a user enters a specific tournament page, it will display all the details of a tournament according to the user's role.



[SD13]. Show battle page

This sequence diagram represents the interaction that occurs when a User wants to show the specific battle page in the CKB application.

Whenever a user enters a specific battle page, it will display all the details of a battle according to the user's role.



2.5. Component Interface

This section will list the interfaces of all the methods that each component provides.

- *Bean*: is an object that stores data that is retrieved from the database.
 - Battle
 - Ranking
 - SessionUser
 - Team
 - User
 - Tournament
 - User
- *DAO*: is an object that is responsible for interfacing with the database to request and modify data in it.
 - BattleDAO
 - `public ArrayList<Battle> showBattlesByTournamentId(int tournamentId) throws SQLException`
 - `public Battle showBattleById(int battleId) throws SQLException`
 - `public boolean checkBattleByName(String name) throws SQLException`
 - `public int getBattleId(String name) throws SQLException`
 - `public Boolean checkBattleNotStarted(int battleId) throws SQLException`
 - `public Boolean checkBattleOngoing(int battleId) throws SQLException`
 - `public Boolean checkEducatorManageBattle(int battleId, int userId) throws SQLException`
 - `public boolean createBattle(Battle battle) throws SQLException`
 - `public void startBattle() throws SQLException`
 - `public void closeBattle() throws SQLException`
 - `public ArrayList<Ranking> showRanking(int battleId) throws SQLException`

- TeamDAO

public List<Integer> getTeamInBattle(int battleId) throws SQLException

public List<Team> showTeamInBattle(int battleId) throws SQLException

public String getTeamName(int teamId) throws SQLException

public int getTeamId(int battleId, String teamName) throws SQLException

public boolean joinBattleAlone(int userId, int battleId, String username) throws SQLException

public boolean joinBattleAsTeam(int userId, int battleId, List<Integer> teammateId, String teamName, String phase) throws SQLException

public boolean checkStudentInOtherTeam(int userId, int battleId) throws SQLException

public List<SessionUser> showStudentNotInBattle(int userId, int battleId) throws SQLException

public boolean checkTeamName(String teamName, int battleId) throws SQLException

public List<Team> showTeamRequest(int userId, int battleId) throws SQLException

public void joinTeam(int teamId, int studentId) throws SQLException

public boolean updatePointTeam(int teamId, int point) throws SQLException

public boolean checkStudentHasTeamInvitation(int teamId, int studentId) throws SQLException

public Boolean checkTeamsInABattle(int battleId, int teamId) throws SQLException

- TournamentDAO

public ArrayList<Tournament> showTournamentByUserId (int userId) throws SQLException

public Tournament showTournamentById (int id) throws SQLException

public boolean checkTournamentByName (String name) throws SQLException

public boolean checkUserInTournament(int tournamentID, int userID) throws SQLException

public ArrayList<SessionUser> showEducatorNotInTournament(int tournamentID) throws SQLException

public void createTournament(Tournament tournament) throws SQLException

public void closeTournament(int tournamentID) throws SQLException

public boolean addCollaborator(int tournamentID, List<Integer> collaboratorUsernameList) throws SQLException

public ArrayList<Tournament> showAllTournamentsByString (String string) throws SQLException

public boolean joinTournament(int userId, int tournamentId) throws SQLException

public void startTournament() throws SQLException

public ArrayList<Ranking> showRanking(int tournamentId) throws SQLException

public boolean checkBattleNotClosed(int tournamentId) throws SQLException

- UserDAO

public SessionUser checkUsername(String username, String password) throws SQLException

public Boolean checkRole(int id) throws SQLException

public Boolean checkStudent(int id) throws SQLException

public List<String> allStudentEmail() throws SQLException

public List<String> allStudentTournamentEmail(int tournamentID) throws SQLException

public List<String> allEducatorTournamentEmail(int tournamentID) throws SQLException

public List<String> allStudentBattleEmail(int battleId) throws SQLException

public List<String> allStudentBattleGitHub(int teamId) throws SQLException

public int createUser(User user) throws SQLException

public User getUserById(int id) throws SQLException

- *Servlet*: is a Java object that extends the functionality of a Web server to provide dynamic services to users through interaction based on requests and responses.

- SignInManager

<i>Type</i>	POST
<i>Req body</i>	username: [String], password: [String], role: [int], name: [String], surname: [String], birthdate: [Date], email: [String], githubUser: [String]
<i>Success code</i>	code 200: OK message: {Registration has gone successfully you will receive a confirmation email}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 500: Error body: {"error": "The server does not respond"}

- GetUser

<i>Type</i>	GET
<i>Req body</i>	user: [SessionUser]
<i>Success code</i>	code 200: OK
<i>Error code</i>	code: 401: Error body: {"error": "You can't access to this page"} code: 404: Error body: {"error": "User selected doesn't exist"} code: 500: Error body: {"error": "The server does not respond"}

- LoginManager

<i>Type</i>	POST
<i>Req body</i>	username: [String], password: [String]
<i>Success code</i>	code 200: OK
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code 401: Error body: {"error": "Incorrect credential"} code: 500: Error body: {"error": "The server does not respond"}

- ManageTournament: within this service are the servlets that compose it.

- ShowTournament

<i>Type</i>	GET
<i>Req body</i>	userID: [int]
<i>Success code</i>	code 200: OK body: {Tournament: tournament}
<i>Error code</i>	code 401: Error body: {"error": "Incorrect credential"} code: 500: Error body: {"error": "The server does not respond"}

- ShowTournamentInfo

<i>Type</i>	GET
<i>Req body</i>	tournamentID: [int]
<i>Success code</i>	code 200: OK body: {Tournament: tournament}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code 401: Error body: {"error": "Incorrect credential"} code: 403: Error body: {"error": "User doesn't have the authorization to do that"} code: 404 Error body: {"error": "Tournament not found"} code: 500: Error body: {"error": "The server does not respond"}

- CheckTournamentRanking

<i>Type</i>	GET
<i>Req body</i>	tournamentID: [int]
<i>Success code</i>	code 200: OK body: {TournamentRanking: List<User>}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code 401: Error body: {"error": "Incorrect credential"} code: 404 Error body: {"error": "Tournament not found"} code: 500: Error body: {"error": "The server does not respond"}

- CreateTournament

<i>Type</i>	POST
<i>Req body</i>	Name: [String], Description: [String], RegistrationDeadline: [DateTime]
<i>Success code</i>	code 200: OK body: {tournamentId: int}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 409: Error body: {"error": "Existing tournament name"} code: 500: Error body: {"error": "The server does not respond"}

- CloseTournament

Type	POST
Req body	tournamentID: [int]
Success code	code 200: OK body: {message: String}
Error code	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 406 Error body: {"error": "The tournament is not closable now"} code: 409: Error body: {"error": "The tournament has battle not closed"} code: 500: Error body: {"error": "The server does not respond"}

- ShowAddCollaborator

Type	GET
Req body	userID: [int], tournamentID: [int]
Success code	code 200: OK body: {message: String}
Error code	code: 400 Error body: {"error": "Malformed request"} code: 401 Error body: {"error": "User not authorized"} code: 500: Error body: {"error": "The server does not respond"}

- AddCollaborator

<i>Type</i>	POST
<i>Req body</i>	userID: [int], tournamentID: [int]
<i>Success code</i>	code 200: OK body: {message: String}
<i>Error code</i>	code: 400 Error body: {"error": "Malformed request"} code: 401 Error body: {"error": "User not authorized"} code: 406 Error body: {"error": "The tournament has already been closed"} code: 409: Error body: {"error": "There is a conflict"} code: 500: Error body: {"error": "The server does not respond"}

- ShowJoinTournament

<i>Type</i>	GET
<i>Req body</i>	userID: [int], tournamentID: [int]
<i>Success code</i>	code 200: OK body: {message: String}
<i>Error code</i>	code: 400 Error body: {"error": "Malformed request"} code: 401 Error body: {"error": "User not authorized"} code: 500: Error body: {"error": "The server does not respond"}

- JoinTournament

<i>Type</i>	POST
<i>Req body</i>	tournamentID: [int], userID: [int]
<i>Success code</i>	code 200: OK body: {message: String}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 500: Error body: {"error": "The server does not respond"}

- SearchTournament

<i>Type</i>	POST
<i>Req body</i>	tournament: [String]
<i>Success code</i>	code 200: OK body: {message: String}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 500: Error body: {"error": "The server does not respond"}

- LogoutManager

<i>Type</i>	POST
<i>Req body</i>	username: [String]
<i>Success code</i>	code 200: OK body: {message: String}

- ModifyPersonalData

<i>Type</i>	POST
<i>Req body</i>	role: [int], name: [String], surname: [String], birthdate: [Date], username: [String], email: [String], password: [String]
<i>Success code</i>	code 200: OK body: {message: String}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 404 Error body: {"error": "User not found"} code: 409 Error body: {"error": "The current password not correspond"} code: 500: Error body: {"error": "The server does not respond"}

- ManageBattle: within this service are the servlets that compose it.

- ShowBattles

<i>Type</i>	GET
<i>Req body</i>	tournamenID: [int]
<i>Success code</i>	code 200: OK body: {BattleList: List<battle>}
<i>Error code</i>	code: 401: Error body: {"error": "User not authorized"} code: 500: Error body: {"error": "The server does not respond"}

- ShowBattleInfo

<i>Type</i>	GET
<i>Req body</i>	battleID: [int]
<i>Success code</i>	code 200: OK body: {Battle: battle}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 404 Error body: {"error": "Battle not found"} code: 500: Error body: {"error": "The server does not respond"}

- CheckBattleRanking

<i>Type</i>	GET
<i>Req body</i>	battleID: [int]
<i>Success code</i>	code 200: OK body: {Ranking: List<User>}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 404 Error body: {"error": "Battle not found"} code: 500: Error body: {"error": "The server does not respond"}

- CreateBattle

<i>Type</i>	POST
<i>Req body</i>	name: [String], description: [String], regDeadline: [DateTime], subDeadline: [DateTime], minMember: [int], maxMember: [int], CodeKata: [File], testcase: [File]
<i>Success code</i>	code 200: OK body: {message: String}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 406 Error body: {"error": "The tournament has not started yet"} code: 409 Error body: {"error": "Existing battle name"} code: 500: Error body: {"error": "The server does not respond"}

- JoinBattleAlone

<i>Type</i>	POST
<i>Req body</i>	battleID: [int], userID: [int]
<i>Success code</i>	code 200: OK body: {message: String}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 406 Error body: {"error": "Request is not acceptable"} code: 409 Error body: {"error": "You are already signed up to another team"} code: 500: Error body: {"error": "The server does not respond"}

- ManageTeam: within this service are the servlets that compose it.

