Design Document
CodeKataBattles

Federica Maria Laudizi, Antonio Marusic, Sara Massarelli

December 2023

POLITECNICO
MILANO 1863

Contents

List of Tables

List of Figures

1 Introduction

1.1 Purpose

The purpose of this document is to underline and explain in detail the design of the architecture of the CodeKataBattle application. The analysis has been made at different levels to underline different views of the same system. The following are the main levels explored in the document:

- An high level architecture
- The components that make up the system
- The deployment view of the system
- The interfaces provided by components
- The pattern and technologies used in the system
- The User Interfaces provided by the system

Moreover, this document describes the design characteristics required for the implementation, as well as an overview of the implementation, integration and testing plans.

1.2 Scope

Here we give a quick summary of the application's purpose. For a more thorough explanation you may refer to the RASD.

CodeKataBattle (CKB) is a platform designed to enhance students software development proficiency. Educators leverage the platform to engage students in challenging code kata battles (a programming exercise in a programming language of choice), fostering competition among teams and providing a means for students to demonstrate and enhance their skills. Battles are organized in tournaments and, at the end of each, educators can manually adjust the scores of the participants on top of the automatically assigned score.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Kata battle**: a coding exercise where a task is given to the participants and they have to submit their solution through GitHub.
- **Tournament**: a competition made of a series of Kata battles , each contributing to a final score. The score of these battles are then used to establish a leaderboard that ranks the participants.
- **Badge**: A badge is a collectible award earned when specific predefined conditions are met. The admins define a boolean formula and a value X that represents the number of participants eligible to receive the badge. It could be, for instance, the first 10 participants who complete a certain challenge or meet a particular criterion.

1.3.2 Acronyms

- **UI**: User interface through users require available functionalities.
- **UX**: User Expeirence, way in which the user interacts with the service.

1.3.3 Abbreviations

- **CKB**: CodeKataBattle, the platform itself.
- **G.i**: i-th goal.
- **R.i**: i-th requirement.

1.4 Revision History

1.5 Reference Documents

- The specification of the RASD and DD assignment of the Software Engineering II course, held by professor Matteo Rossi, Elisabetta Di Nitto and Matteo Camilli at the Politecnico di Milano, A.Y 2023/2024;
- Slides of Software Engineering 2 course on WeBeep;
- DD Sample from A.Y. 2022-2023

1.6 Document Structure

The other sections of the Design Document (DD) are organised in this way:

Chapter 1 *Introduction*: presents the scope, purpose, and structure of the document and provides the definitions, acronyms and abbreviations used in it.

Chapter 2 *Architectural Design*: presents an in-depth description of the System's architecture. It defines the main components, the relationship between them and the deployment of components. There are different views and levels of analysis of the components plus some subsections useful for identifying how the components interact and for the architectural styles and patterns.

Chapter 3 *User Interface Design*: it's a complementary section of what was included in the RASD. It includes the definition of the UX processes through a model that represents the flows between interfaces.

Chapter 4 *Requirements Traceability*: a complementary section of what was included in the RASD. It contains all the requirements and shows the relationship between them and design choices done in order to satisfy them.

Chapter 5 *Implementation, Integration and Test Plan*: it shows the order of the implementation and integration of all the components and subcomponents, explaining how the application will be tested.

Chapter 6 *Effort Spent*: it's a section that contains a table for identifying the hours and the effort spent by the team to deliver the Design Document.

Chapter 7 *References*:in this last chapter are mentioned the references to documents or sites about technologies, patterns and architectures used in this document.

2 Architectural Design

2.1 Overview: High level components and their interaction

This section is intended to explain in detail all the components used by the system in order to offer all the functionalities. The figure below shows the high-level architecture of the system-to-be. All the components are better explained in the following paragraph.

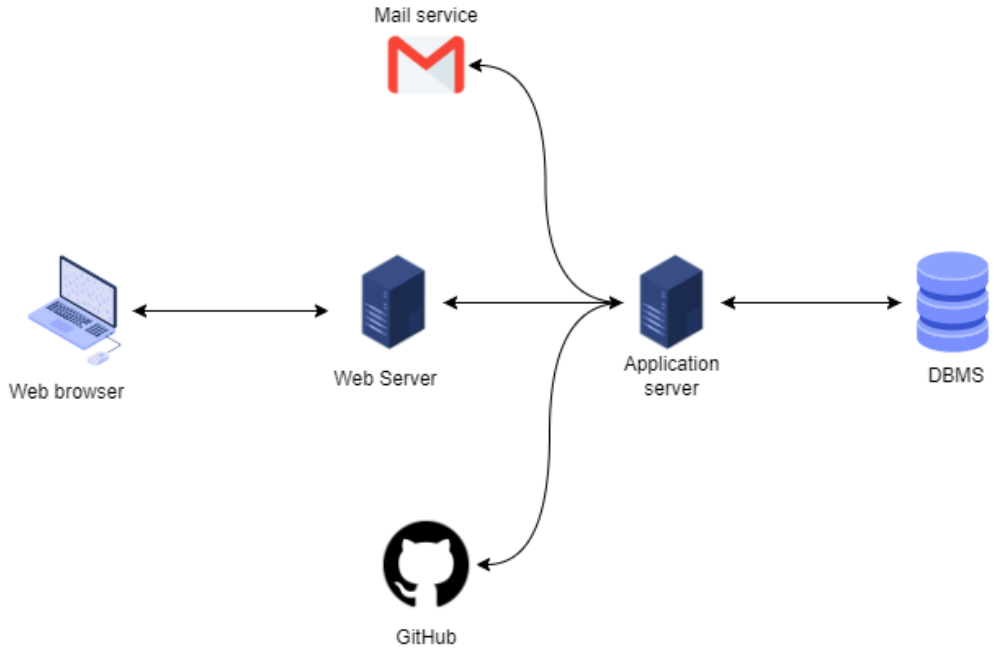


Figure 1: System overview graph.

The user interacts with the web application, sending requests to the web server which serves static content and forwards dynamic requests to the application server. The application server processes the request, interacts with the database for data retrieval or modification, and communicates with the GitHub API for version control and repository management.
The Email service is utilized for sending notifications and alerts to users based on specific events or triggers within the application. We decided to not have a dedicated SMTP server, but an off the shelf solution like MailChimp.
Basically, the application is structured with a three-tier application where:

- The UI (**Presentation layer**) formats the information to be shown, its built with HTML5, cascading style sheets (CSS) and JavaScript, is deployed to a computing device through a web browser or a web-based application.
- The **Logic tier** (Web & Application Server), contains the business logic that supports the application's core functions and interacts with the database.
- The **Data tier** consists of a database and a program for managing read and write access to a database.

Finally, the application has to follow the Client - Server architecture. Client and server are going to communicate with each other through a working internet connection. Moreover, the architecture will be event driven, meaning that the system that receives events has to react to them producing certain outputs. In particular, the interaction between client and server will work as follows:

1. The user clicks on interactive components of the UI generating an event.
2. Once the View catches the event, it is turned into a method call to the server.
3. The server analyzes the request made by the user and eventually performs some operations that could require access to the database.
4. Then, operations' output will be sent back to the user.

2.2 Component View

Illustrated in *Figure 2*, the diagram presents the components of the system. Components shaded in yellow represent the application server and the blue one is the WebServer both constituting elements within the logic layer. In contrast, those highlighted in red signify components directly accessible to users, forming the Presentation tier. The solitary green-colored element represents the Database Management System (DBMS), exclusive to the data tier.

Additionally, components such as Github and Mail are rendered in purple to underscore their third-party nature, distinctly separate from the Code Kata Battle system.

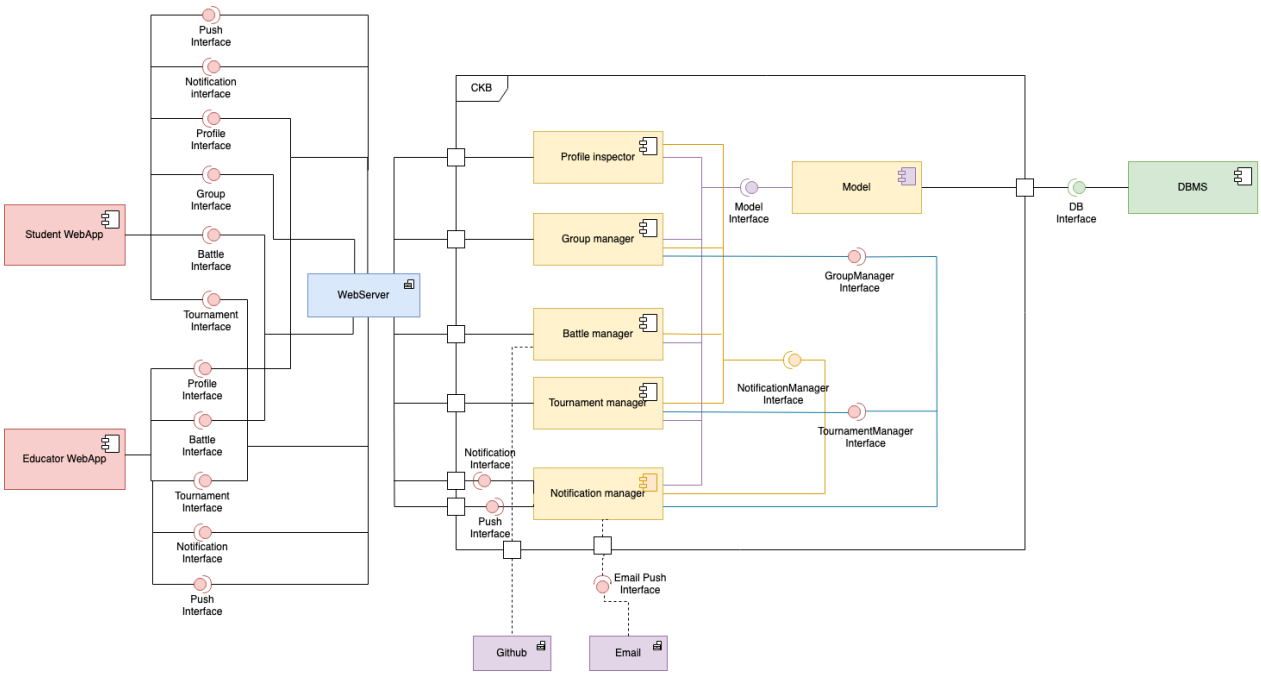


Figure 2: Component view of the system.

Let's now analyze the single components in detail:

- The **WebApp** is the web interface used by users to access the system. Basically it communicates with the system via HTTP requests to the WebServer, for this reason it can simply be executed on a web browser.

- The **WebServer** component is designed to satisfy all the requests from the CKB web application accessed by Users devices' browser. It sends on the requests to the application Server which, after forwarding them to the competent components, returns the answer web pages that are finally forwarded to the user by the web server itself.
- The **Model** component is used to store the data that are part of the persistent state of the system (which in our case is stored on a database) when this is loaded into the application. Moreover, it provides all the methods that are necessary to access those data and it also implements the logic to manipulate them. In other words: whenever some data contained in the database is needed by any component of the system, for any kind of computation, they delegate the task of retrieving such information to the model component. It not only retrieves them but it also stores them in ad-hoc data structures so that they are ready to be used for the required computations. The model component is the only one interacting with the DBMS service.
- The **Profile Inspector** component has the role of granting the privacy of the customers' data by denying the access to unauthorized users, in addition it handles new registrations from guests. If the credentials that users provide are correct, based on the type of user (educator or student) it will return the educator or student home page.
- The **Group Manager** component serves as the interface for user interaction when joining or creating a group. Specifically, when a user intends to join a group, it engages with the model to check the group's status (public or private) in the DBMS, effectively handling both scenarios. Similarly, when a user aims to create a new group, the Group Manager communicates with the model to update the DBMS with the relevant group information. In either case, if applicable, the Group Manager can also liaise with the notifications manager to inform other students within the system.
- The **Battle Manager** serves its designated role by overseeing all tasks associated, as implied by its name, with a battle. This component interacts with the Model to perform various tasks such as conducting tests and assigning scores to submissions. Additionally, it communicates with the Notification Manager to inform participants once the battle concludes and the final rankings are available.
- The **Notification Manager** is the component that is in charge of handling the notifications generated by the system. Every time the system has to send a notification to the user's web application this component manages those notifications. This component also receives the actions from the students, such as accept or reject notifications. Whenever a user opens a notification from the email a link redirects the user to his/her notification page on the web application where this component will detect if the notification is accepted or declined and in general handle it.
- The **Tournament Manager** is the component designated, as the name suggests, to manage all the tasks related to a tournament. In particular, its going to interact with the Model to access the database for data retrieval and it will communicate with the Notification manager to notify users when the tournament has been created or advise participants that it has finished.
- The **DBMS** contains all the relevant data to the application and it resolves the queries received from the model component with the requested data.

