Design Document

Francesco Spangaro - Tosetti Luca - Francesco Riccardi

07 January 2024

POLITECNICO MILANO 1863

Prof.
Matteo Camilli

Contents

1   Introduction

1.1   Purpose

The purpose of this document is to provide an exhausting implementative description of the platform that will be developed (CKB platform). The document focuses on the description of the architectural styles and decisions that will be adopted, the modules that compose the platform and their interfaces. The document contains several details regarding the deployment choices and the runtime view of the platform's core functionalites. The document contains some mockups of the user interface design. The document covers the implementation, integration and testing processes required to correctly implement the CKB platform.

1.2   Scope

CodeKataBattle (CKB) is a platform that aims to give Educators an easy-to-use way to let them propose homework and/or lessons to their Students. One of CKB platform's goals is to give Students the possibility to improve and acquire new software developing skills by particpating in several coding battles found in tournaments proposed by Educators. The platform lets Educators create tournaments and battles within tournaments. This is done to challenge Students to compete in them, so that they can find the best possible solution to the battle's problem. That solution will be then automatically evaluated by the platform, which will give it a score and, eventually, will be corrected by the Educator who created the battle. The correcting Educator can change the Students' score in that battle. The platform also allow Educators to add several recognition badges for the Students' work. These badges can be personalized by the Educators themselves.

From an architectural point of view we have decided to adopt a 4-Tier Client-Server architecture combined with a MSA server side, in addition to a MVC software architectural choice.

1.3 Definitions, acronyms, abbreviations

1.3.1 Definitions

Term	Definition
4-Tier Architecture	→ A 4-Tier architecture in the informatic systems field is a SW and HW architecture in which a running application/platform is divided in four different modules, also called "layers", which usually are: Presentation Layer, WebServer layer, Logic Layer, Data Layer.
Presentation Layer	→ The top layer of the 4-Tier architecture. It takes care of the interaction between the user and the application.
WebServer Layer	→ The second layer of the 4-Tier architecture. It takes care of handling the requests sent by the users through a browser application.
Logic Layer	→ The third layer of the 4-Tier architecture. It takes care of implementing the real application logic that allows the application/platform to work seamlessy.
Data Layer	→ The bottom layer of the 4-Tier architecture. It takes care of all the data provided by the users or by the application and with which the application itself has to interact. The interactions can include queries, updates, deletions, ...

Term	Definition
Microservice architecture	→ An architectural style that consist in the creation of an application/platform as a suite of small services, each one handling a part of the application's business logic and that communicates with each other through lightweight protocols (such as HTTP).
Model-View-Controller	→ An architectural pattern used to develop the software logic of an application. This pattern consists in dividing the application in three different parts: Model, View and Controller.
Model	→ Part of the MVC pattern. It provides the Controller the methods it needs to access the application's data and to modify them.
View	→ Part of the MVC pattern. It takes care of the visualization of the data contained in the model and the interaction with the user.
Controller	→ Part of the MVC pattern. It receives commands from the user and executes them by modifying the View and/or Model.
DBMS	→ A software system that allows the creation, manipulation and querying of one or more databases in an effecient and correct way.
Unit Testing	→ A type of testing where lines of code are tested in order to discover bugs contained in them.
Functional Testing	→ A type of testing where the SW functionalities are tested in order to check if they respect the functional requirements.

Term	Definition
Performance Testing	→ A type of testing where the system is filled with data within the expected workload. This is done in order to detect bottlenecks, inefficiencies and HW or network issues.
Load Testing	→ A type of testing where the system gets tested at increasing workloads up until it can support it. Used to detect memory issues.
Stress Testing	→ A type of testing where the system is overwhelmed in a controlled environment. This is done by making requests its resources cannot handle or by reducing the amount of comcputational power the system has. Used to check whether the system can recover smoothly from a failure.

1.3.2   Acronyms

Acronym	Meaning
MSA	→ MicroServices Architecture
MVC	→ Model-View-Controller
RASD	→ Requirement Analysis and Specification Document
DD	→ Design Document
CKB	→ CodeKataBattle
DB	→ DataBase
DBMS	→ DataBase Management System
HTTP	→ HyperText Transfer Protocol

1.3.3 Abbreviations

Abbreviation	Meaning
e.g.	→ Exempli gratia, latin phrase meaning "for example".
i.e	→ Id Est, latin phrase meaning "that is".
id	→ Identifier.
SW	→ Software.
HW	→ Hardware.

1.4 Revision history

- 07 January 2024: version 1.0

1.5 Reference documents

GitHub references:

- Official documentation to get started with GitHub: → `https://docs.github.com/en/get-started/quickstart`

- Official documentation about fork process → `https://docs.github.com/en/get-started/quickstart/fork-a-repo`

- Official documentation about GitHub actions → `https://docs.github.com/en/actions`

UML official specification → `https://www.omg.org/spec/UML`

Sequence diagrams specification → `https://www.uml-diagrams.org/sequence-diagrams.html`

Component diagrams specification → `https://creately.com/blog/software-teams/component-diagram-tutorial/`

1.6 Document structure

- ***Section 1: Introduction***
  This section offers a brief description of the problem and the platform/application that will be developed. It describes the major purpose of this document, a very brief recap of the domain that is described in major detail in the

RASD document. In this section are inserted definitions, acronyms and abbreviations used in the document, its revision history and refereced documents or web pages.