- ShowTeamInBattle

<i>Type</i>	GET
<i>Req body</i>	userID: [int], battleID: [int]
<i>Success code</i>	code 200: OK body: {message: String}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 406 Error body: {"error": "Request is not acceptable"} code: 500: Error body: {"error": "The server does not respond"}

- JoinBattleAsTeam

Type	POST
Req body	battleID: [int], teamName: [String], userID: [int], teammates: [List<User>]
Success code	code 200: OK body: {message: String}
Error code	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 406 Error body: {"error": "Request is not acceptable"} code: 409 Error body: {"error": "There is a conflict"} code: 500: Error body: {"error": "The server does not respond"}

- ShowTeam

Type	GET
Req body	userID: [int], battleID: [int]
Success code	code 200: OK body: {message: String}
Error code	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 406 Error body: {"error": "Request is not acceptable"} code: 500: Error body: {"error": "The server does not respond"}

- JoinTeam

<i>Type</i>	POST
<i>Req body</i>	battleID: [int], teamID: [int]
<i>Success code</i>	code 200: OK body: {message: String}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 406 Error body: {"error": "Request is not acceptable"} code: 409 Error body: {"error": "You are already signed up to another team"} code: 500: Error body: {"error": "The server does not respond"}

- ShowUserTeam

<i>Type</i>	GET
<i>Req body</i>	battleID: [int]
<i>Success code</i>	code 200: OK body: {message: String}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 406 Error body: {"error": "Request is not acceptable"} code: 500: Error body: {"error": "The server does not respond"}

- **ModifyGrade**

<i>Type</i>	POST
<i>Req body</i>	battleID: [int], teamID: [int]
<i>Success code</i>	code 200: OK body: {message: String}
<i>Error code</i>	code: 400: Error body: {"error": "Malformed request"} code: 401: Error body: {"error": "User not authorized"} code: 406 Error body: {"error": "Request is not acceptable"} code: 500: Error body: {"error": "The server does not respond"}

- *JavaScript web page*: is an object that manages user interaction on a web page by dynamically changing its visual elements in response to events that occur during the interaction.
 - *HomePage*
 - this.show()
 - this.showError(String message)
 - *PersonalHomePage*
 - function search(form)
 - function creationTournament(form)
 - this.updateTournamentTable(tournaments, isSearch)
 - this.openPage()
 - this.hide()
 - *PersonalInfoPage*
 - this.updatePersonalInfo(info)
 - this.openPage()
 - this.hide()

- TournamentPage
 - function closingTournament()
 - this.showAddCollaborator()
 - this.updateTournamentInfo(tournament)
 - this.hideTournamentButton(tournament)
 - this.updateBattleTable(battles)
 - this.updateCollaboratorList(collaborators)
 - this.updateRankingTable(rankings)
 - this.openPage(id)
 - this.hide()
- BattlePage
 - function updateTeamsGrade(teams)
 - function updateTeammates(teammates)
 - function updateTeams(teams)
 - function hideAllButton()
 - this.updateBattleInfo(battle)
 - this.updateRankingTable(rankings)
 - this.openPage()
 - this.hide()
- PageOrchestrator
 - function PageOrchestrator()
- Utils
 - function makeCall(method, url, formElement, cback)
 - function openModal(type)
 - function closeModal()
 - function clearForm(type)
- *Other classes*: these classes are internal to the system and do not interact directly with the external user.
 - EncryptorTripleDES
 - public String encrypt(String input)

- FolderManager
 - `public static String getFileName(Part part)`
 - `public static String getFileExtension(Part part)`
 - `public static void saveFile(Part part)`
 - `public static void saveFile(Part part, String directoryToSave)`
 - `public static void deleteDirectory(File directoryToBeDeleted)`
 - `public static void writeFile(File fileToWrite, String text)`
- ZipFolderManager
 - `public static void unzip(String zipFileName)`
 - `public static void deleteZipFile(String zipFileName)`
- Evaluation

This class is responsible for automatically generating the evaluation based on the execution data received from GHA.
- StartTournament

This is a ServletContextListener in charge of periodically checking the database to see if a tournament has reached the registration deadline. In case this condition is met, the listener initiates the necessary procedures to start the tournament.
- StartBattle

This is a ServletContextListener in charge of periodically checking the database to see if a battle has reached the registration deadline. In case this condition is met, the listener initiates the necessary procedures to start the battle.

It is also in charge of managing the closure of a battle when it exceeds the submission deadline.
- GitHubManager
 - `public static void createGitHubRepository(String repositoryName, boolean isPrivate)`
 - `public static void uploadFolderOnGitHubRepository(String projectPath, String repositoryURL)`
 - `public static void createGitHubRepositoryPerTeam(int teamId, String codeKata, Connection connection)`
- ReceiveFromGitHubServlet

This servlet takes care of receiving messages from GH and if the battle is still in progress saves the automatic evaluation received from the team to the database.

- EmailManager

```
public static boolean isValidEmail(String email)
```

```
public static void sendEmailToAllStudentNewTournamentCreated(String tournamentCreator, Timestamp time, Connection connection)
```

```
public static void sendEmailToAllStudentEnrolledInTournamentClosed(Tournament tournament, Connection connection)
```

```
public static void sendEmailToAllStudentEnrolledInTournamentCreationBattle(Battle battle, Connection connection)
```

```
public static void sendEmailToAllStudentEnrolledInTournamentStarted(int tournamentId, Connection connection)
```

```
public static void sendEmailToAllCollaboratorInTournamentClosed(int tournamentId, Connection connection)
```

```
public static void sendEmailToAllStudentBattleClosed(int battleId, Connection connection)
```

2.6. Architectural styles and patterns

2.6.1. Architectural style

The implementation of CKB will take place following a three-tier architecture; this structure confers numerous advantages due to the modular division of the system into three distinct and autonomous layers.

1. *Presentation Tier*: the tier responsible for interfacing with the client, by changing or replacing the displayed pages, will receive the data sent by the *Business Logic Tier*. This data will be adapted so that it is visible and understandable to the user.
2. *Business Logic Tier*: this tier includes the application of business logic, which is responsible for performing calculations and making decisions based on the data stored in the data tier. Later, the results will be shown in the presentation tier.
3. *Data Tier*: this tier is responsible for storing, managing, and making accessible the data that will be used by the business logic tier.

By adopting this architecture, it is possible to modify one part of the system without involving the others. Using an intermediate tier to manage the data in the database provides an additional layer of security because users cannot directly modify the saved data. To modify the data in the database they must necessarily interact with the Business Logic Tier which imposes specific formats on the data.

2.6.2. Patterns

2.6.2.1. Model-View-Controller Pattern

We will adopt the Model-View-Controller (MVC) pattern to organize and structure the code. In the context of MVC, the Model assumes the crucial role of storing essential objects during computation and decision making, with responsibilities given to *Data Access Objects* (DAOs). These components ensure the accurate management of the underlying data. On the other hand, *web pages* constitute the components dedicated to presenting information resulting from decisions made by servlets in a clear and understandable way to the user. *Servlets*, grouped under the Controller category, emerge as key figures responsible for reading data from the Model and orchestrating changes in the View. This functional subdivision, typical of MVC, facilitates maintenance, scalability and code comprehensibility, contributing to effective management of business logic and user interactions within the application.

2.6.2.2. Observer Pattern

We will adopt the Observer design pattern to notify users about relevant changes in the application, such as the creation or the closure of a tournament and creation of a battle. Each class responsible for issuing notifications will send its own messages to the *SendNotification* class, which, acting as an intermediary, will interface with the email API to send notifications to interested clients. The *SendNotification* class acts as a coordination point, playing a key role in orchestrating communication between the application and users, thus ensuring an efficient and effective flow of relevant information.

2.6.2.3. Command Pattern

We will adopt the Command pattern so that each individual action or command is encapsulated in an object that can be handled by distinct classes. This approach offers a significant advantage: the invoker, or caller of the action, does not need to know the implementation details of the command it is calling, only its interface. This separation allows the calling functions to remain independent of the classes involved in creating and handling the commands. In this way, the interface between the invoker and the command remains stable, facilitating code maintenance and allowing new commands to be introduced without modifying existing parts of the system. This design pattern is particularly beneficial in ensuring a clear separation of responsibilities and helps make the code more modular, flexible, and easily extensible.

2.7. Others design decision

In this section are presented some of the design decisions made for the system to make work as expected.

2.7.1. Availability

We opted to implement load balancing and server replication in order to enhance system availability and reliability. The adoption of load balancing allows our system to respond more promptly by distributing requests among less stressed servers. This results in higher operational efficiency which ensures faster response to users. In parallel, through server replication, we are able to improve system resilience: if one server fails, service can be maintained unaffected by using operational replicas on other servers. This replication strategy helps to ensure continuity of service without significant interruptions, improving the robustness of our system in the face of any hardware failure.

2.7.2. Notification timed

The role of managing the sending of all notifications from the application to users is entrusted to the *SendNotification* component. This component interacts directly with the email API, facilitating the sending of notifications to the specified recipients.

2.7.3. Data storage

The decision to adopt a relational database, rather than a nonrelational one, was driven by the need to efficiently represent the inherent relationships present in the data to be stored. A relational database offers a structured model that lends itself well to the representation and management of connections between different entities. In addition, the use of a relational database greatly simplifies complex operations, such as joining multiple tables and updates within the database itself. The tabular structure of the relational database provides a clear organization of the data, making it easier to navigate and manipulate information. This choice results in more efficient management of complex data relationships, contributing to the consistency and optimal structuring of our storage system.

2.7.4. Security

Our system is provided with three firewalls, which are crucial for the protection of the different components of the application. Given the shared nature of the system, which will not be limited to a single machine, the implementation of these firewalls is essential. The external network, deemed untrusted, could pose a potential threat to the application LAN. The introduction of firewalls creates a demilitarized zone (DMZ), providing security and protection within the application environment. This configuration helps mitigate risks from unauthorized access and provides an additional layer of security, protecting the internal network from potential threats from outside.

3. USER INTERFACE DESIGN

In this chapter, we will show and elaborate on the user interface previously described in RASD, providing a detailed presentation and explanation of it.

The first page a user will see as soon as they enter the web application will be the HomePage. On this page the user will have the option to register on the application while those who already have an account will be able to log in directly.

Below, through Figures 5, 6, 7, images of the HomePage and the forms that allow registration and login to the application will be shown.

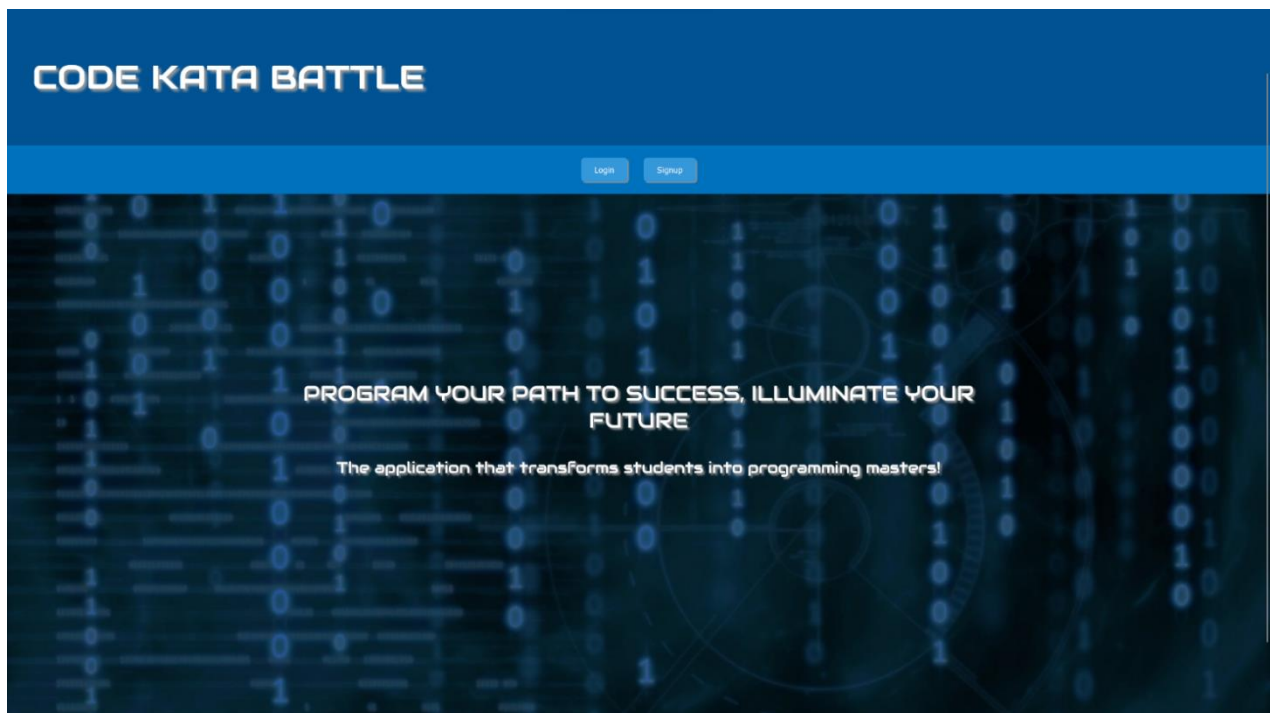
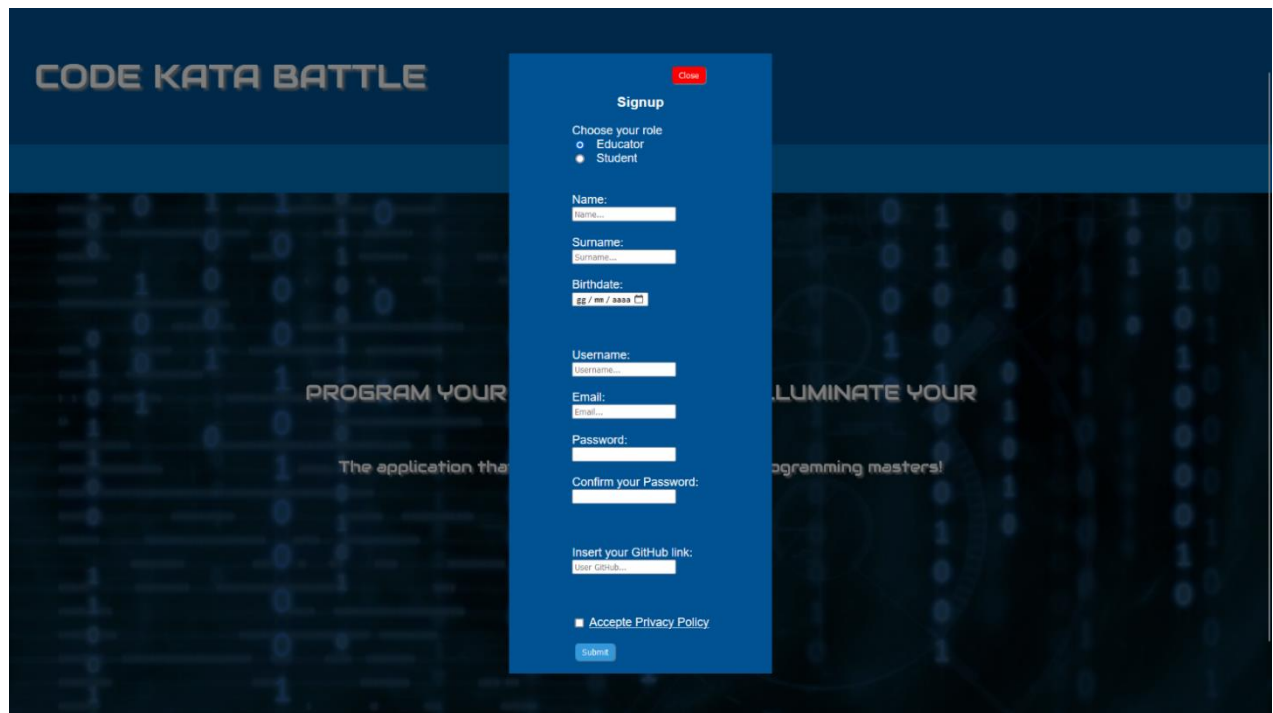