2.3 Deployment View

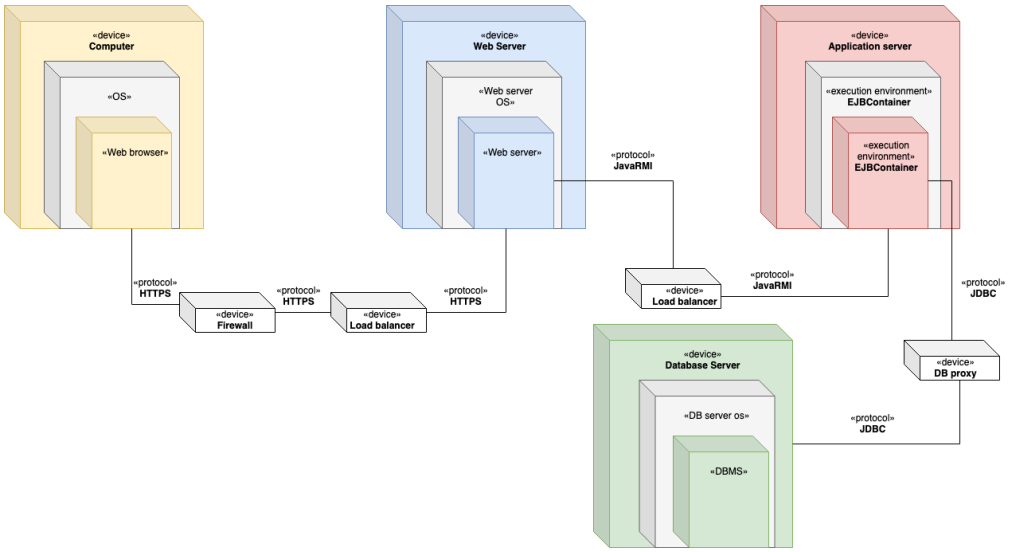


Figure 3: Deployment diagram

Further details about the elements in the graph are provided in the following.

- **Computer**: It is a normal computer used both by students and educators. It does not require any special application except for a web browser that allows users to connect to the platform, GitHub, and their personal email. The system establishes a secure connection with the web server via an HTTPS request. This connection passes through a firewall and a load balancer to ensure secure and balanced access to the platform.
- **Web server**: This server handles requests for static content like HTML, CSS, and images, as well as dynamic content generated by the CodeKataBattle application server. Requests for static resources are served directly, while dynamic requests are forwarded to the application server for processing. The server plays a crucial role in delivering a seamless experience for users participating in coding battles, managing both static and dynamic content to create an interactive and engaging platform.
- **Application server**: This server handles dynamic content generation and business logic for the CodeKataBattle web application. It processes requests from the web server and interacts with databases to provide personalized and real-time information to users participating in coding battles.
  The application server is responsible for managing user sessions, evaluating code submissions, updating leaderboards, and maintaining the overall state of the CodeKataBattle platform.
- **Database server**: This DBMS stores and manages the data crucial for the CodeKataBattle web application. It handles the persistence of user profiles, coding challenge details, submissions, and leaderboard information.
  The DBMS ensures data integrity, supports efficient queries for dynamic content generation, and plays a vital role in maintaining the consistency of user data across the different replicas of the DB.
- **Firewall**: This security component ensures the protection of the CodeKataBattle web application by monitoring and controlling incoming and outgoing network traffic. It enforces security policies to prevent unauthorized access, safeguarding the application against potential threats and vulnerabilities. The firewall acts as a barrier between the CodeKataBattle platform and the external network, enhancing the overall security posture of the system.

- **Load Balancer**: This component optimizes the distribution of incoming network traffic across multiple servers hosting the CodeKataBattle web application. We have a first load balancer that distributes the incoming connections across multiple web servers, then we have a second load balancer that distributes the connections amongst different replicas of the application server. Having load balancers improves performance, ensures high availability, and prevents any single server from being overloaded. The load balancer enhances the scalability and reliability of the CodeKataBattle platform by efficiently managing the workload and directing user requests to available server resources, contributing to a responsive and resilient user experience.

- **Database proxy**: This component acts as a middle layer between the application servers and the database replicas that intercepts incoming database requests and uses algorithms to determine how to distribute incoming connections among the available replicas. One common algorithm is Round Robin, that distributes connections in a circular sequence. In addition to load balancing, our database proxy will handle fail-over scenarios. If one replica becomes unavailable, the proxy can automatically redirect connections to a healthy replica. A possible example of database proxy for SQL databases is ProxySQL.

In the deployment view, the components identified in the component view are mapped onto the physical infrastructure. The web server component in the component view corresponds to the web server in the deployment view.

On the other hand, all the other components identified in the component view, including the Profile Inspector, Group Manager, Battle Manager, Notification Manager, Tournament Manager, and the Model, are collectively mapped onto the application server in the deployment view. The application server is the core processing unit that manages dynamic content generation, business logic, and interactions with the database server.

The Database Management System (DBMS) identified in the component view is separately mapped to the database server in the deployment view. The DBMS is responsible for storing and managing crucial data for the CodeKataBattle web application. It ensures data integrity, supports efficient queries, and maintains the consistency of user data.

Furthermore, to enhance security and performance, components such as the firewall, load balancer, and database proxy are introduced in the deployment view. The firewall provides security by monitoring and controlling network traffic, the load balancer optimizes the distribution of incoming traffic across multiple servers, and the database proxy acts as a middle layer between the application servers and the database replicas, handling load balancing and fail-over scenarios.

2.4 Runtime Views

Here, we further develop the use cases analysed in the RASD document, showing the way those are accomplished using the various components defined in section 2.2.

- **Enter a page**: Here we show how the system reacts when an user loads a page. The process is similar for all pages and for both types of user, so we show only a case and we assume that it will be clear the behaviour for all pages of the website. Moreover, in the following sequence diagrams we assume this procedure to be implicit. In this scenario a student want to enter his personal page that comprises notifications, tournament to which the student participated, badges collected and some statistics relative to past tournaments and battles (this last section is not covered in the sequence diagram for brevity). When entering the page the Web App posts an HTTP GET request for each portion of data that needs to load the page. Each request will be handled by the proper Manager, the manager will perform some security checks (in this case will ensure that the user is logged in using the session, we omitted this check to keep the diagram brief) and it will query the DBMS for the necessary data. If the security check fails the Manager will respond with an error message and the request will not be fulfilled.

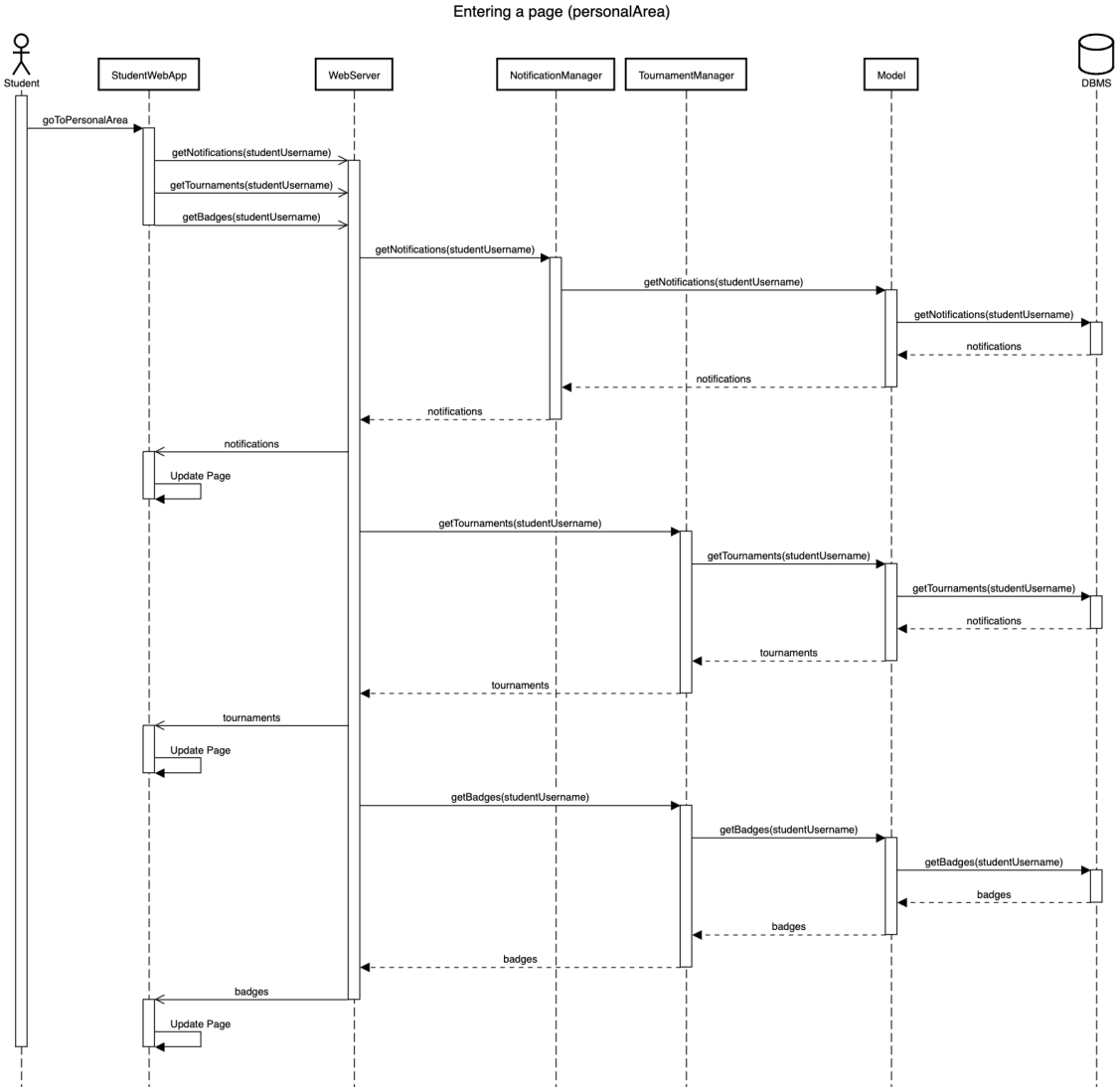


Figure 4: Enter a page

- **Log in**: A user initiates the login process by accessing the website and entering their credentials into the login form. The WebApp captures these credentials and forwards them to the ProfileInspector for authentication. The ProfileInspector consults the Model and the DBMS, to verify the correctness of the credentials. Upon successful authentication, the ProfileInspector informs the WebApp, which redirects the user to the homepage. If authentication fails, an error message is displayed to the user.

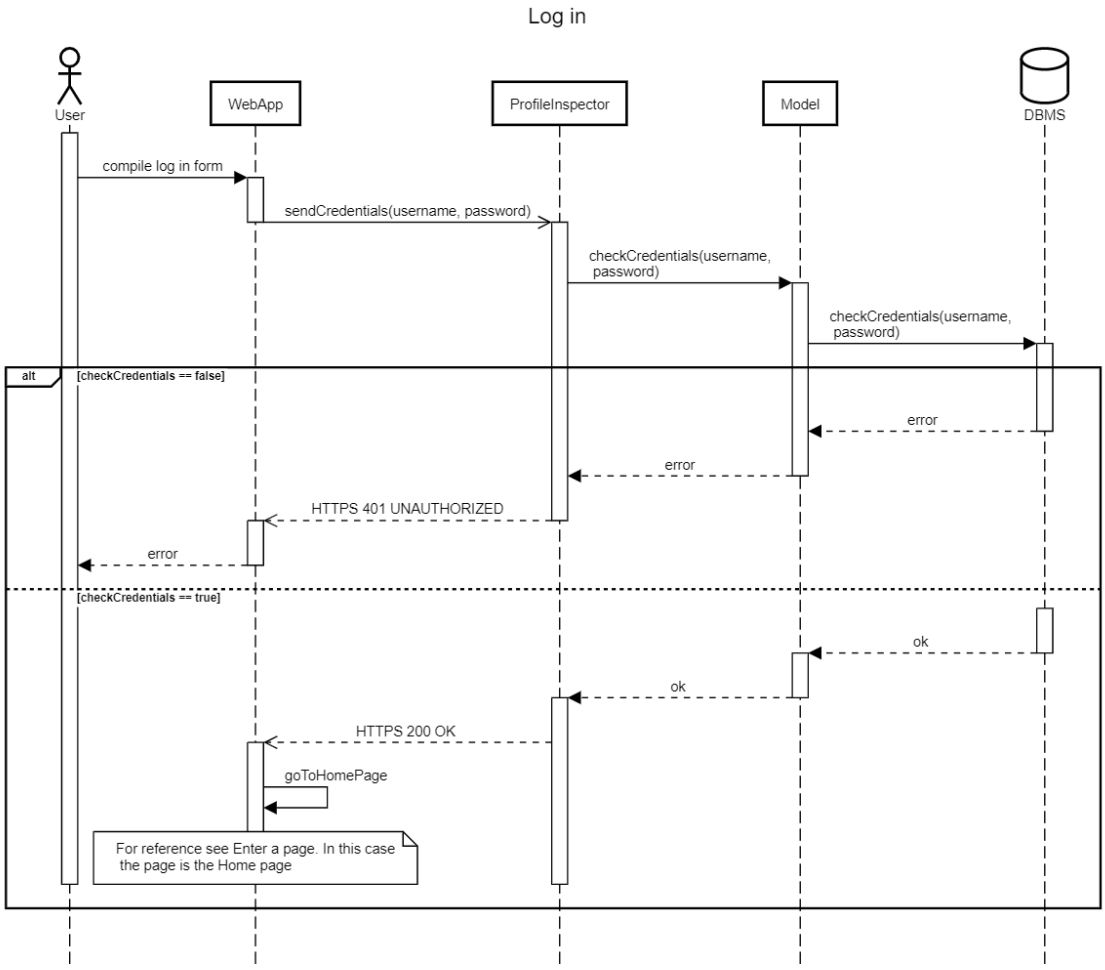


Figure 5: Log in

- **Sign up**: A user initiates the registration process by filling out a sign-up form with their personal information. The WebApp forwards this information to the ProfileInspector for validation. The ProfileInspector checks the user's information against the Model to ensure that it is complete and unique. Once validated, the ProfileInspector creates a new user account in the DBMS and redirects the user to the homepage.