Figure 5 Home Page



The image shows a web application interface for 'CODE KATA BATTLE'. The background is dark blue with a pattern of binary code (0s and 1s) and faint circular patterns. The main heading 'CODE KATA BATTLE' is in a light blue, sans-serif font at the top left. Below it, there's a horizontal bar with the text 'PROGRAM YOUR FUTURE' and 'ILLUMINATE YOUR SKILLS' on either side of a central graphic. The central graphic is a stylized 'K' made of binary code. Below this bar, there's a text line: 'The application that transforms students into programming masters!'. In the center, there's a blue modal box for the 'Signup' form. The form has a 'Close' button in the top right corner. It starts with 'Choose your role' and two radio buttons: 'Educator' and 'Student' (which is selected). Below this are input fields for 'Name:', 'Surname:', 'Birthdate:' (with a date picker showing '22 / 09 / 2000'), 'Username:', 'Email:', 'Password:', and 'Confirm your Password:'. At the bottom of the form is a checkbox for 'Accept Privacy Policy' and a 'Submit' button.

CODE KATA BATTLE

Close

Signup

Choose your role

☐ Educator

☒ Student

Name:

Surname:

Birthdate:

22 / 09 / 2000

Username:

Email:

Password:

Confirm your Password:

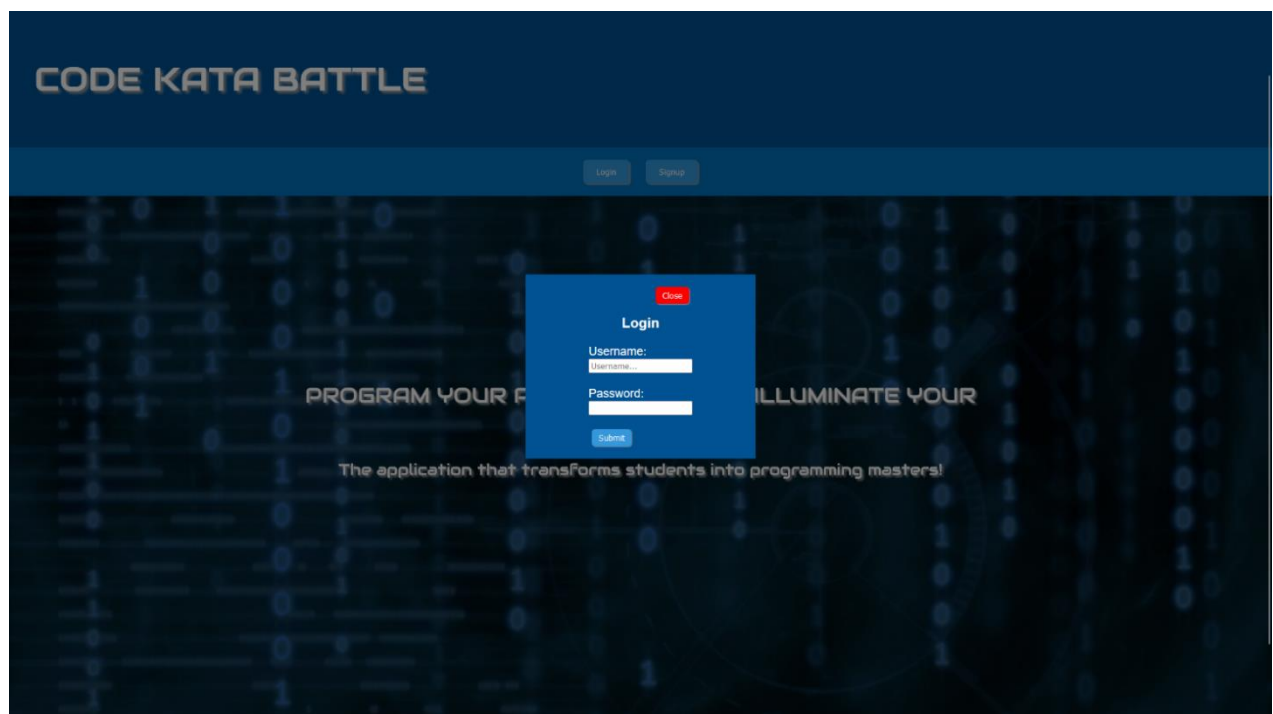
Insert your GitHub link:

User GitHub:

☐ Accept Privacy Policy

Submit

Figure 6 Signup Form



The image shows the same web application interface as Figure 6, but with the 'Login' form modal box open. The background and header elements are identical. The 'Login' modal box is blue and has a 'Close' button in the top right corner. It contains input fields for 'Username:' and 'Password:', followed by a 'Submit' button. Above the modal box, there are 'Login' and 'Signup' buttons in the header area.

CODE KATA BATTLE

Login Signup

Close

Login

Username:

Password:

Submit

PROGRAM YOUR FUTURE

ILLUMINATE YOUR SKILLS

The application that transforms students into programming masters!

Figure 7 Login Form

Once a user has logged into the application, his or her personalized main page will be shown.

If the user had registered as an educator, the page will show the tournaments he or she created and those in which he or she collaborates with other educators. In addition, a button called *Create New Tournament* will be visible that will allow him, by filling out the appropriate form, to create a new tournament.

In case the user has registered as a student, the page will show only the tournaments in which he is registered.

Below, through Figures 8, 9, 10, the main pages of the users according to their roles and the form that allows an educator to create a new tournament are shown.