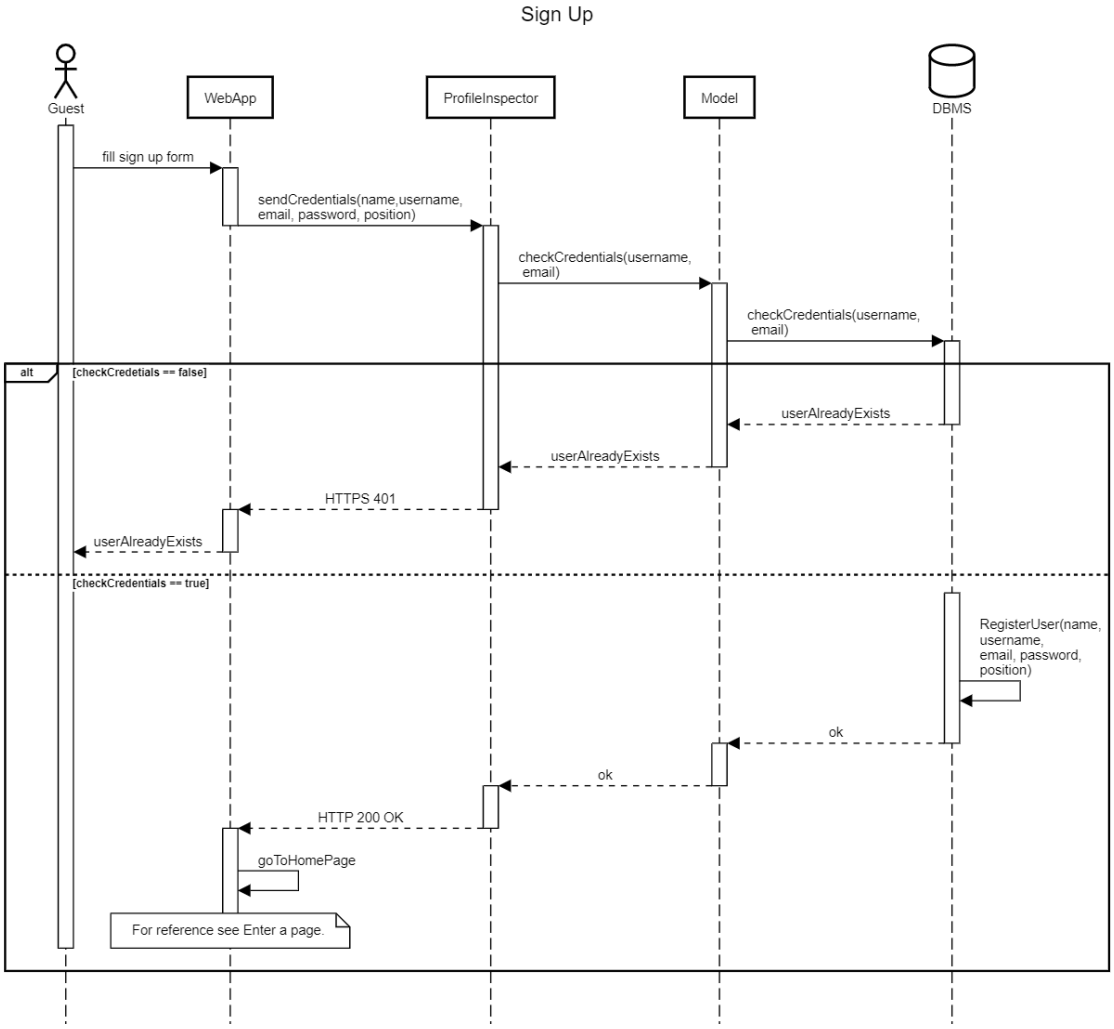


Figure 6: Sign up

- **Creation of a tournament**: An educator initiates the tournament creation process by entering the tournament information into the EducatorWebApp. The EducatorWebApp forwards this information to the TournamentManager, which creates a new tournament record in the Model. The TournamentManager then registers the tournament with the DBMS and triggers the NotificationManager to notify all users. The Model updates the state of the tournament to "Created" and sends a notification to the EducatorWebApp. The EducatorWebApp then informs the educator that the tournament has been created.

  For the clearness of the diagram the process of adding an administrator is not described here but in Figure 14.

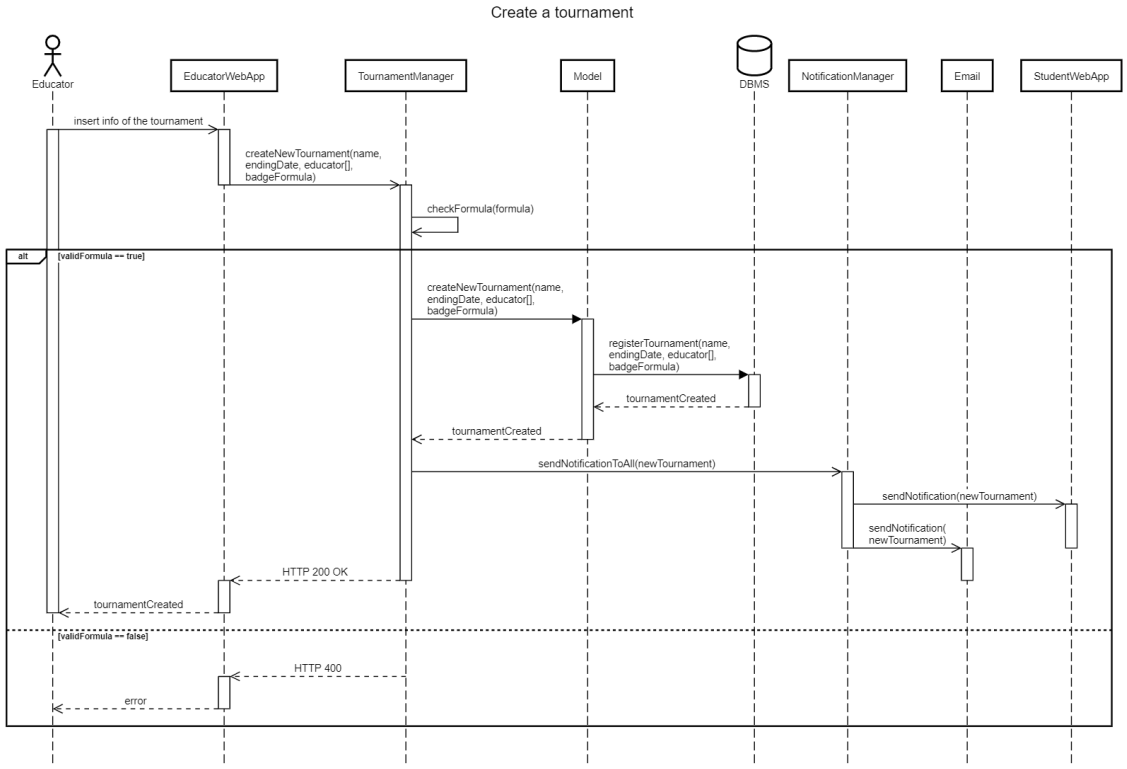


Figure 7: Creation of a tournament

- **Creation of a battle**: an educator sends a request to the EducatorWebApp to create a new battle. The EducatorWebApp invokes the createNewBattle() method on the BattleManager. The BattleManager creates a new battle record in the Model and registers the battle with the DBMS. When the battle manager realizes the battle has been created triggers github to create a new repository and the notificationManager to notify all students subscribed in the tournament in which the battle was created.

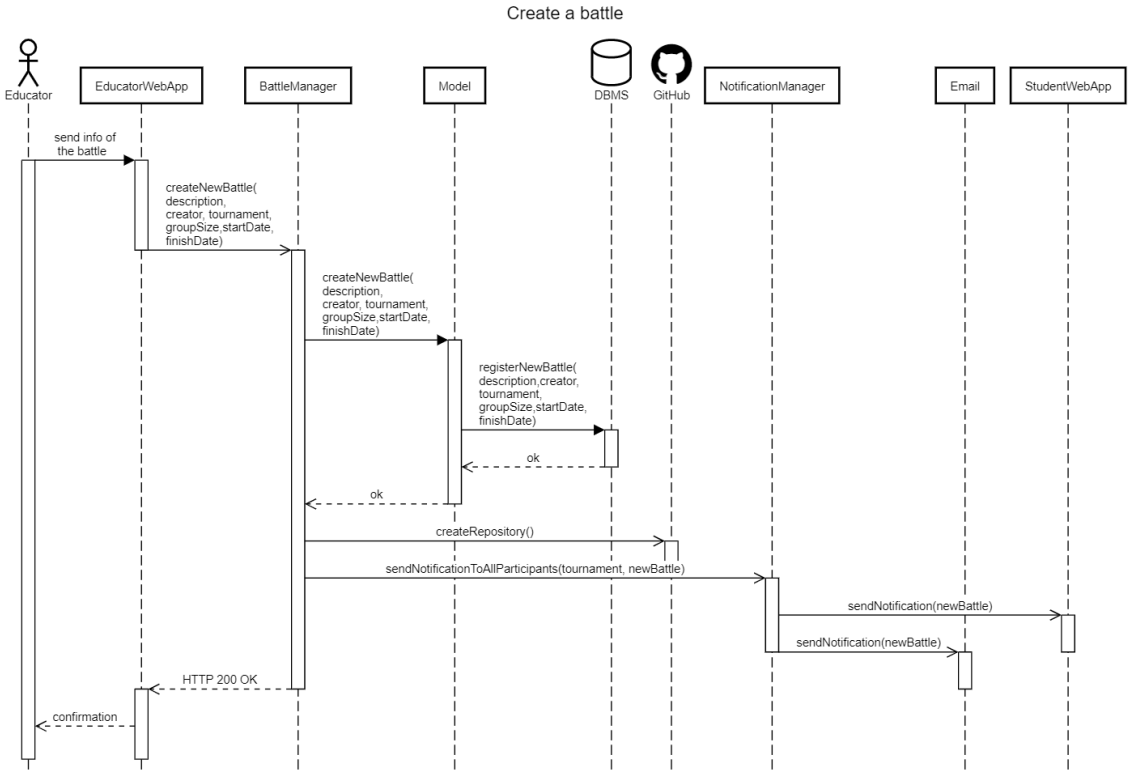


Figure 8: Creation of a battle

- **Join tournament** : The student receives the notification to join a tournament and when he accepts it the NotificationManager sends a notification to the TournamentManager, which updates the student's status in the Model and sends an acknowledgment to the TournamentManager. The TournamentManager then sends a confirmation to the StudentWebApp, which then informs the student that they have successfully joined the tournament.

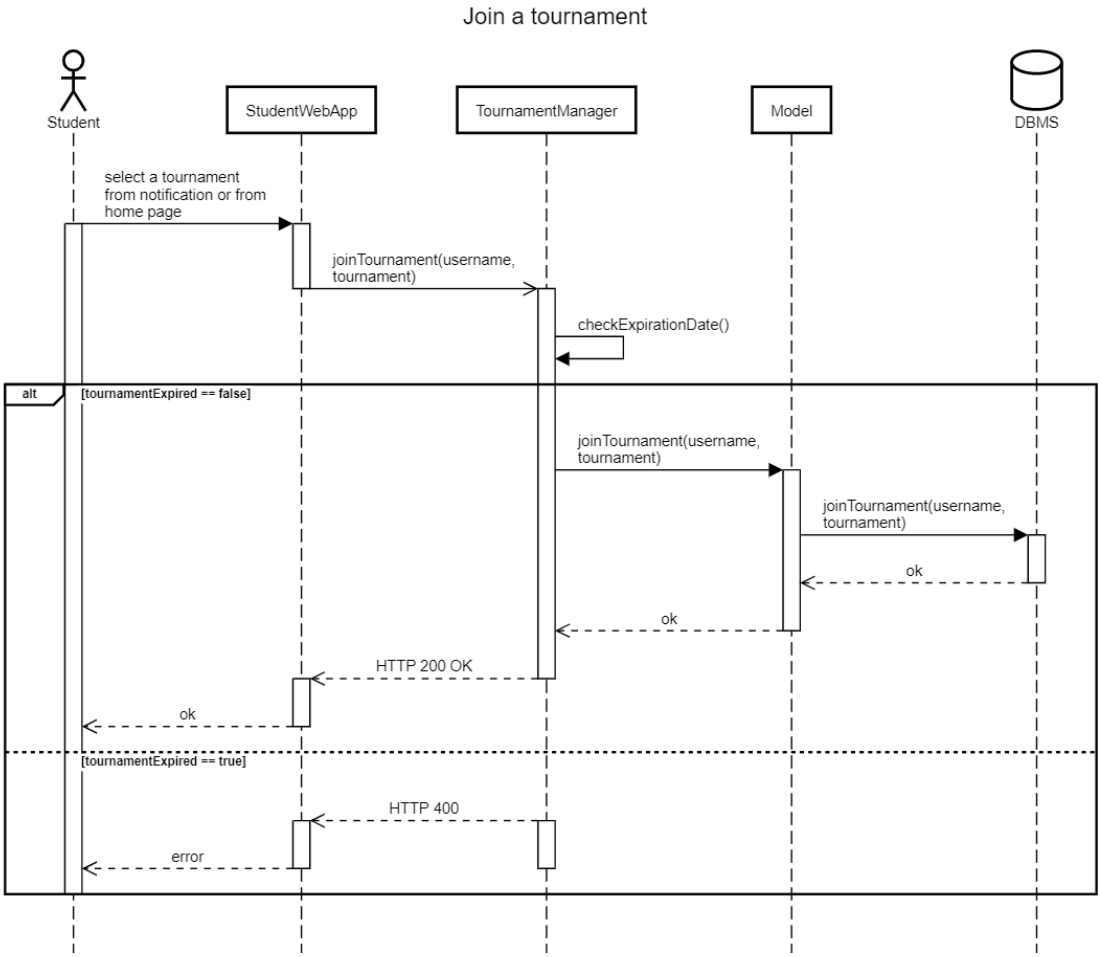
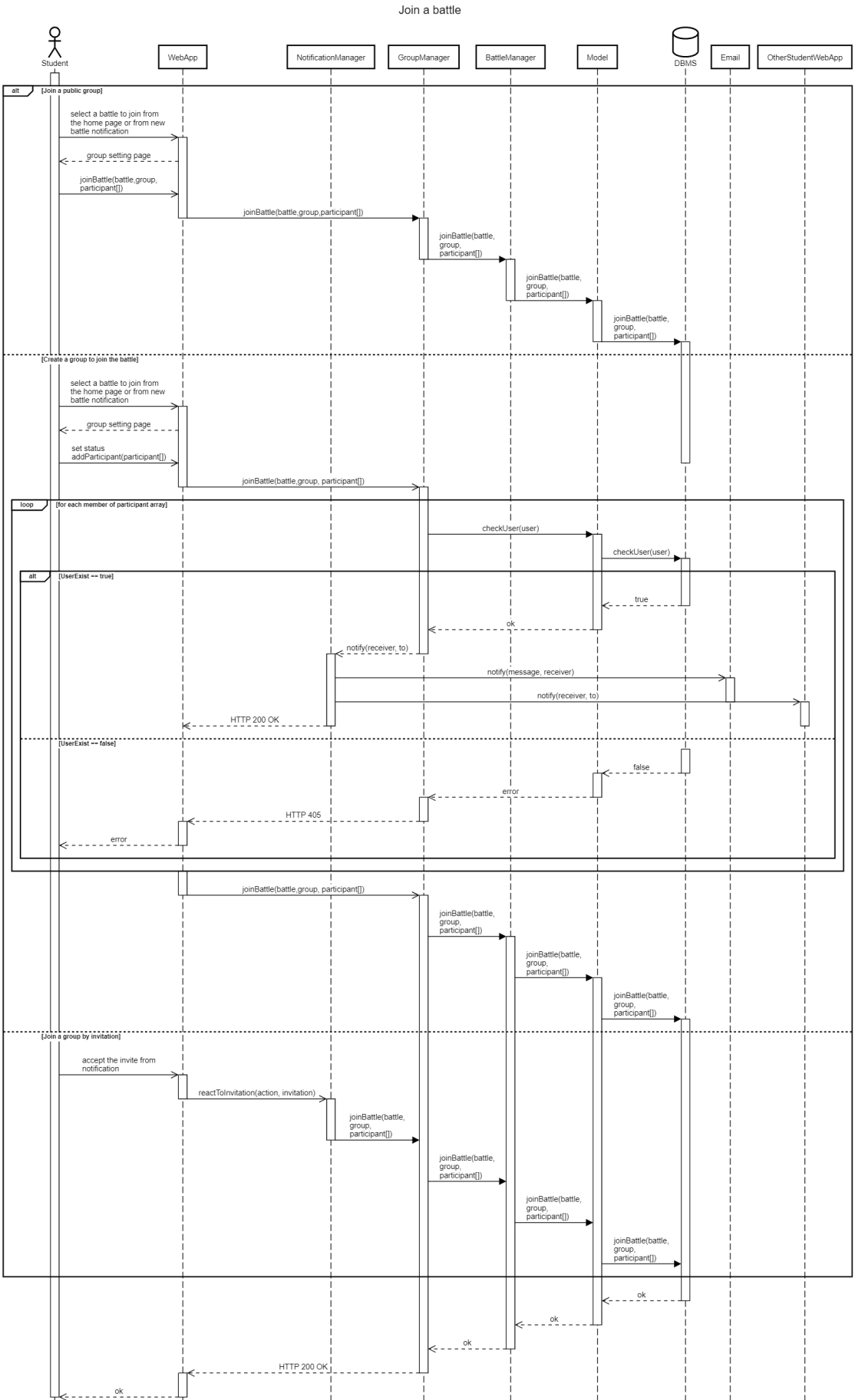


Figure 9: Join a tournament

- **Join a battle**: The sequence starts with a student selecting the battle setting from the WebApp. At this point there are three options:
  - Joining a public group: The sequence starts with a student selecting the battle from the WebApp. At this point, the WebApp notifies the GroupManager calling joinBattle(group,participant) which then updates the DMBS going through the BattleManager and the model.
  - Creating a group: just like in the previous, the student selects battle setting from the WebApp. At this point the student adds his/her group mates and decide whether the group will be public or private. Then GroupManager validates the students to add in the group through the DBMS, which then notifies the NotificationManager if the data is correct. If so, the NotificationManager sends the invitations to all the members of the group, otherwise if some user doesn't exist error 405 is notified.
  - Joining a group by invitation: in this last option a user accepts a notification to join a group and the dbms is updated passing through GroupManager, BattleManager and model.

  The sequence ends in the same way for all three cases, that is, the student is notified with the outcome message.


Join a battle
Student
WebApp
NotificationManager
GroupManager
BattleManager
Model
DBMS
Email
OtherStudentWebApp
alt
[Join a public group]
select a battle to join from the home page or from new battle notification
group setting page
joinBattle(battle,group, participant[])
joinBattle(battle,group,participant[])
joinBattle(battle, group, participant[])
joinBattle(battle, group, participant[])
joinBattle(battle, group, participant[])
[Create a group to join the battle]
select a battle to join from the home page or from new battle notification
group setting page
set status addParticipant(participant[])
joinBattle(battle,group, participant[])
loop
[for each member of participant array]
checkUser(user)
checkUser(user)
alt
[UserExist == true]
true
ok
notify(receiver, to)
notify(message, receiver)
notify(receiver, to)
HTTP 200 OK
[UserExist == false]
false
error
HTTP 405
error
joinBattle(battle,group, participant[])
joinBattle(battle, group, participant[])
joinBattle(battle, group, participant[])
joinBattle(battle, group, participant[])
[Join a group by invitation]
accept the invite from notification
reactToInvitation(action, invitation)
joinBattle(battle, group, participant[])
joinBattle(battle, group, participant[])
joinBattle(battle, group, participant[])
joinBattle(battle, group, participant[])
ok
ok
ok
HTTP 200 OK
ok

- **Push the code on Github**: The sequence diagram shows what happens when a student pushes the code to GitHub. The sequence starts with the Student pushing the code, then GitHub then notifies the BattleManager which then then computes the Student's score and updates the rank in the database. The ranking is then disposable in the WebApp.

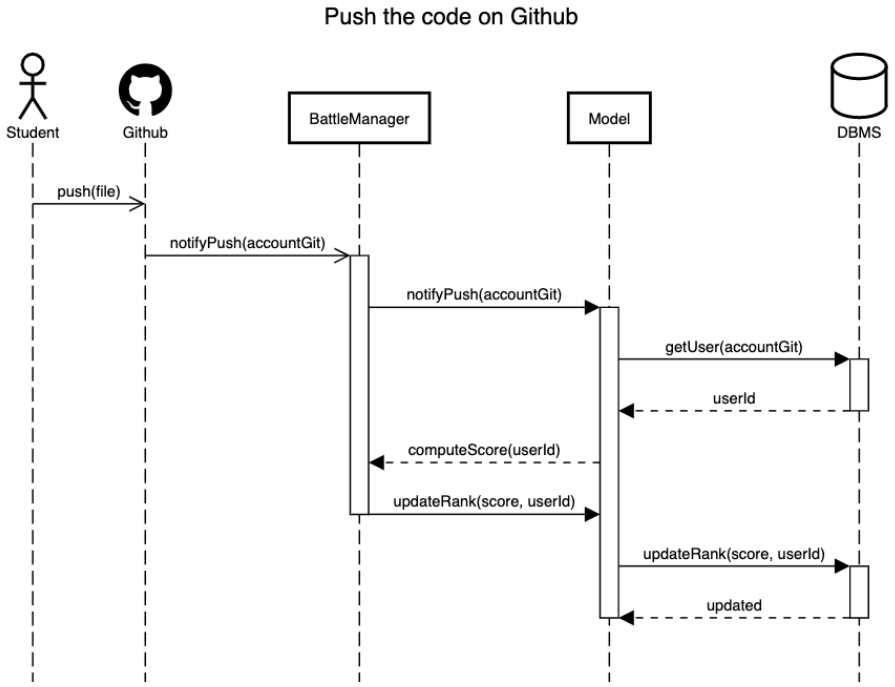


Figure 11: Push code on Github

- **Visualize active tournaments**: In order to visualize the active tournaments the user needs to open the WebApp and enter the home page. For further details see Figure 4.

- **Visualize the participants of a tournament**: The sequence starts with the user selecting a Tournament. This sends a getTournament(tournament) message to the TournamentManager. The TournamentManager then retrives the tournament information thorough the model and the DBMS and finally, displays the data to the WebApp.

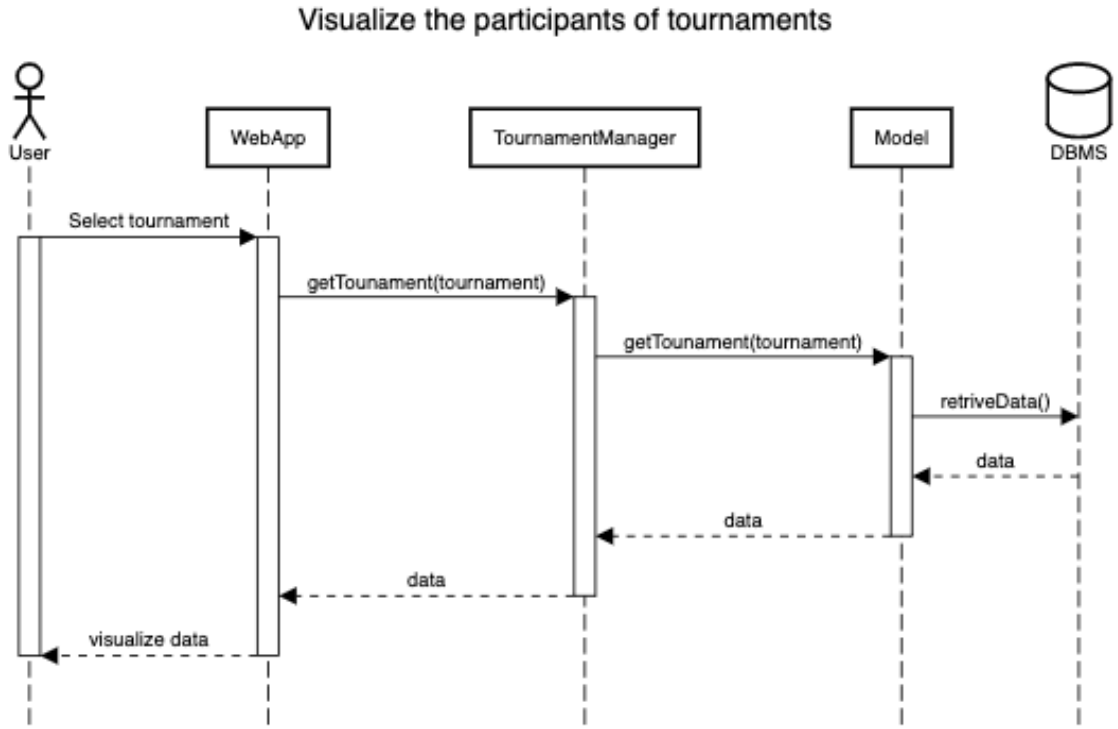


Figure 12: Visualize the participants of a tournament

- **Visualize user details**: The sequence diagram shows how the different components interact to visualize a user's details. The user first selects a user, which triggers the WebApp to send a 'getBadges(studentUsername)' message to the TournamentManager. The TournamentManager then retrieves the user's data from the Model, which retrieves the data from the DBMS. The data is then passed back up the chain, through the TournamentManager and WebApp, and finally visualized for the user.