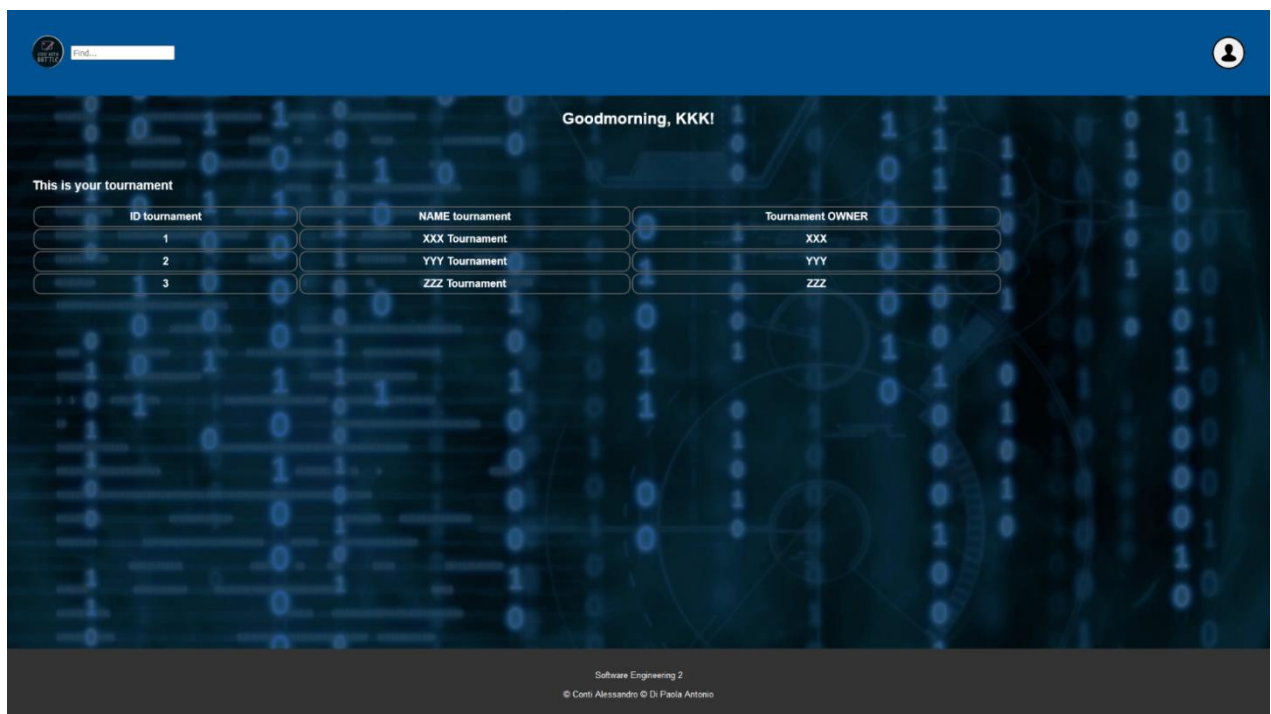


Figure 8 Student Home Page

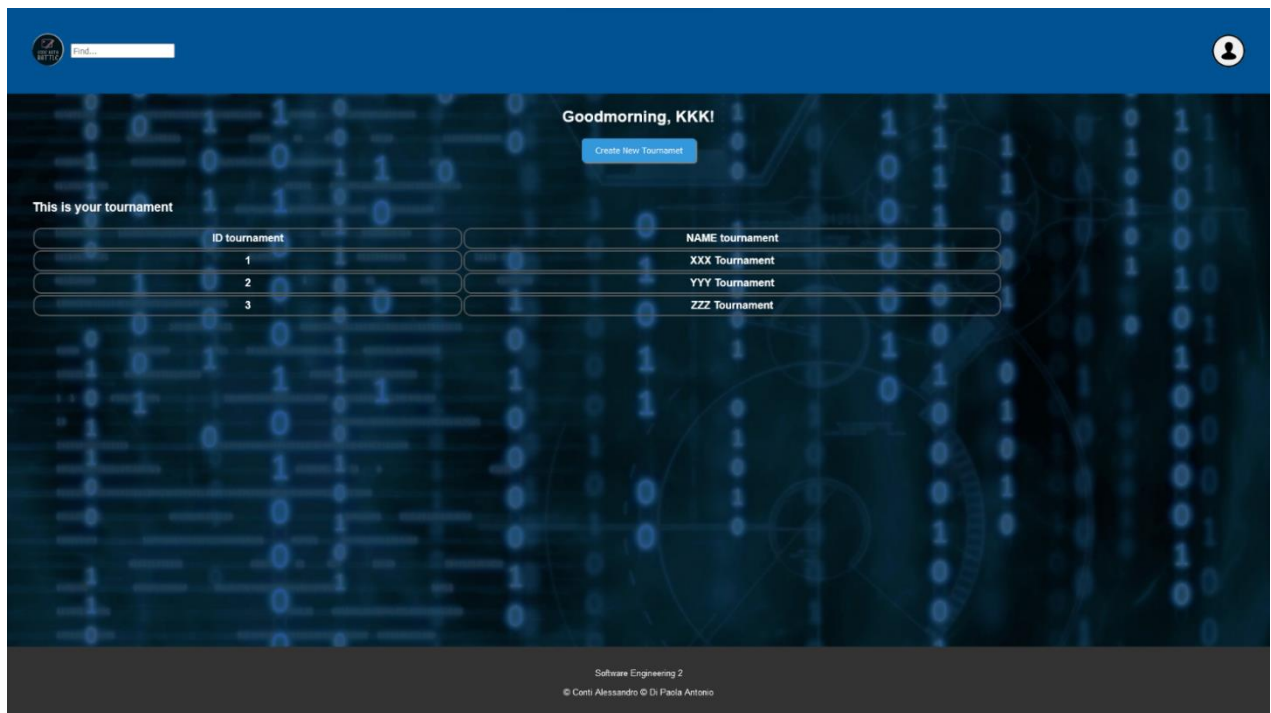


Figure 9 Educator Home Page

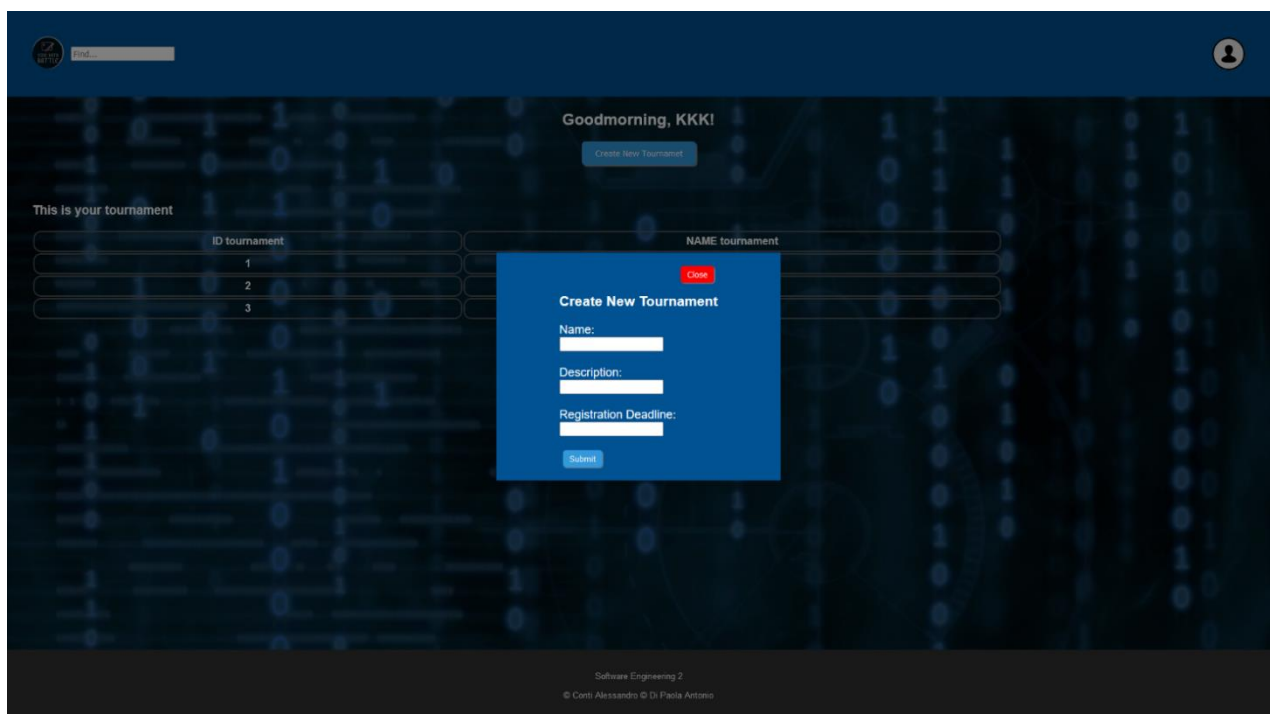


Figure 10 Create New Tournament Form

When a user enters the page of a tournament he will find the details of it, the list of all battles, and the overall standings of that tournament.

In case the user is an educator, three buttons will also be shown that will allow him to manage the tournament in case it is not finished. The first is "Create New Battle" which will allow him to create a new battle to check a certain competency of the students enrolled in the tournament, "Add Collaborator" to add an educator to collaborate in the tournament, and finally "Close Tournament" which allows him to close the tournament.

Instead, if the user is a student, the "Join Tournament" button will be shown, which will allow him/her to join the tournament, in case the tournament is not ending

Below, through Figures 11, 12, 13, 14, is shown the tournament page according to the user's role and the relevant forms they will need to fill out to perform the actions described above.

Goodmorning, KKK!

Join Tournament

This is the XXX tournament

This tournament allows you to improve your skills in the XXX programming language.