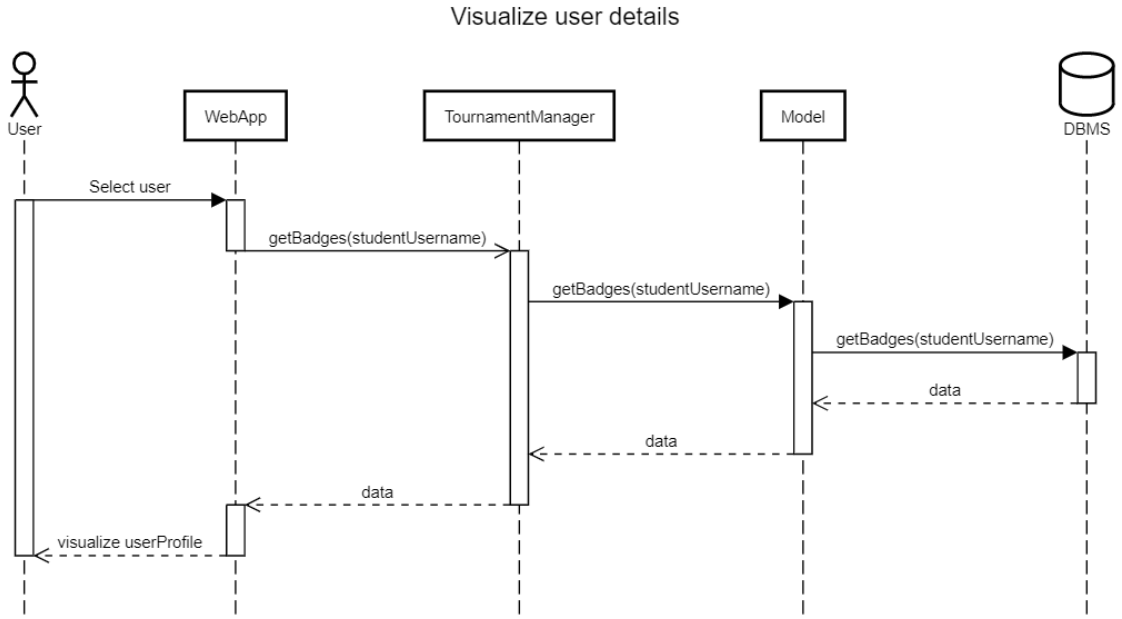


Figure 13: Visualize users details

- **Add an admin**: The sequence starts with the Admin selecting the Add Educator option. This sends a addAdmin(admin. tournament) message to the TournamentManager. The TournamentManager then validates the educator's information through the DBMS. If the information is valid, the TournamentManager sends a notification to the invited educator and registers the educator, as admin in that tournament, into the database. If the information is not valid the DBMS sends error 405. Finally, the TournamentManager sends a notification to the Admin whether the educator has been added or some error occured.

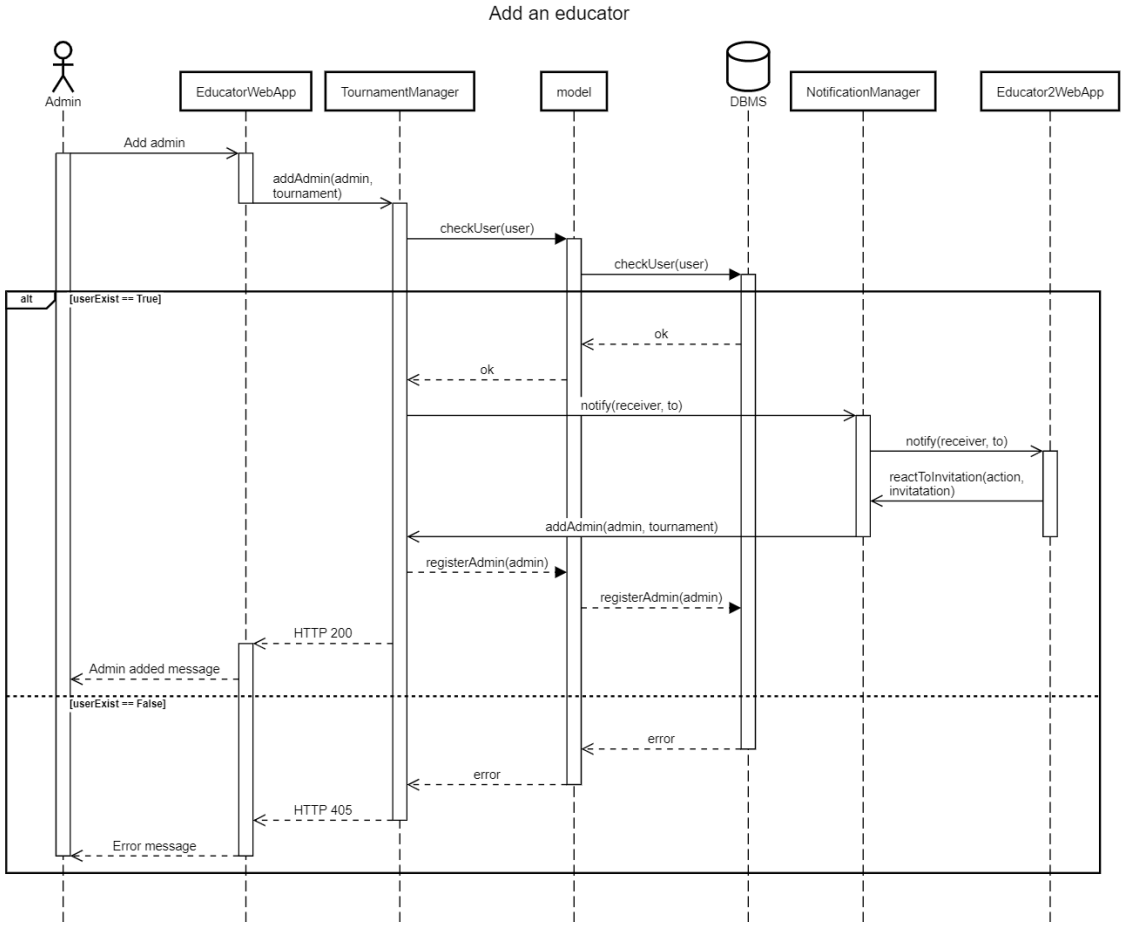


Figure 14: Add an admin

2.5 Component interfaces

Here we list the methods of every interface, with the expected parameters, effects on the system, the return values and codes. Server error are omitted for simplicity but has to be considered in case of server side failures by sending 500 type HTTPS code.

- **Profile Interface**
  - **sendCredentials(username, password)** is used to check the credentials of the user when he/she logs in.

Request data:	username: string password: string
Success response:	HTTPS 200 OK : the username of the receiver exists and the password matches.
Effects in case of success:	After this method it is created the session id associated to that user, so that further request incoming are accepted without the necessity of loggin in again.
Failure response:	* HTTP code : 400 Bad Request if some parameters are missing or are of the wrong type or there are too many of them * HTTPS 401 UNAUTHORIZED: if one between username and password are wrong.

- **sendCredentials(name,username,email,password, GitHubName, position)** is used to check the user's credentials when he/she signs up. Position parameter is used to define whether a user is an educator or a student.

Request data:	name: string username: string password: string email: string password: string GitHubName: string position: string
Success response:	HTTPS 200 OK : the username and the email have never been used, the password is secure (no repeated patterns, at least one number and special character, at least 8 characters long), GitHubName exists, position is either student or educator.
Effects in case of success:	After this method a new user is created and all the datas are saved in the DB.
Failure response:	* HTTP code : 400 Bad Request if some parameters are missing or are of the wrong type or there are too many of them * HTTPS 405 METHOD NOT ALLOWED: if the parameters sent do not match the requirements. * HTTPS 401 UNAUTHORIZED: if one between username and password are wrong.

- **Battle Interface**
  - **getBattles (tournament)** is used by both types of user to retrieve the battles managed or participated by user.

Request data:	tournament : string
Success response:	HTTPS 200 OK : The user is authenticated.
Effects in case of success:	Returns all the battles of the given tournament.
Failure response:	* HTTP code : 400 Bad Request if some parameters are missing or are of the wrong type or there are too many of them * HTTPS 405 METHOD NOT ALLOWED: if the parameters sent do not match the requirements.

- **createNewBattle(creator, tournament, description, groupSize, startDate, finishDate)** Creates a new battle with the provided description, groupSize, starting date and finish date. After the creation this method uses the method **sendNotificationToAllParticipants** of the Notification Manager interface.

Request data:	description : string creator: string tournament: string groupSize: int startDate: date finishDate: date
Success response:	HTTPS 200 OK : The battle is created .
Effects in case of success:	After this method a new battle is created and all the datas are saved in the DB.
Failure response:	* HTTP code : 400 Bad Request if some parameters are missing or are of the wrong type or there are too many of them * HTTPS 405 METHOD NOT ALLOWED: if the parameters sent do not match the requirements.

- **joinBattle(battle, group, participant[])** is used to join a battle with the given group. The participant's array contains the user to be added to the group.

Request data:	battle:string group : string participant: string
Success response:	HTTPS 200 OK : The user joined the battle with the given group.
Effects in case of success:	After this method the user correctly joined the battle with his/her group.
Failure response:	* HTTP code : 400 Bad Request if some parameters are missing or are of the wrong type or there are too many of them * HTTPS 405 METHOD NOT ALLOWED: if the parameters sent do not match the requirements.

- **notifyPush(accountGit)** is used to trigger the platform when a student pushes his/her code on GitHub and the system needs to run tests and update the rank.
- **changeEvaluation (battle, group, newEvaluation, comment)** is used by the administrator of a tournament to change the evaluation of a group at the end of a battle.
- **getGroups (battle)** returns all the groups in the battle.
- **computeScore(userId)** is used to compute the user's score of a battle.

Request data:	userId : string
Success response:	HTTPS 200 OK : The test has been runned and the score has been calculated.
Effects in case of success:	After this method the tests on the given code has been runned and the score is ready.
Failure response:	* HTTP code : 400 Bad Request if some parameters are missing or are of the wrong type or there are too many of them * HTTPS 405 METHOD NOT ALLOWED: if the parameters sent do not match the requirements.

- **Tournament Interface**
  - **getTournaments ()** returns all open tournaments (used from the home page).
  - **createNewTournament(name,endingDate, educator[], badgeFormula)**

Request data:	name : string endingDate: date educator[]: string badgeFormula: string
Success response:	HTTPS 200 OK : Tournament created. The formula for the badge is valid.
Effects in case of success:	A new tournament is created and registered in the DBMS.
Failure response:	* HTTP code : 400 Bad Request if some parameters are missing or are of the wrong type or there are too many of them * HTTPS 405 METHOD NOT ALLOWED: if the parameters sent do not match the requirements.

  - **checkFormula(formula)** checks if the formula is valid.
  - **joinTournament(username, tournament)** puts username in the tournament if the tournament exists and is not closed.

Request data:	username : string tournament: string
Success response:	HTTPS 200 OK : ok .
Effects in case of success:	User joins the selected tournament.
Failure response:	* HTTP code : 400 Bad Request if some parameters are missing or are of the wrong type or there are too many of them * HTTPS 405 METHOD NOT ALLOWED: if the parameters sent do not match the requirements.

  - **getBadges(studentUsername)** returns all the badges of the given student.

Request data:	studentUsername : string
Success response:	HTTPS 200 OK : ok .
Effects in case of success:	All the badges of the given students are returned
Failure response:	* HTTP code : 400 Bad Request if some parameters are missing or are of the wrong type or there are too many of them * HTTPS 405 METHOD NOT ALLOWED: if the parameters sent do not match the requirements.