Registration Deadline: YYYY/MM/DD

Submission Deadline: YYYY/MM/DD

ID battle	NAME battle
1	YYY Battle
2	ZZZ Battle

Tournament Ranking

Position	Username
1	XXX
2	YYY
3	ZZZ
4	KKK
5	WWW
6	VVV

Software Engineering 2
© Conti Alessandro © Di Paola Antonio

Figure 11 Tournament Page Displayed By Student

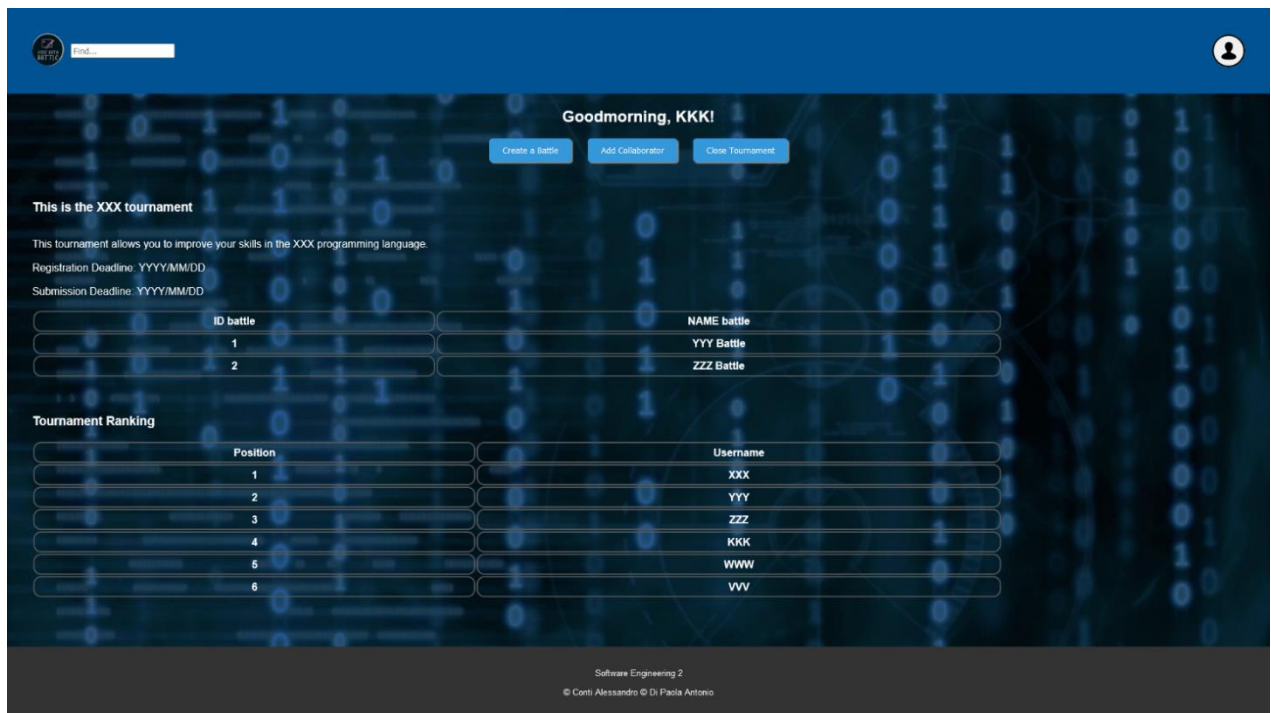


Figure 12 Tournament Page Displayed By Educator

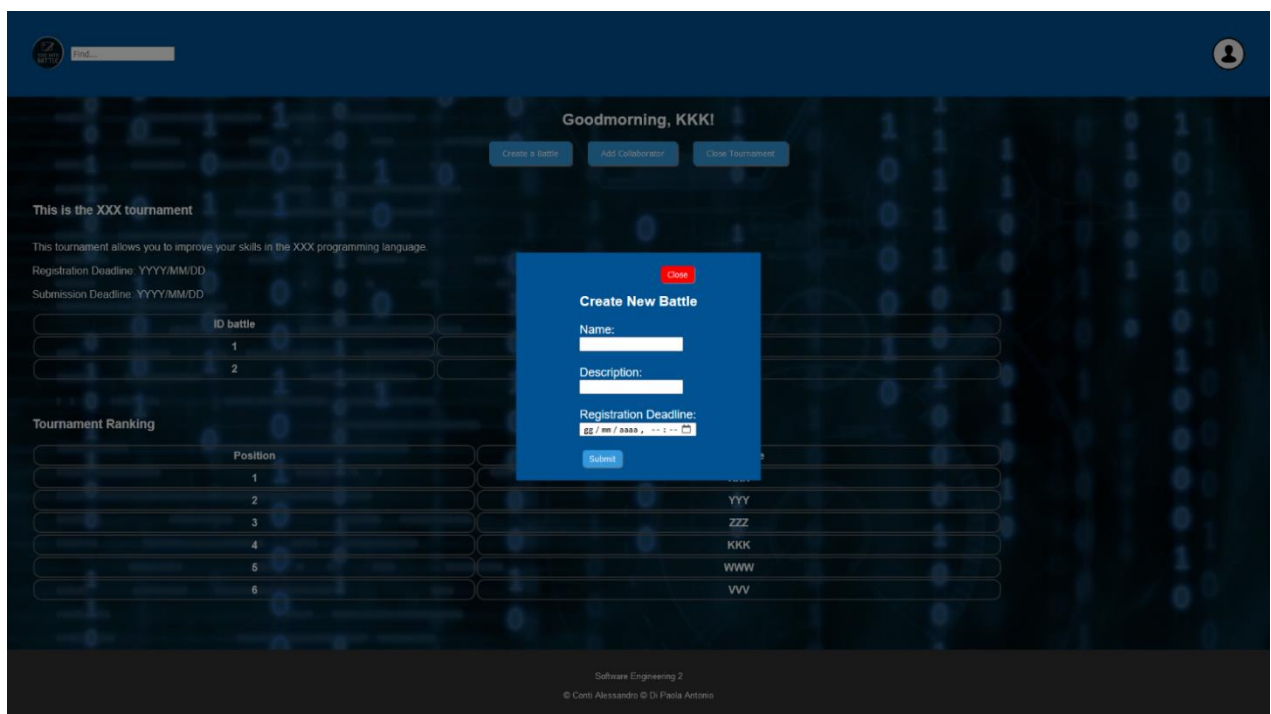


Figure 13 Create New Battle Form

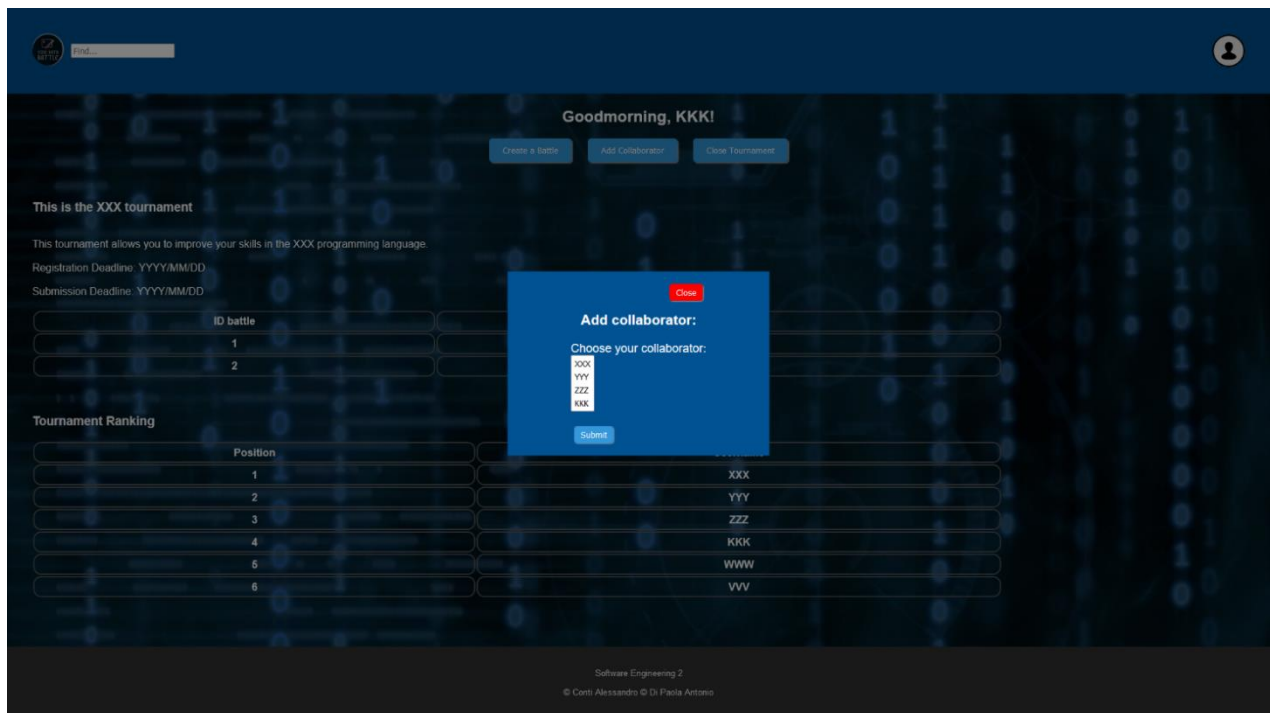


Figure 14 Add Collaborator Form

When a user enters the page of a battle, he will find the details of the battle and the ranking updated in real time.

In case the user is an educator, if the battle has started, he will find a "Modify Grade" button that will allow him to manually modify the grade assigned to a team for their submitted solution.

Conversely, if he is a student, three buttons will be shown in case the battle has not yet started. The first, "Join Battle Alone," if the battle allows, allows the student to participate alone. The second, "Join Battle As A Team," allows the student to invite other students to join to form a team and thus sign up for the battle. The last, "Select Your Team," allows the student to sign up for the battle along with a team that has sent him or her a request to participate.

Below, through Figures 15, 16, 17, 18, 19, the battle page is shown according to the role the user has and the related forms they will need to fill out to do the actions described above.

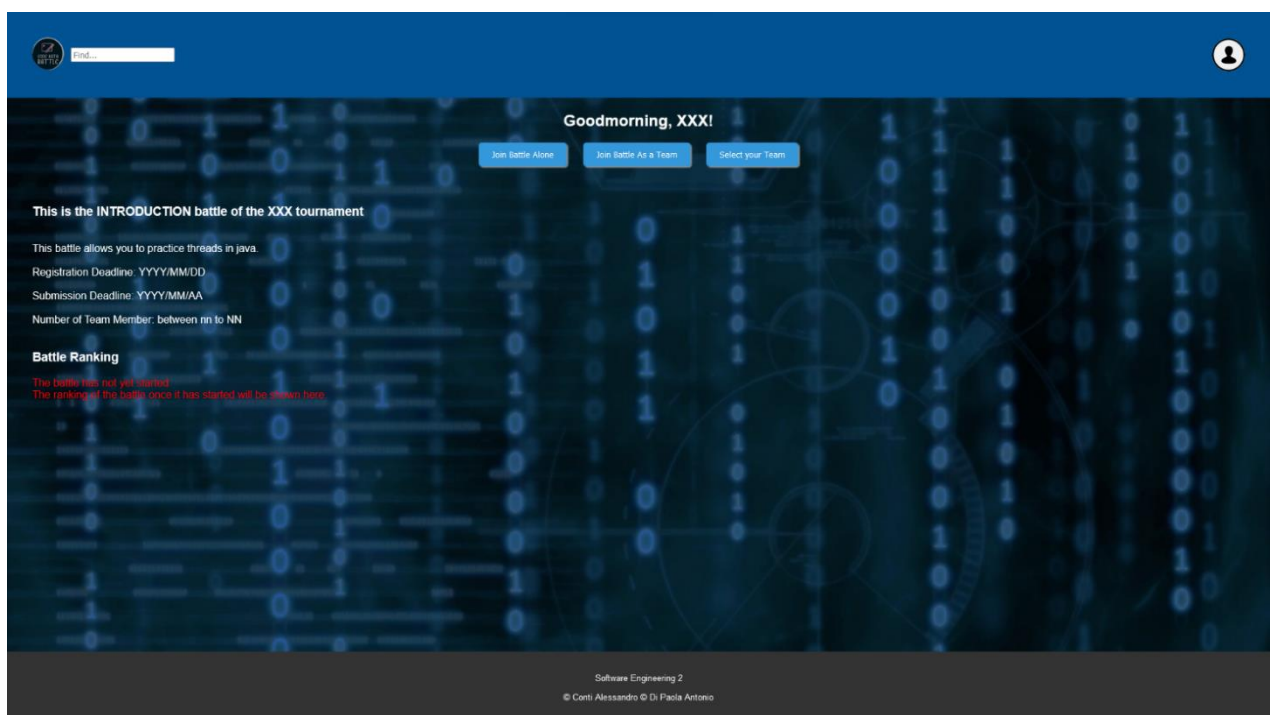


Figure 15 Battle Page Displayed By Student

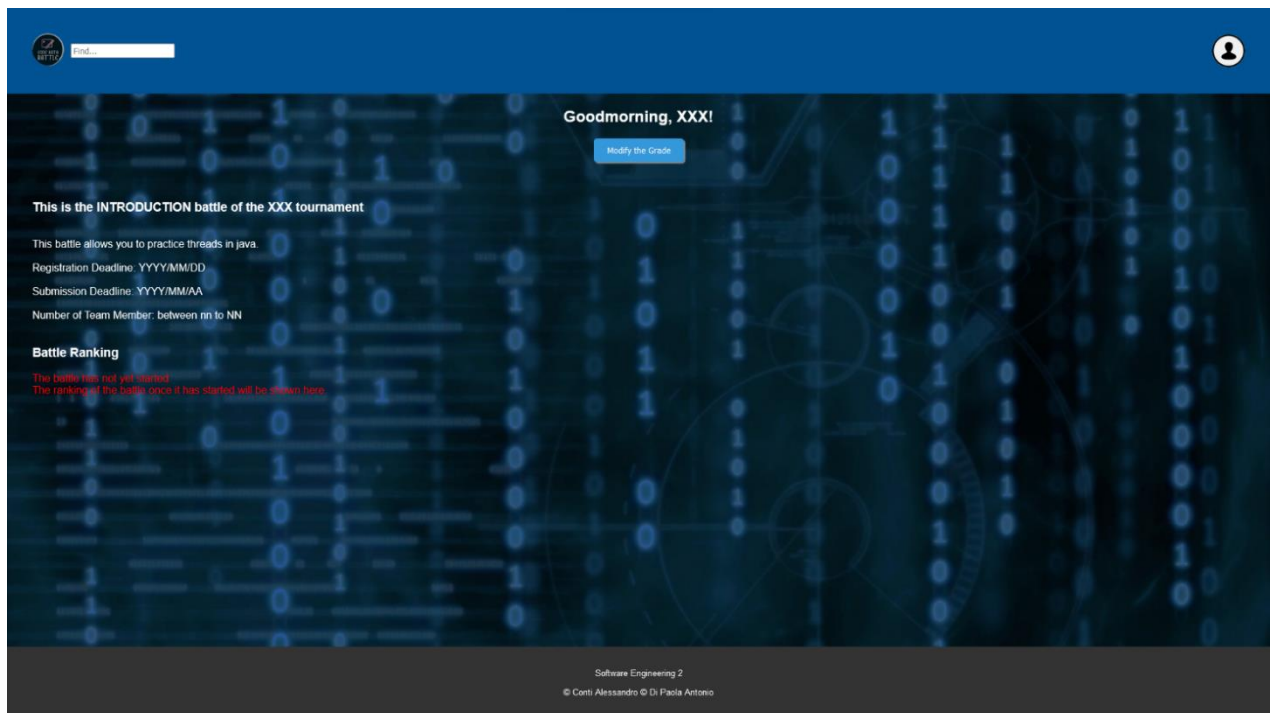


Figure 16 Battle Page Displayed By Educator

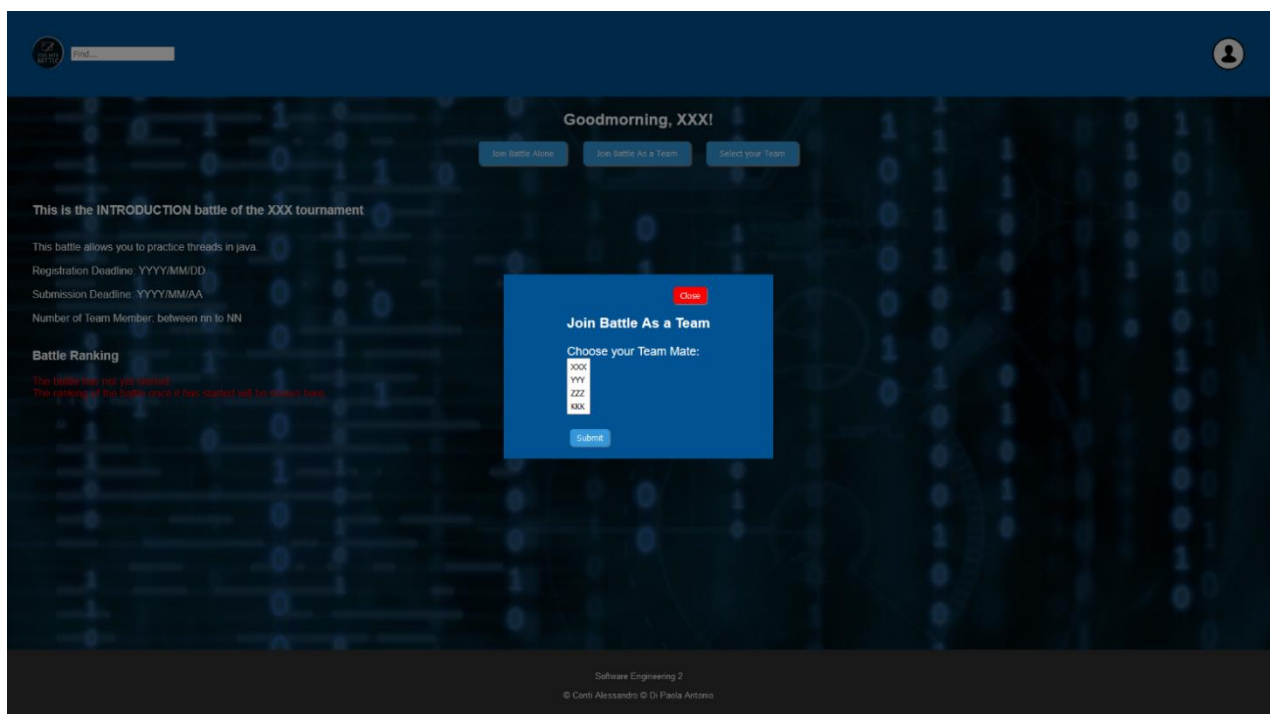


Figure 17 Join Battle As A Team Form

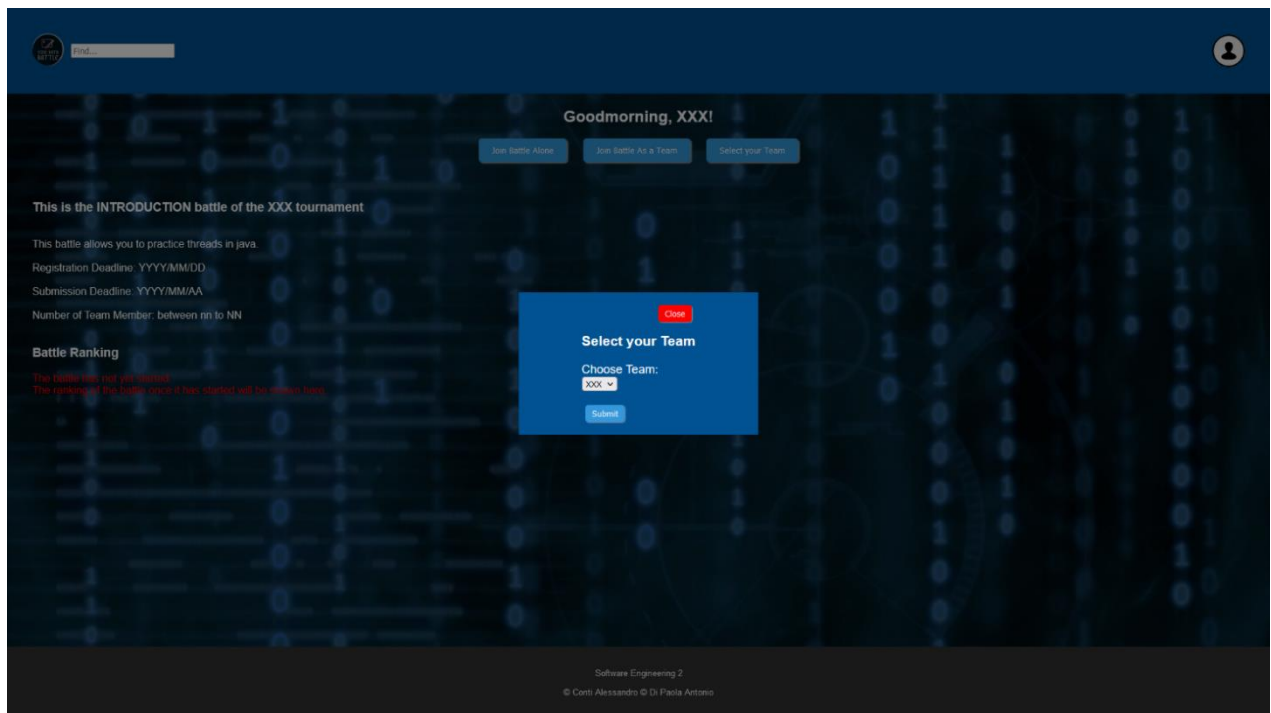


Figure 18 Select Your Team Form

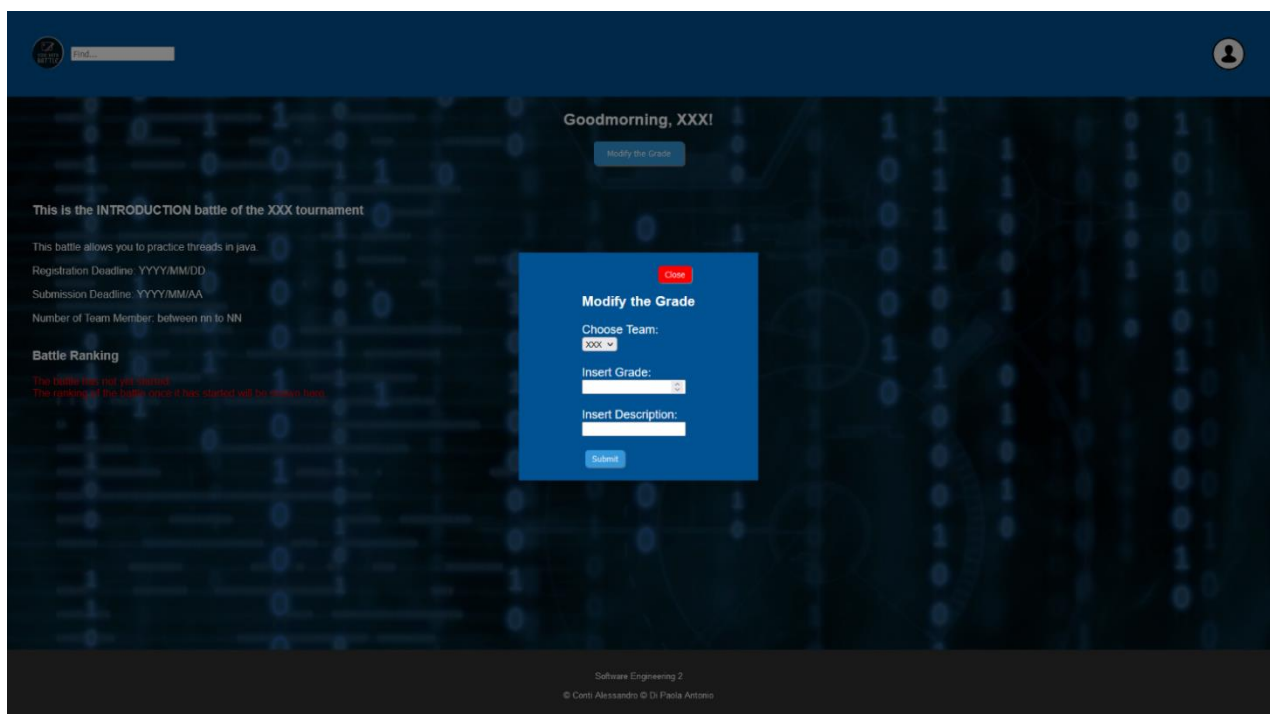


Figure 19 Modify The Grade Form

Should a user wish to view or change their personal information, they will be able to do so by accessing the page dedicated to managing their personal information. To ensure security and confirmation of changes, the user will be required to enter his or her password in order to verify his or her identity.

Below, through Figure 20, the personal information page is shown.

Find...

Goodmorning, XXX!

These are your personal information

Role: EDUCATOR
Educator

Name: XXX

Surname: XXX

Birthdate: YYYY/MM/DD
eg. 01 / 01 / 2000

Username: XXX

Email: XXX@YYY.ZZZ

New Password

Confirm your New Password

If you want to change your personal information enter your current password to confirm.

Password

Submit

Software Engineering 2
© Conti Alessandro © Di Paola Antonio

Figure 20 Personal Information Page

The application also has a page designed to displaying all notifications received by a user, as shown in Figure 21 below.

Goodmorning, XXX!

These are your Notification

Time Notification	Type	Description
YYYY/MM/DD	Join Team	Tournament: XXX Battle: XXX Student: XXX
YYYY/MM/DD	New Tournament	Tournament: XXX
YYYY/MM/DD	New Battle	Tournament: XXX Battle: XXX
YYYY/MM/DD	Manually Evaluation	Tournament: XXX Battle: XXX
YYYY/MM/DD	Close Tournament	Tournament: XXX

Software Engineering 2
© Conti Alessandro © Di Paola Antonio

Figure 21 Notification Page

Each page of the application features, in the upper left corner, the CKB logo, which, when clicked, redirects directly to the user's personal home page. In addition, a search field is available to quickly locate a specific tournament.

While, in the upper right corner, the user's profile photo is displayed. This photo provides quick access to the personal data editing page, the received notifications section, or to log out of the application.

All these elements are clearly illustrated in Figures 22, 23 below.

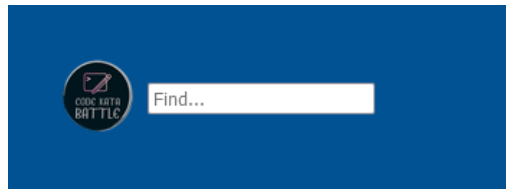


Figure 22 CKB Logo and Search Field

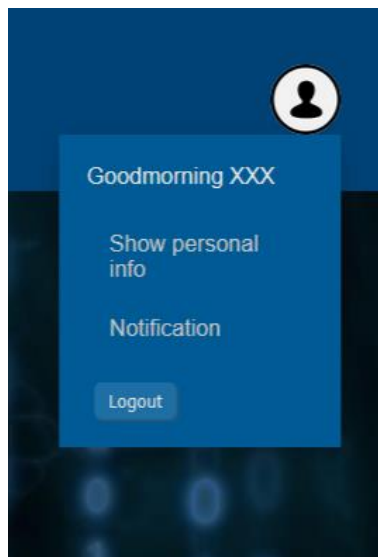


Figure 23 Profile Picture

4. REQUIREMENT TRACEABILITY

In this chapter, we will show the various system components involved in implementing the functional requirements defined in the RASD. The relationship between the functional requirements and the respective components that are going to implement them within the application will be illustrated through a series of tables that will serve as the mapping.

<i>Functional requirements</i>	[R1]. System allows students to sign up. [R2]. System allows educators to sign up
<i>Components</i>	Web Site Web Server Sign In Page Application Server Servlet Sign In Manager DAO UserDAO SendNotification Database

<i>Functional requirements</i>	[R3]. System allows students to login. [R4]. System allows educator to login.
<i>Components</i>	Web Site Web Server Login Page Application Server Servlet LoginInManager DAO UserDAO Database

<i>Functional requirements</i>	[R5]. System allows educator to create a new tournament and to add all the needed information.
<i>Components</i>	Web Site Web Server Tournament Page Application Server Servlet CreateTournament DAO TournamentDAO UserDAO SendNotification Database

<i>Functional requirements</i>	[R6]. System allows educator to grant the permission to other colleagues to create battle in a specific tournament.
<i>Components</i>	Web Site Web Server Tournament Page Application Server Servlet AddCollaborator DAO TournamentDAO Database

<i>Functional requirements</i>	[R7]. System allows educator to create a battle inside a particular tournament by adding CK and the deadlines.
<i>Components</i>	Web Site Web Server Battle Page Application Server Servlet CreateBattle DAO TeamDAO StartBattle SendNotification Database EmailAPI

<i>Functional requirements</i>	[R8]. System allows educator to see rankings of each tournament, which he created or in which he has been nominated collaborator. [R18]. System allows students to see rankings of each tournament, which they are subscribed.
<i>Components</i>	Web Site Web Server Tournament Page Application Server Servlet CheckTournamentRanking DAO TournamentDAO Database

<i>Functional requirements</i>	[R9]. System allows educator to see ranking of a specific battle. [R19]. System allows students to see ranking of a specific battle.
<i>Components</i>	Web Site Web Server Battle Page Application Server Servlet CheckBattleRanking DAO BattleDAO Database

<i>Functional requirements</i>	[R10]. System allows educator to modify the evaluation manually of a specific group in a battle.
<i>Components</i>	Web Site Web Server Battle Page Application Server Servlet ManualEvaluation DAO BattleDAO Database

<i>Functional requirements</i>	[R11]. System allows educator to close a tournament.
<i>Components</i>	Web Site Web Server Tournament Page Application Server Servlet CloseTournament DAO TournamentDAO SendNotification Database Email API

<i>Functional requirements</i>	[R12]. System notifies students about the creation of a new tournament.
<i>Components</i>	Web Site Web Server Tournament Page Application Server Servlet CreateTournament DAO TournamentDAO UserDAO SendNotification Database Email API