  - **chechExpirationDate()** checks if the tournament ending date has expired.
  - **getTournaments(studentUsername)** returns all the tournaments in which the studentUsername is participating.
  - **getTournament(tournament)** gets the data relative to the specified tournament.
  - **addAdmin(admin, tournament)** adds the given admin to the list of administrator of the tournament.
  - **closeTournament(tournament)** closes the given tournament.

- **Group Interface**
  - **joinBattle(battle,group, participant[])**

Request data:	battle : string group : string participant[]: string
Success response:	HTTPS 200 OK : ok .
Effects in case of success:	User join the selected group and consequently the battle.
Failure response:	* HTTP code : 400 Bad Request if some parameters are missing or are of the wrong type or there are too many of them * HTTPS 405 METHOD NOT ALLOWED: if the parameters sent do not match the requirements.

  - **createGroup(status)** creates a group with the provided status (status can be Private or Public).
  - **getParticipants(group)** returns the participants of the given group.

- **Model Interface**
  - **checkCredentials(username, password)** is used to check the credentials for the log in.

Request data:	username: string password: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	The user credentials are verified, the user can correctly log in.
Failure response:	The given credentials are incorrect, please retry.

– **checkCredentials(username, email)** is used to verify that does not exist a user with the same credentials during the sign up.

– **createNewTournament(name, endingDate, educator[], badgeFormula)**

Request data:	name: string endingDate: string educator[]: string badgeFormula: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	A new tournament is created and saved in the DBMS.
Failure response:	∗ The educator does not exist ∗ The badge formula is invalid ∗ The data is invalid

– **createNewBattle(description, creator, tournament, groupSize, startDate, finishDate)**

Request data:	description: string creator: string tournament: string groupSize: int startDate: date finishDate: date
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	A new battle is created and saved in the DBMS.
Failure response:	∗ Invalid parameters

– **joinTournament(username, tournament)**

Request data:	username: string tournament: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	User joins the selected tournament.
Failure response:	∗ Tournament is closed

– **joinBattle(battle, group, participant[])**

Request data:	battle: string group: string participant[]: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	User joins the selected group and consequently the battle.
Failure response:	* Invalid group * Participant does not exist

- **checkUser(user)**

Request data:	user: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	The database confirms the existence of the user.
Failure response:	* User does not exist

- **notifyPush(accountGit)** is used when someone submits their code, and the system needs to identify who pushed the code by using the Git username.
- **registerAdmin(admin)** registers an admin to the group of administrator of the battle
- **getBadges(studentUsername)** returns the badges af the given student
- **getTournament(studentUsername)** returns all the tournaments in which studentUsername is participating
- **getTournament(tournament)** returns the participants of a tournament
- **getNotifications(studentUsername)** gets all the notifications sent and received by a student
- **updateRank(score, userID)** register the new score and updates the rank.

- **Email Push Interface** This interface is exposed by the email service provider and allows to send emails. This interface is used in pairs with the Push Interface to send notifications to the clients. The e-mails sent to the users, are going to have a link created by this interface, which redirects the receiver to his/hers personal notification page. Notifications to the web app are sent only if the web app is connected, conversely, emails are sent in either case.
  - **notify(message, receiver)** send an email with a message to the receiver.

Request data:	message: string receiver: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	The email is successully sent.
Failure response:	* The email is not existent. * Email service is down

- **sendNotification(newTournament)** is used to send the notification of a new tournament to the user.

- **sendNotification(newBattle)** is used to send the notification of a new battle to the user.
- **Push Interface** This interface is exposed by the webApps and allows to push messages to the webApp. It can be implemented for example using the JavaScript WebSocket API.
  - **notify(notification)** push a single message to the webApp. This method is used by the server to push notification to connected webApps, if the client is not connected the message is not sent.

Request data:	notification: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	After this method the client webApp received the notification.
Failure response:	* Disconnected webApp.

- **Notification Interface** Every method in this interface will be implemented to send notifications both on the email and the webApp of the recipient.
  - **reactToInvitation(action, invitation)** used to accept or reject invitations. Action is a string that corresponds to the action that the caller of the method wants to do on the invitation. Possible actions are accept or reject. The caller cannot reject an accepted invitation, nor accept a rejected invitation. When the invitation is accepted the methods of the Tournament Manager Interface and Group Manager Interface are called to put the user in the correct admin group or student group respectively.

Request data:	action: string invitation : string
Success response:	HTTPS 200 OK : the sender is autenticated, the invitation exists and the receiver of the invitation is the caller of this method. Action is a valid string and corresponds to an action that can be performed on the invitation.
Effects in case of success:	The invitation in successfully modified and the receiver of the invitation is inserted either in the group or in the admins of the tournament.
Failure response:	* HTTP code : 400 Bad Request if some parameters are missing or are of the wrong type or there are too many of them * HTTPS 401 UNAUTHORIZED: if the caller is not autenticated. * HTTPS 405 METHOD NOT ALLOWED: if the parameters do not satisfy the conditions of the success response. * HTTPS 406 NOT ACCEPTABLE: if the receiver is a student and is not part of the tournament where there is the battle where the group participates to.

  - **getNotifications()** gets all the notifications sent and received by the caller user.

Request data:	-
Success response:	HTTPS 200 OK : the sender is autenticated. It is returned a JSON object containing the invitations with their characteristics.
Effects in case of success:	The invitations are successfully retrieved by the DB and returned.
Failure response:	* HTTP code : 400 if there are parameters. * HTTPS 401 UNAUTHORIZED: if the caller is not autenticated.

- **Notification Manager Interface** This interface is used by group, battle and tournament manager to send informative broadcast notifications
  - **sendNotificationToAll(newTournament)** sends in broadcast to all the students a notification for the creation of the newTournament.
  - **sendNotificationToAllParticipants(tournament, type)** used to inform the participants of a tournament of two types of events: the closure of the tournament and the creation of a battle. this method will be automatically invoked by the server upon the closure of a tournament o the creation of the battle
  - **notify(receiver, to)** used to ask the notification manager to notify an user (either an admin or a student) of an invitation. "to" is an object that can be either a tournament (in case an admin invites a collaborator to his tournament) or a group (for invitations for students)
- **Group Manager Interface**
  - **joinBattle(battle, group, participant[])** This method allows to add a participant to a group if the username belongs to a student and the user is not already in the group, otherwise it sends an error.
- **Tournament Manager Interface**
  - **addAdmin(username, tournament)** This method adds an admin to a tournament if the username belongs to a teacher profile and the user is not already an admin, otherwise it returns an error.
- **DB Interface**
  - **checkCredentials(username, password)**

Request data:	username: string password: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	The user credentials are verified, the user can correctly log in.
Failure response:	The given credentials are incorrect, please retry.

  - **checkCredentials(username,email)**

Request data:	username: string email: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	The database checks if already exists an user with this credentials, if not the user is registered.
Failure response:	The given credentials are already in use.

  - **registerUser(name, username, password, position)**

Request data:	name: string username: string password: string position: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	The user and his/her data are registered in the DBMS.
Failure response:	User not registered.

– **registerTournament(name, endingDate, educator[], badgeFormula)**

Request data:	name: string endingDate: date educator[]: string badgeFormula: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	A new tournament is created and registered in the DBMS. The given formula for the badge is valid or is null
Failure response:	∗ The educator does not exist ∗ The badge formula is invalid

– **registerBattle(description, creator, tournament, groupSize, startDate, finishDate)**

Request data:	desription: string creator: string tournament: string groupSize: int startDate: date finishDate: date
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	A new battle is created and saved in the DBMS.
Failure response:	∗ Invalid parameters

– **joinTournament(username, tournament)**

Request data:	username: string tournament: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	User joins the selected tournament.The datas are registered in the DBMS.
Failure response:	∗ Tournament is closed

– **joinBattle(battle, group, participant[])**

Request data:	battle: string group: string participant: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	User joins the selected group and consequently the battle. The datas are registered in the DBMS
Failure response:	* Invalid group * Participant does not exist

– **checkUser(user)**

Request data:	user: string
Success response:	This function returns a success code to the caller if the operation is successful.
Effects in case of success:	Given user exists.
Failure response:	* Given user does not exists

– **registerGroup(partecipants)** registers the given participants to the group

– **getUser(accountGit)** returns the userID associated to the given GitHub username

– **updateRank(score, userID)** updates the rank with the score of the group of the given userID

– **retriveData()** returns all the data of a tournament

– **registerAdmin(admin)** registers the given educator as administrator of the tournament

– **getNotifications(studentUsername)** gets all the notifications sent and received by a student

– **getTournaments(studentUsername)** returns the tournaments in which the studentUsername is involved

– **getBadges(studentUsername)** returns the badges of the given studentUsername

2.6 Selected Architectural Styles and Patterns

2.6.1 3-tier Architecture

The construction of CKB will follow a 3-tier architecture, offering numerous advantages due to the system's modularization into three independent layers:

1. **Presentation Tier**: Positioned at the top level, this tier encompasses the customer interface. Its primary role involves organizing data received from the application tier and presenting it to customers in a more intuitive and understandable manner.

2. **Business Logic Tier**: This tier houses the application's business logic, determining how the application makes decisions and performs calculations. Additionally, it facilitates the passing and processing of data between the surrounding layers.

3. **Data Tier**: Encompassing the Database system, this tier provides an API to the application tier, enabling access to and management of the data stored in the database.

Adopting this architectural approach ensures enhanced flexibility, allowing for the independent development and updating of specific system components. Furthermore, the inclusion of a middle tier between the client and data server enhances data security. Specifically, information is accessed through the application layer

rather than directly by the client, offering improved protection for stored data.

We decided on having a single model architecture, this might become a bottleneck in large and complex applications as all data-related operations are handled by a single model; however, CKB can be considered a medium sized application and we don't plan to expand it extensively in the future, so we believe that inside a single server the amount of requests to the single model will be sustainable.

Having a sigle model carries some advantages: first, it centralizes business logic, making it easier to maintain and update; then, it simplifies communication between components since they interact with a single source of truth; it grants consistency since no synchronization operations are necessary.

In addition, our application has data that is strictly related to one another, so it might be messy to have multiple models.

Scalability can be provided with horizontal scalability, by deploying the server on multiple machines.

2.6.2 Model View Controller pattern

To implement the web application, allowing users access to CKB functionalities, the decision was made to adhere to the Model-View-Controller (MVC) pattern. This pattern comprises three key components:

- **Model**: This component houses the application's data and provides methods for its manipulation.
- **View**: Responsible for displaying the various visual representations of the data, offering diverse perspectives on the information.
- **Controller**: Positioned between the model and the view, the controller responds to events (such as button presses) by executing predefined reactions. These reactions may involve operations on the model, subsequently reflected in the view.

The division of responsibilities among these three elements fosters increased decoupling of components, leading to several advantages such as enhanced re-usability and simplified implementation.

2.6.3 Thin Client

The thin client paradigm is implemented for the interaction between the front-end and the back-end. A thin client needs to perform very few computations, but it has to handle the communication and display data to the user. It's convenient as it requires less effort to create new client implementation because of the lack of logic. A thin client does not rely on local storage since all the business logic is on the application servers, which have enough computing power.

Our web application will have JavaScript code on the client to have engaging animations and so increasing user experience, however all the data that is displayed on the screen and the computation performed will be provided by the server, hence, we can safely say that our application is thin client.

2.7 Stateless

CodeKataBattle's application primarily adopts a stateless architecture, although it incorporates sessions for authentication, introducing a degree of statefulness due to the inherent storage of state information on the server side. These sessions are utilized to maintain user identity across subsequent requests.

A stateless web application architecture provides numerous advantages that contribute to scalability, simplicity, and ease of maintenance. Key benefits include:

Scalability: Stateless applications inherently scale more effectively. Each client request contains all necessary information for processing, allowing servers to handle requests independently without relying on shared session data. This facilitates the distribution of workload across multiple servers in a load-balanced environment.

Caching Opportunities: Stateless applications are well-suited for caching strategies. Responses to requests can be cached at various levels, such as Content Delivery Networks (CDN) and proxy servers, without concerns about session-specific data.

Easier Testing: Stateless applications streamline the testing process. Each request can be treated as an independent unit, simplifying unit testing and integration testing. This approach eliminates the need for complex setups or management of intricate state scenarios.

2.8 Other design decision

2.8.1 Firewall

There are two firewalls that shield the application server from the outside security threads. One between the web server and the web application view, and another between the Load Balancer and the mobile application view.

2.8.2 Database

The application uses a relational database, which is the most appropriate choice given that it deals with structured data. Given the prevalence of repetitive operations within the system, a relational database excels in swiftly executing these tasks. The efficiency gained from leveraging the relational model significantly outweighs the drawbacks associated with non-relational models. Furthermore, the use of SQL for querying enhances the database's versatility, aligning with industry standards and facilitating seamless integration into common practices.

3 User Interface Design

Here we provide the mockups for the web application, which will be used by users to interact with the system.

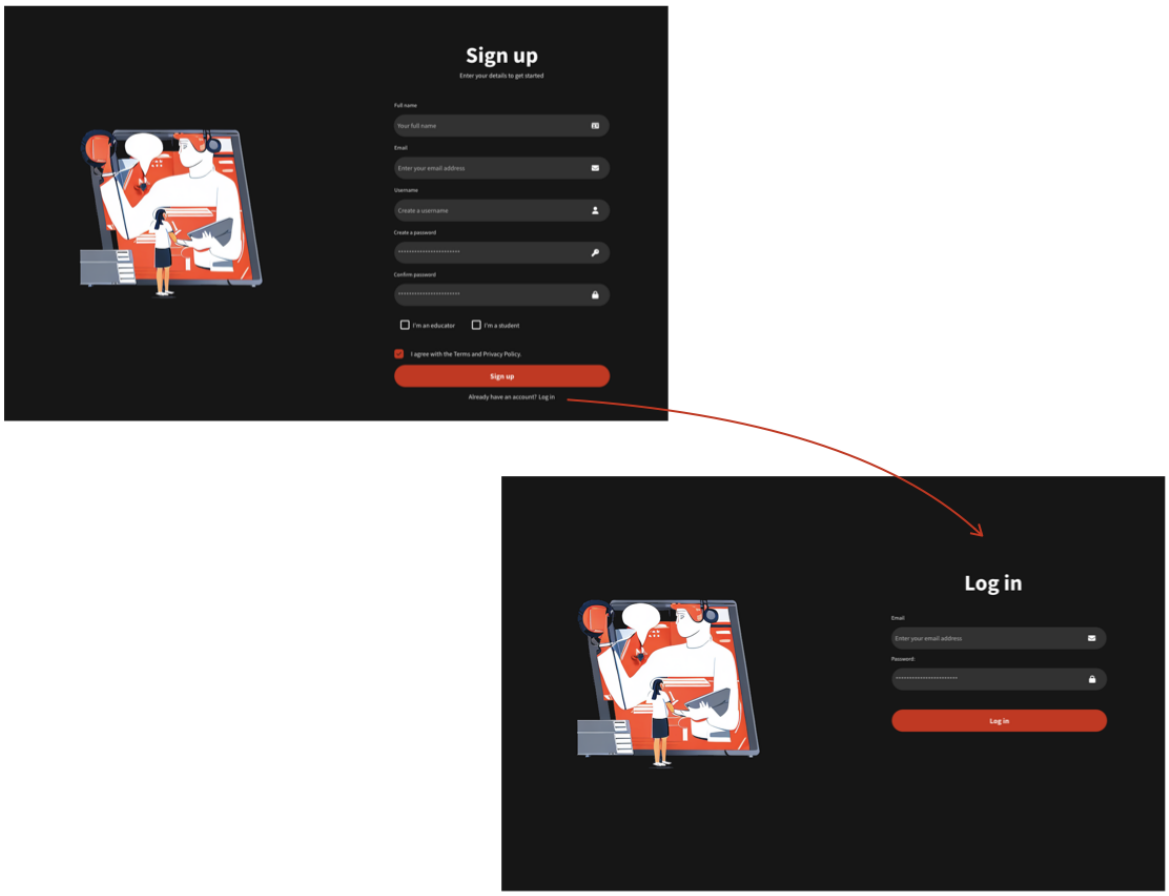


Figure 15: Signing up or logging in the system

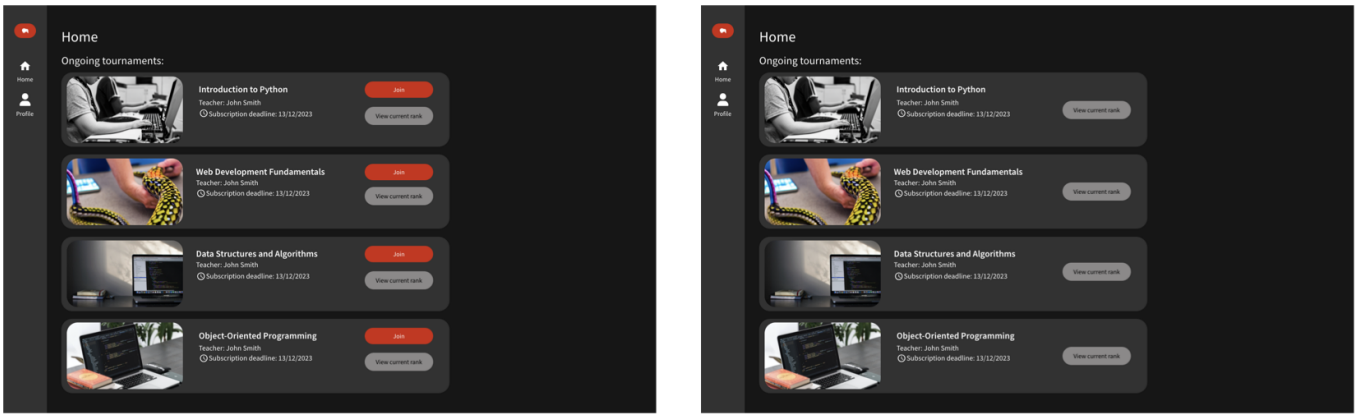


Figure 16: Students and Educator home page MOCKUP

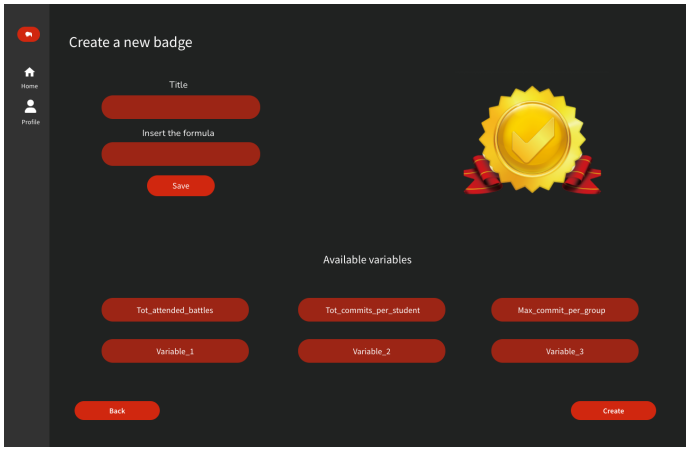


Figure 17: Creating a badge

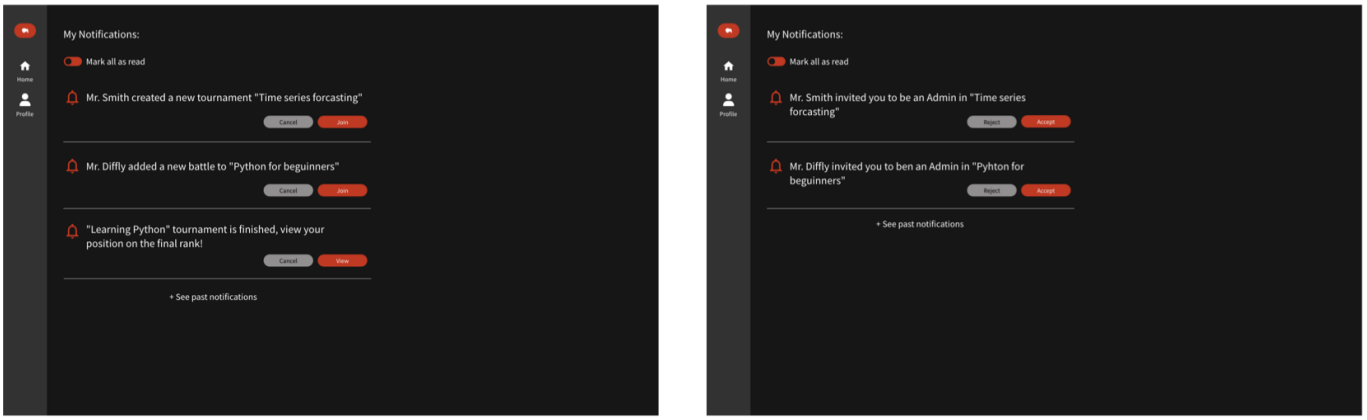


Figure 18: Students and educators notification screen

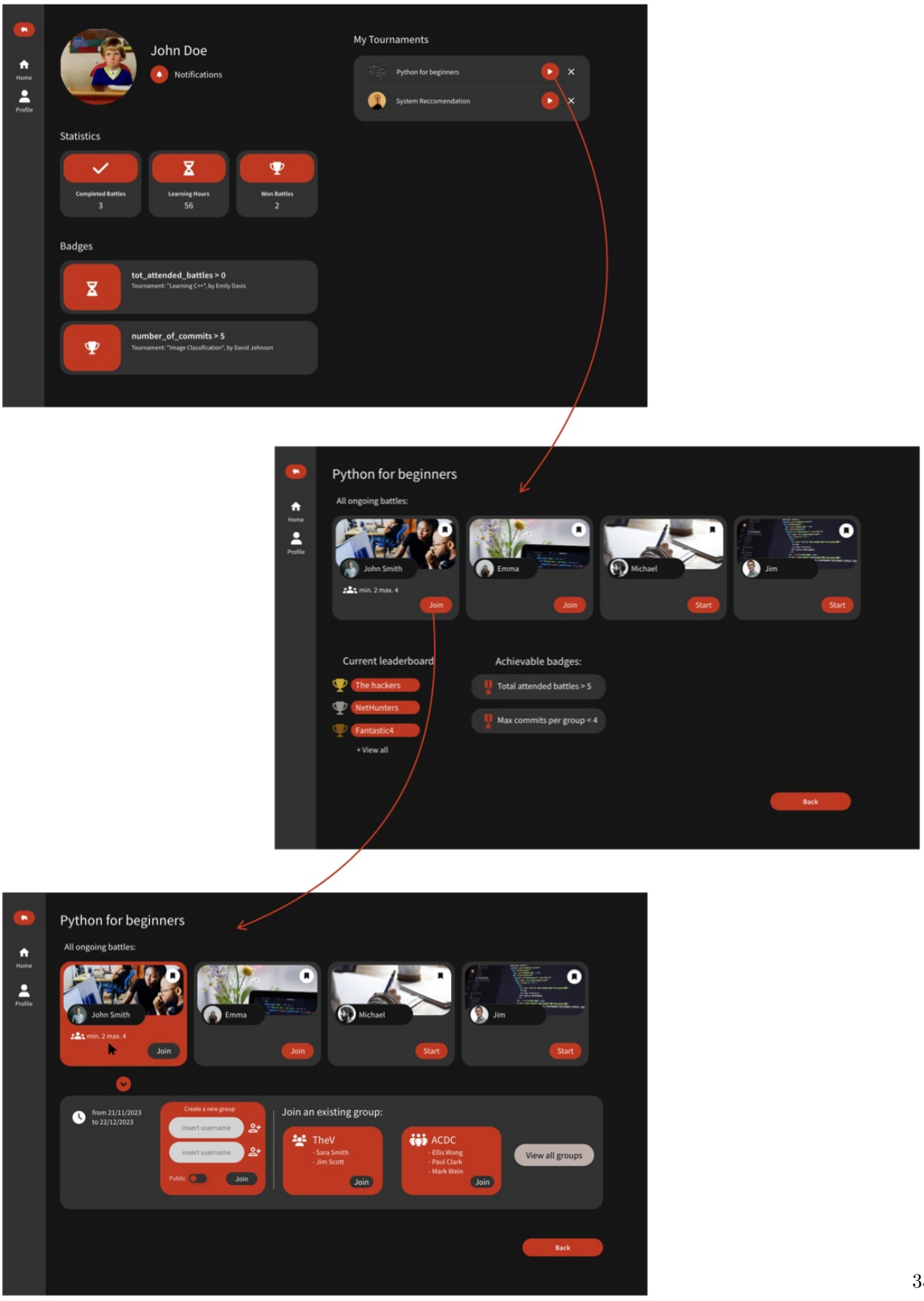


Figure 19: Student joining a battle

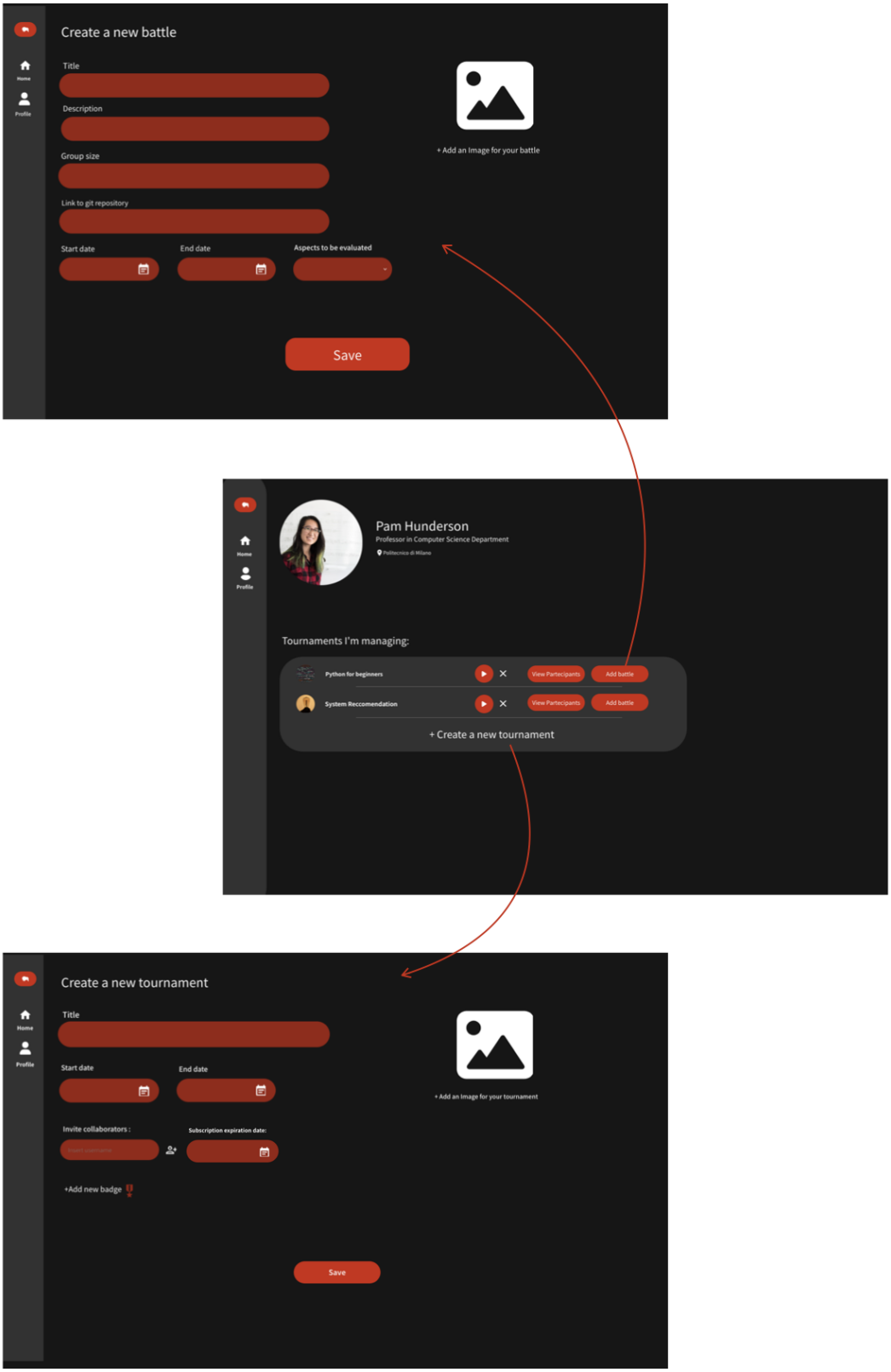


Figure 20: Educator creating tournament or battle

4 Requirement traceability

Requirements:	R1: The system allows user to sign up R2: The system allows registered user to log in
Components:	• WebApp • WebServer • Profile Inspector • Model • DBMS

Requirements:	R3: The system allows educator to create a tournament R4: The system allows admin to invite collaborators to the tournament R6: The system allows admin to create badges when they create a tournament R8: The system allows admin to close a tournament R11: The system allows user to visualize other user's details R12: The system allows user to visualize currently active tournaments R14: The system allows student to join a tournament R28: The system awards the badge to the deserving student at the end of the tournament
Components:	• WebApp • WebServer • Tournament Manager • Model • DBMS

Requirements:	R10: The system allows user to visualize the rank
Components:	• WebApp • WebServer • Tournament Manager • Battle Manager • Model • DBMS

Requirements:	R5: The system allows admin to create battles within the tournament they are managing R7: The system allows admin to evaluate manually participant's work R13: The system allows user to see the battles created in a tournament
Components:	• WebApp • WebServer • Battle Manager • Model • DBMS

Requirements:	R15: The system allows tournament's participant to join a battle R16: The system allows tournament's participants to create a group R17: The system allows tournament's participant to join an existing group
Components:	• WebApp • WebServer • Group Manager • Battle Manager • Model • DBMS

Requirements:	R29: The system allows participant of a group to invite new group members unless the invited is in another group of the same battle
Components:	• WebApp • WebServer • Group Manager • Battle Manager • Notification Manager • Model • DBMS

Requirements:	R18: The system allows tournament's participant to submit their code R22: The system creates a GitHub repository containing the code kata R24: The system after each push before the deadline pulls the code, analyzes it and run tests
Components:	• WebApp • WebServer • GitHub • Battle Manager • Model • DBMS

Requirements:	R9: The system allows admins to accept invitations R20: Notify the users of the newly created tournaments R27:The system notifies the tournament's participant when an admin close a tournament they are subscribed to
Components:	• WebApp • WebServer • Notification Manager • Tournament Manager • Model • DBMS

Requirements:	R21: Notify the users of the newly created battle R19: The system allows student to accept invitations to groups R26: The system notifies all the participants of a tournament when consolidation stage has finished and the final rank is available
Components:	• WebApp • WebServer • Notification Manager • Battle Manager • Model • DBMS

Requirements:	R23: The system sends the link of the created GitHub repository to all students who are subscribed teams when the registration deadline expires
Components:	• WebApp • WebServer • Notification Manager • GitHub • Model • DBMS

Requirements:	R25: The system after each push updates the battle's and tournament's rank
Components:	• WebApp • WebServer • GitHub • Tournament Manager • Battle Manager • Model • DBMS

5 Implementation, Integration and Test Plan

5.1 Overview

In this section we discuss the implementation and testing strategy that we deem the best for the system at hand.

5.2 Implementation Plan

The implementation plan depends strongly on the development team size; here we are assuming that there will be three developers working on the implementation and testing.

5.2.1 Features Identification

The features to implement can be divided in three groups, according to the chosen architecture:

1. The data layer comprises
   (a) Writing a stub for the model
   (b) The identification of the schema for the tables of the SQL database.

   (c) Writing the actual SQL for the schema creation.
   (d) Writing the required queries on the DB.
   (e) Writing the model and the relative interface.

2. The business layer includes this components and connectors:
   (a) The profile inspector component, the notification manager component with the integration with the email and the integration between the two components.
   (b) The group manager and the integration with the notification manager
   (c) The tournament manager and the integration with the notification manager.
   (d) The battle manager and the integration with the notification manager.
   (e) The integration between the notification manager and the tournament manager, with the tournament manager interface.
   (f) The integration between the notification manager and the group manager, with the group manager interface.
   (g) integration between battle manager and GitHub.
3. The presentation layer is composed of this tasks
   (a) Writing the html strucure of each page.