<i>Functional requirements</i>	[R13]. System notifies students about the creation of a new battle in a specific tournament in which they have subscribed.
<i>Components</i>	Web Site Web Server Battle Page Application Server Servlet CreateBattle DAO TeamDAO SendNotification Database EmailAPI

<i>Functional requirements</i>	[R14]. System notifies students about publication of final ranking of a specific tournament.
<i>Components</i>	Web Site Web Server Tournament Page Application Server Servlet CheckTournamentRanking DAO TournamentDAO SendNotification Database

<i>Functional requirements</i>	[R15]. System notifies students about publication of final ranking of a specific battle.
<i>Components</i>	Web Site Web Server Battle Page Application Server Servlet CheckBattleRanking DAO BattleDAO SendNotification Database

<i>Functional requirements</i>	[R16]. System allows students to join a tournament.
<i>Components</i>	Web Site Web Server Tournament Page Application Server Servlet JoinTournament DAO TournamentDAO Database

<i>Functional requirements</i>	[R17]. System allows students to join a battle.
<i>Components</i>	Web Site Web Server Battle Page Application Server Servlet JoinBattle JoinTeam DAO TeamDAO SendNotification Database EmailAPI

<i>Functional requirements</i>	[R20]. System, at end of the registration period of a battle, create a GH repo and send invitation to all member of the group.
<i>Components</i>	Application Server DAO TeamDAO BattleDAO StartBattle GH EmailAPI Database

<i>Functional requirements</i>	[R21]. System automatically evaluates the code in the main branch of the repo after each commit in it.
<i>Components</i>	Application Server DAO TeamDAO Evaluation GHA Database

<i>Functional requirements</i>	[R22]. System allows students to search tournament by key words.
<i>Components</i>	Web Site Web Server Tournament Page Battle Page Personal Home Page Application Server Servlet SearchTournament DAO TournamentDAO Database

<i>Functional requirements</i>	[R23]. System allows interaction with GH platform.
<i>Components</i>	Web Server Application Server GHA

5. IMPLEMENTATION, INTEGRATION AND TEST PLANNING

5.1. Overview

This chapter will describe how the system will be implemented, focusing on the development, testing, and integration phases of the individual components that will form it.

The main focus of all testing, whether targeted at individual components or more broadly on the system as a whole, is to identify and correct as many errors as possible in the application before its official release.

During implementation, great importance will be placed on writing the code clearly and documenting it as much as possible with dedicated tools making the code more understandable and accessible. This approach will speed up the process of detecting and correcting errors that emerge during the testing phase, but it also facilitates the integration of new features during the implementation phase.

5.2. Implementation plan

The system will be developed following a combination of bottom-up approach and threaded strategies. This combination of approaches allows each feature to be presented to stakeholders as it is added to the system.

The next section will illustrate and explain the functionalities to be implemented. Each functionality will be developed following a bottom-up approach. We will start with the implementation of the database and all the components that manage the interaction with it. Once this phase is completed, we will proceed with the implementation of the components dedicated to managing the data in the database, which modify it according to the interactions with the user. Finally, the user interfaces that will allow interaction with the application will be implemented.

Adopting this approach allows implementation to be effectively parallelized among all members of the development team, thus contributing to a significant reduction in the overall time required to complete the implementation.

5.2.1. Features identification

The features to be implemented are those described in the RASD in section 2.2 *Product Functions*.

[F1]. Sign up and Login

These functions allow users to sign up and log in to the application.

During the registration process, the application checks, before saving the data, that the username and email address provided are not already in the database. In case they are already in use, the user will be asked to change them to avoid duplication.

Instead during login, the application verifies that the username and password provided are present in the database, if so, it allows the user to log in.

They manage the registration process and user authentication, ensuring that the personal information provided is securely stored in the database using encryption.

It is important to note that this information will not be used for commercial purposes, but only for purposes related to the use of the application.

[F2]. Manage a tournament

This functionality deals with implementing all the components that manage a tournament. As this feature is implemented, components will be developed that will display information about a tournament, allow an educator to create one, add collaborators, and allow students to register.

[F3]. Manage a battle

This functionality deals with implementing all the components that manage a battle. As this functionality is implemented, components will be developed that will display information about a battle, allow an educator to create one, and allow students to sign up as a group, on their own if the battle allows, and accept invitations to participate as a team from other students.

[F4]. GH interaction and evaluation

This functionality deals with implementing all the components that manage interaction with GH. Implementing this functionality will develop the components that will allow automating the assignment of a grade whenever a student commits to the personal battle repository. In addition, the component will be implemented that allows the educator, responsible for the tournament, to change the grade that the student has automatically obtained from the application itself.

[F5]. User notification

This functionality deals with implementing all the components that handle sending notifications to the user via email. Components that implement this functionality will interface with the email API whenever an application component needs to notify the user.

5.3. Unit testing

During the system implementation process, each time a component is implemented, it is tested independently of the other components using stubs and drivers. This approach allows the greatest number of errors to be detected as early as possible.

Stubs simulate the calls that the component will have to make once it is integrated into the system, while drivers simulate calls to functions provided by the component. All of these activities are handled by automated testing tools, which allow the component to be checked for correctness even after changes have been made, thus ensuring early detection of internal errors. If changes are made, the procedure immediately verifies correctness, ensuring an efficient and reliable development process.

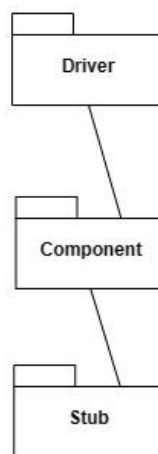


Figure 24 Single Component Testing

5.4. Integration testing

In this section, we will delve into the testing process by focusing on the integration of different components within the system.

Integration testing aims to verify the interaction between groups of components working together to achieve a specific functionality. If that functionality behaves as desired, it is subsequently integrated into the application and subjected to further testing to ensure that its integration with other system functionality occurs without error.

Using this approach, it is possible to verify the correctness and consistency of the interaction between the various functionalities offered by the application. This will help reduce malfunctions in the application, while also ensuring that it is more reliable.

Below, through the figures, we will present the process of integrating the various functionalities within the application, outlining the sequential order that will be followed to implement them.

[IP1]. The first feature to be added to the system is to allow a user to register with the application.

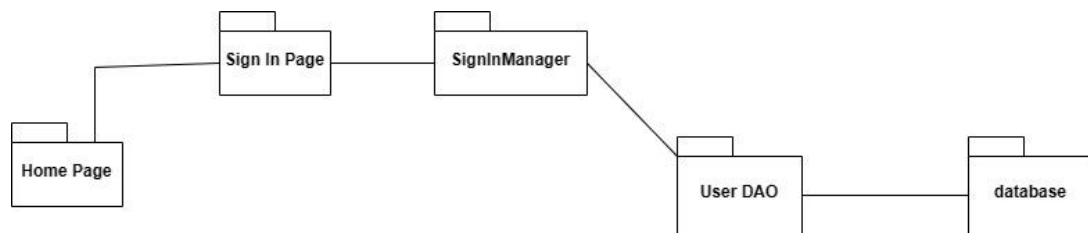


Figure 25 Integration of Sign In Functionality

[IP2]. The second functionality that needs to be added to the system is that which allows the user to log into the application. This functionality must verify the user's existence in the system by checking whether the information provided matches that of a registered user saved in the database.

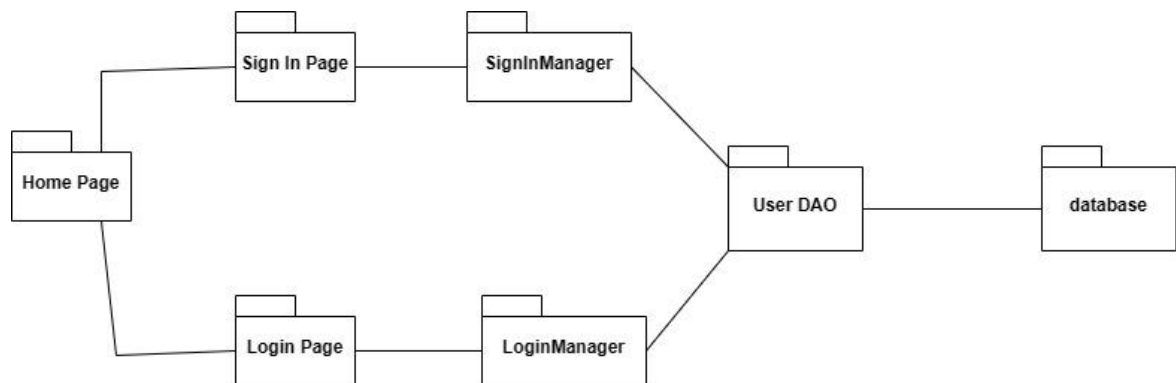


Figure 26 Integration of Login Functionality

[IP3]. At this point, all the components involved in the creation and management of a tournament by an educator, as well as those that allow a student to join in a tournament, can be integrated into the system.

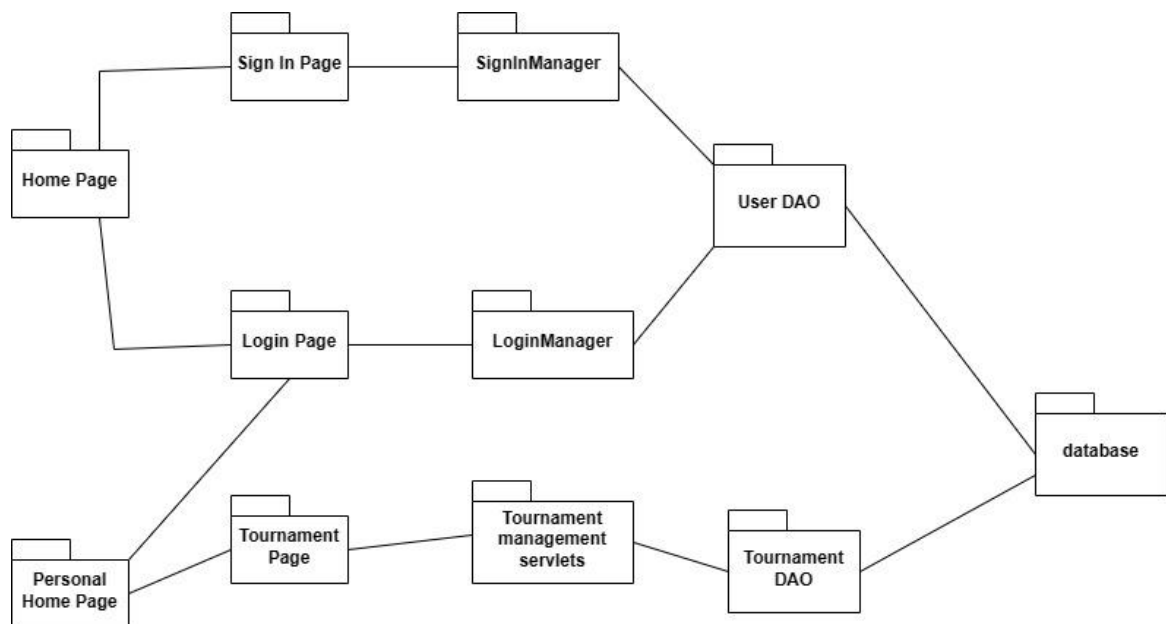


Figure 27 Integration of Manage Tournament Functionality

[IP4]. At this point, all the components involved in the creation and management of a battle by an educator, as well as those that allow a student to join in a battle, can be integrated into the system.

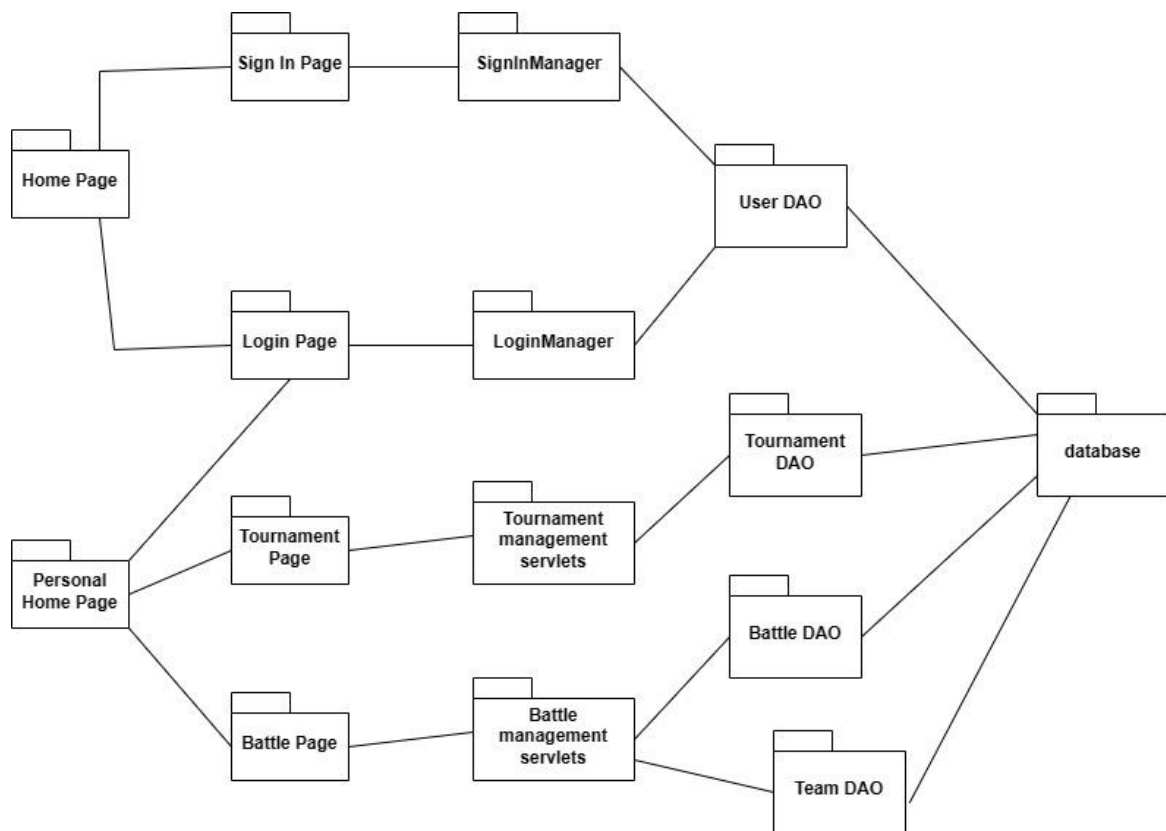


Figure 28 Integration of Manage Battle Functionality

[IP5]. At this time, it is feasible to integrate into the system all components dedicated to evaluating the work done by a student or a group of them. These components assign a grade based on criteria established by the educator during the battle creation phase. Also emerging from this functionality is the ability to display the ranking of tournaments and battles based on the evaluations received from students.

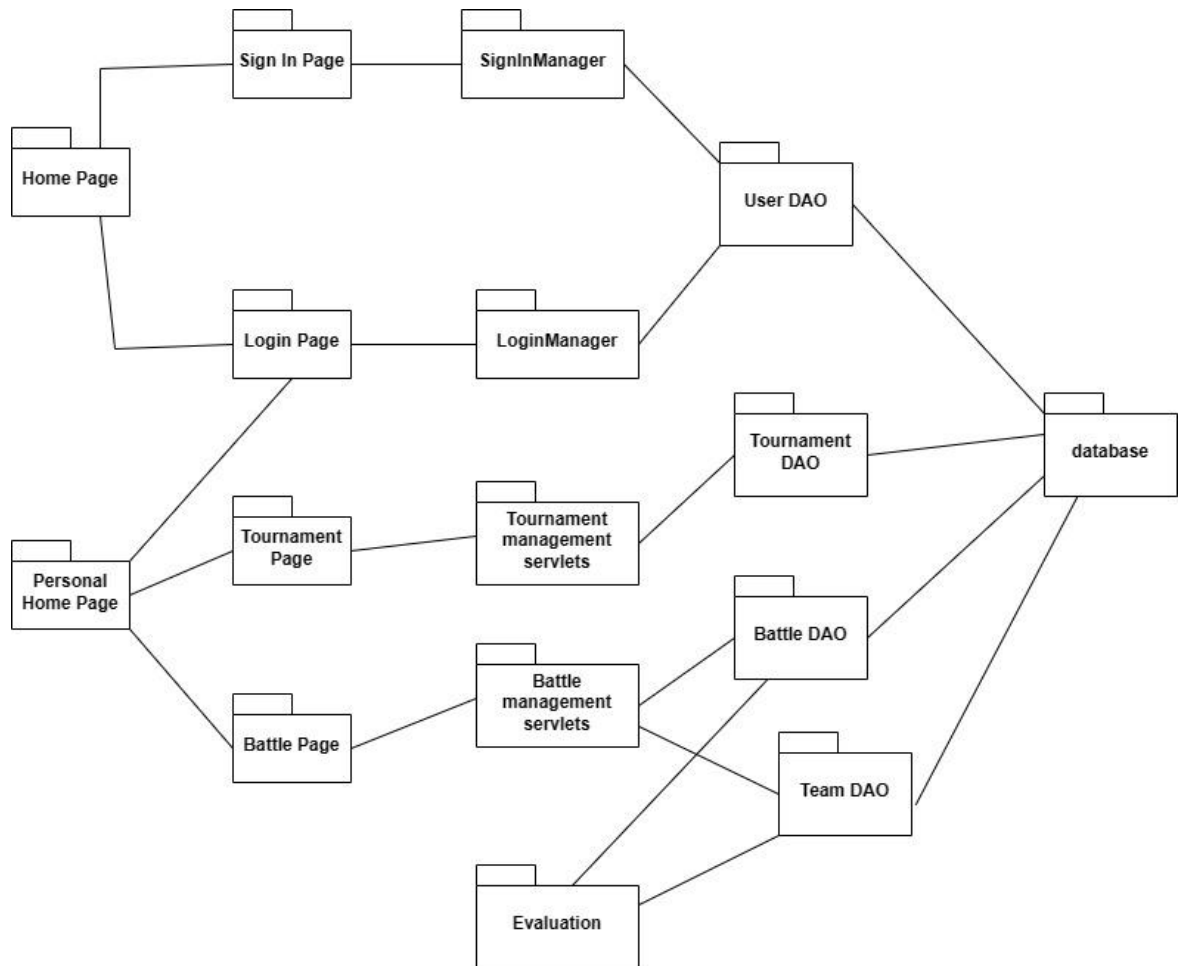


Figure 29 Integration of Evaluation Functionality

[IP6]. The last feature to be integrated into the system is to send targeted notifications to users, differentiating the type of notification to be sent according to specific needs.

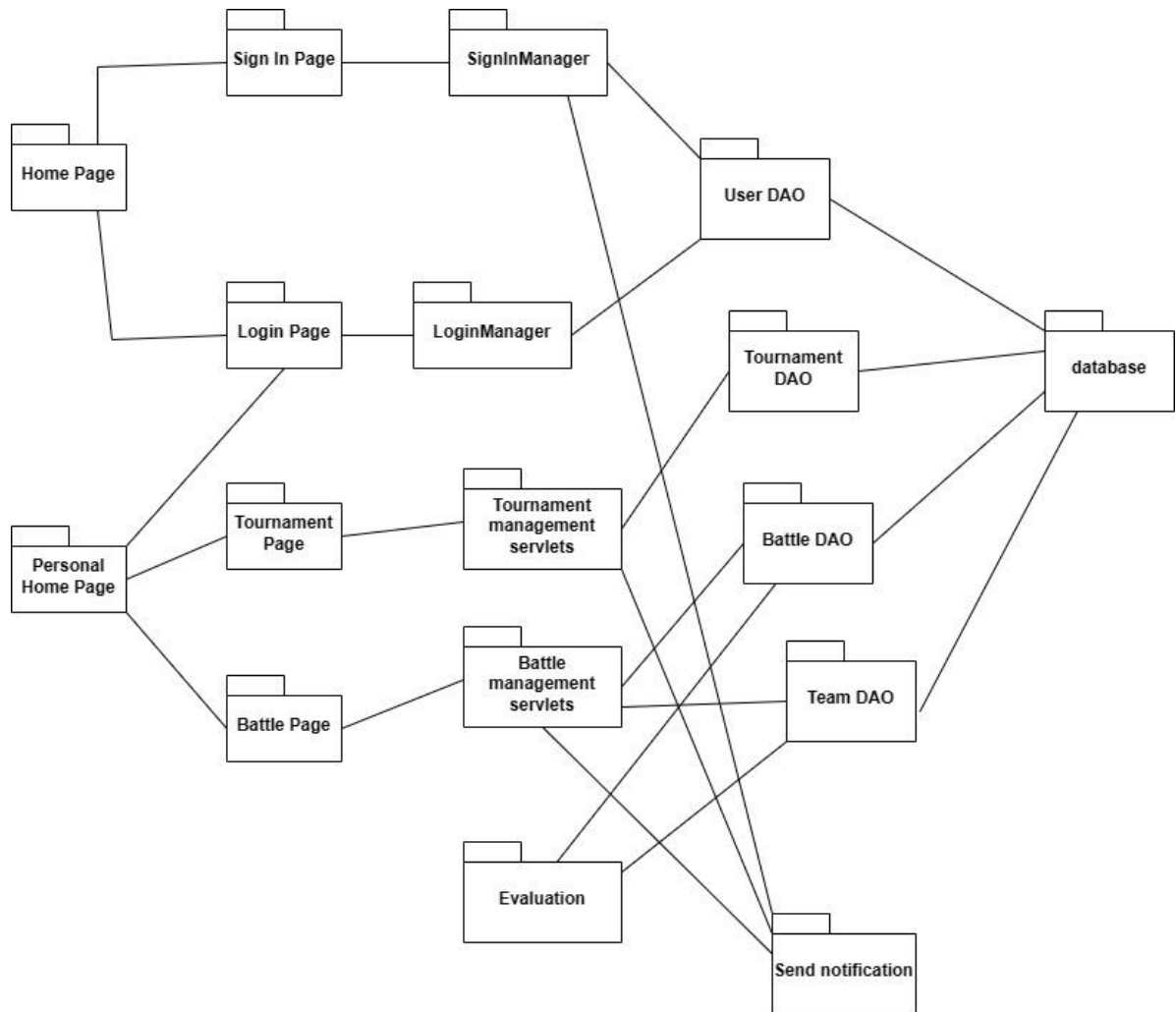


Figure 30 Integration of Send Notification Functionality

5.5. System testing

Once the system has integrated all the required functionalities, it is necessary to subject it to a series of tests to verify its proper functioning and ensure that it meets the performance outlined in the RASD.

These specific tests fall into the following categories:

- **Functional Testing:** These test cases verify that the application meets all the functional requirements defined in the RASD.
- **Performance Testing:** These test cases identify inefficient algorithms and possible bottlenecks that could compromise the overall performance of the application.
- **Load Testing:** These test cases identify memory problems, such as memory leaks or buffer overflows, which may occur during application use.

- *Usability Testing*: These test cases assess the overall usability of the application, trying to understand whether it is intuitive and clear for users. During this phase, a test version can be released to a limited group of users to evaluate its usability.
- *Stress Testing*: These test cases test the system by overloading it or limiting its resources to identify possible vulnerabilities or problems under stress. These tests also examine the security of the application by trying to identify possible vulnerabilities that could be exploited to breach the security of the system.

Performing these tests is crucial to ensure a robust, high-performance and user-friendly application.

6. EFFORT SPENT

In the following tables we will summarize the effort spent by each member of the team on the RASD Document

- Conti Alessandro

<i>Chapter</i>	<i>Effort (in hours)</i>
1	2
2	10
3	10
4	2
5	5

- Di Paola Antonio

<i>Chapter</i>	<i>Effort (in hours)</i>
1	1
2	15
3	1
4	6
5	3

7. REFERENCES

- High Level View, Distributed View, Component View and Integration Functionality made with: *draw.io*
- Sequence Diagrams made with: *SequenceDiagram.org*
- The User Interface overview was written in HTML with: *Visual Studio Code*