   (b) Write the CSS code and javascript code for the animations of the web site.
   (c) Writing the javascript listener functions to react to the user interaction with the web page and the callback functions to receive and display data from the server.

Let's name the three developer A, B and C. The development will be carried out as follows: First, A and B will write the stub for the Model since it is fundamental to test the business and presentation components. Meanwhile C will be dedicated just for the frontend development. Since in the first part the stub will not be ready yet he will focus on writing the HTML code for the pages using mockup data to display some content in the various container of the pages. In parallel he will write the CSS code to make the page prettier. When A and B will finish the mockup, A will work just on the data layer and will perform the tasks in the order defined in the list above. Conversely, B will write the business components in the order defined in the list above.

5.2.2 Component Integration and Testing

Let's introduce again the three developers A, B and C. The three will have the same roles defined in the previous section. Integration and testing will be carried out in a thread, top down fashion. For every component developed in the business tier the integration will happen first with the model stub. When C has written the functions that retrieve (or push) the data from the server relative to the component developed by B, both C and B will test the system. Subsequently, when A has implemented the functions of the model relative to that component, he/she will substitute them in the model stub and the same tests will be runned again to check the proper integration with that part of the model.

Let's make an example for clarity purpose: C has wrote the HTML code for every page and is going on with the CSS file. B finished implementing the notification manager and the profile inspector. At this point C will implement the listener for incoming notifications, the function associated with the form for login and sign up and will test the behaviour of the website. In particular B and C will test the login procedure, the sign up, the receipt of notifications, the receipt of emails. When every test is successful they will go on with their work. When A finishes the part of the model associated with the functionalities already tested by B and C, he/she will update the stub of the model and the development group will perform again the tests.

5.3 Integration diagram

In Figure 21, we illustrate the system integration, indicating the order with color coding. The integration sequence is as follows:

1. Red
2. Green
3. Yellow
4. Blue
5. Light blue
6. Orange
7. Purple

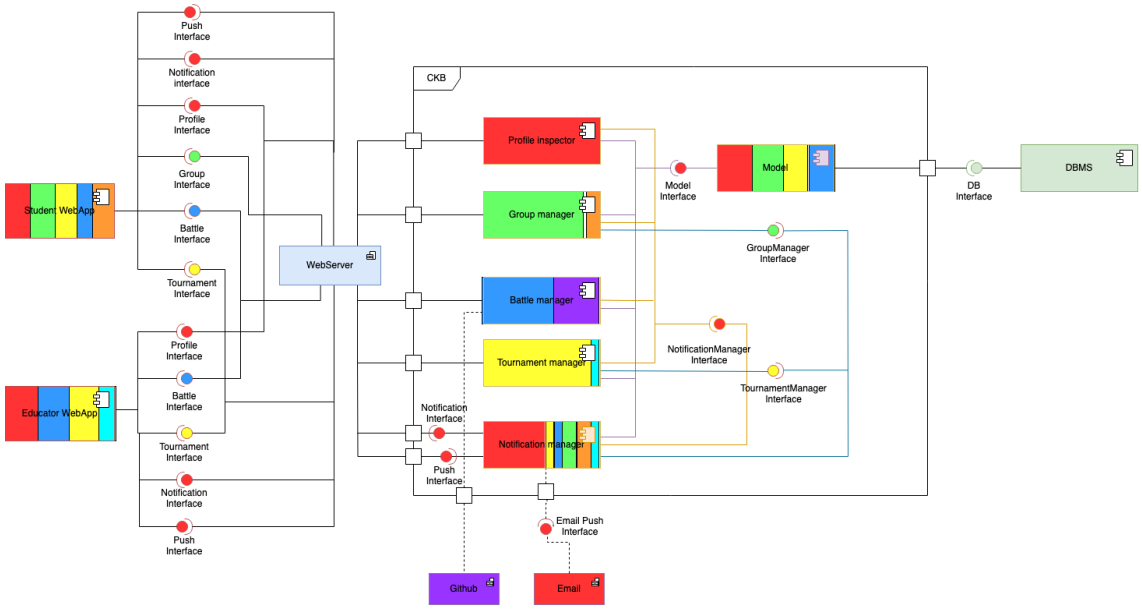


Figure 21: Integration Diagram

5.4 System Testing

Following integration testing, the system undergoes comprehensive system tests to ensure its robustness and effectiveness. The testing process involves an alpha test and a crucial beta test.

1. Alpha Test:

   Execution: Conducted during the integration process by KCB developers. Scope: All functions are rigorously tested with a substantial number of test cases, aiming for a minimum code coverage of 80Functional Requirements: Verify compliance with functional requirements. Non-Functional Requirements: Assess system performance under heavy load. The database and data warehouse is loaded with a significant volume of data, simulating the expected data that will be reached by the application after some years of running. Performance Testing: Evaluate the processing and response times of all functions, ensuring they meet specified criteria.

2. Beta Test: Execution: Carried out in a real production environment with the collaboration of some computer science class that will use the platform during the year as a training tool.

Objectives:

1. Bug Verification: Confirm the absence of bugs in a real distributed environment.
2. Usability Check: Assess the usability of the system. User Acceptance Testing (UAT): Verify that the system meets the needs of various user types.

Additional Details:

1. Real-World Scenario: The beta test creates a real-world scenario, allowing for the identification of any unforeseen issues or challenges that may arise during actual usage.
2. Usability Assessment: The focus is not only on functionality but also on the usability of the system. This involves evaluating how well the system aligns with the needs of different user categories.
3. Acceptance Testing: The beta test serves as an acceptance test, ensuring that the system aligns with the expectations and requirements of end-users.

By conducting these alpha and beta tests, the CodeKataBattle development team aims to validate the system's correctness, performance, and usability in real-world scenarios, laying the foundation for a reliable and user-friendly solution.

6 Time Spent

	Section 1	**Section 2**	**Section 3**	**Section 4**
Federica Maria Laudizi	0:30h	5:00h	4:00h	2:00h
Antonio Marusic	0:30h	5:00h	2:00h	4:00h
Sara Massarelli	0:30h	7:00h	2:00h	2:00h

Table 1: Work Hours Schedule

7 References

- GitHub REST API documentation
- The specification of the RASD and DD assignment of the Software Engineering II course, held by professor Matteo Rossi, Elisabetta Di Nitto and Matteo Camilli at the Politecnico di Milano, A.Y 2023/2024;
- Slides of Software Engineering 2 course on WeBeep;
- RASD Sample from A.Y. 2022-2023