- *Section 2: Architectural Design*
  This section is the main part of the document. It describes the architectures used to create the platform, the CKB platform's components, its interfaces, its deployment structure and finally its runtime behaviour. All these aspects are described through several diagrams such as: component diagrams, sequence diagrams, deployment diagrams and other generic diagrams used to give a representation of the platform's most important features.

- *Section 3: User interface design*
  This section describes the platform's user interface design. It contains several mockups of the interface that the Educators and Students will find when they access to the platform. The presented mockups refer to the client-side experience through an appropriate browser application.

- *Section 4: Requirements traceability*
  This section describes the connection between the RASD and DD document, by providing a complete map of the requirements expressed in the RASD document to the modules presented in this document.

- *Section 5: Implementation, Integration & Test plan*
  This section describes the plan followed for implementing, testing and integrating the platform's components, the order in which these operations are performed and a description of how they work.

- *Section 6: Effort spent*
  This section contains all the information about the time spent by each group member in order to complete the document and its division by each document's section.

2 Architectural Design

2.1 Overview: High-level components and interactions

To ensure high maintainability, security and reliability, the service is structured by following the four-tier architecture model. Figure 1 shows how the tiers are divided, and what are the relations between each system's tier.
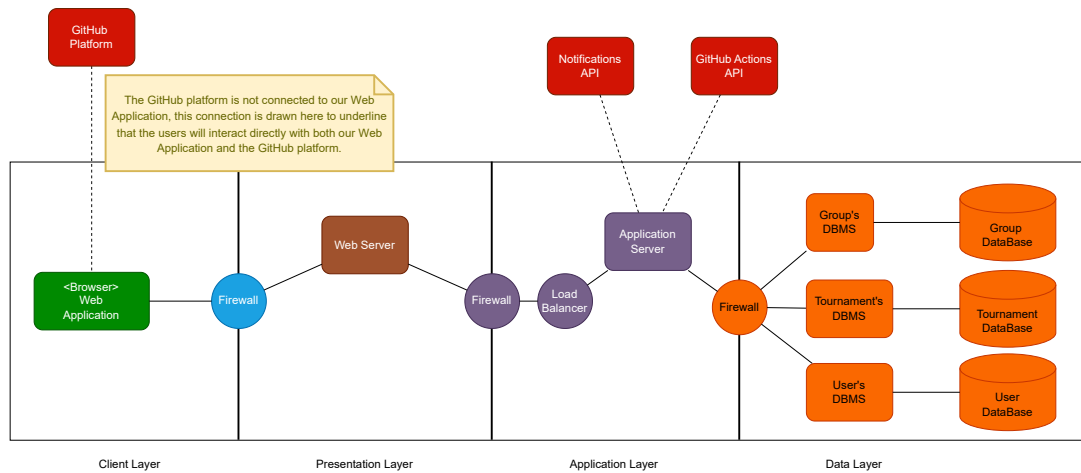

Figure 1: Four-tier architecture

The main components are:

- **Web Application:** The Web application allows users to connect to the platform's services. The Web application can be accessed by any device that is connected to the internet and can browse the Web. Students and Educators will have different views of the Web application, because they have different parameters that they can view and manage. It is connected directly to the GitHub Platform to give Educators the possibility of seeing their Students' uploaded solutions. It is connected to the GitHub platform to also give Students the possibility of seeing their solutions uploaded to the platform in real time.

- **Web Server:** The Web server is what manages the Web application. It is the main container for the JavaScript and general backend code for the platform. The Web server is connected to the Application server because

it needs to be automatically updated when new grades for Students' solutions are generated by the platform.

- **Application Server:** The Application server is the main backend of the CKB Platform. It contains all the code needed for the platform to run smoothly and without interruptions. It contains the logic needed to answer the API requests made by the users to the platform. It automatically evaluates Students' solutions proposed via the GitHub platform and uploads the new grade to the web server.

- **DBMS:** The DBMS is the main interpreter between the CKB platform and the data stored onto the databases.

- **DataBase:** The databases store all the informations needed by the application.

- **External Services:** These services provide informations and funtionalities that the CKB platform alone could not provide. These funtionalities include a *GitHub actions API* that notifies the platform of the upload of a new Student's solution, a *notification API* that notifies Students when a new tournament or a new battle in a tournament they are subscribed to is created or more generally whenever there's the need to notify a User. It's also included a connection to the *GitHub platform* since the code needs to be uploaded from the GitHub platform to the CKB platform automatically. This is done through the *GitHub actions APIs.*
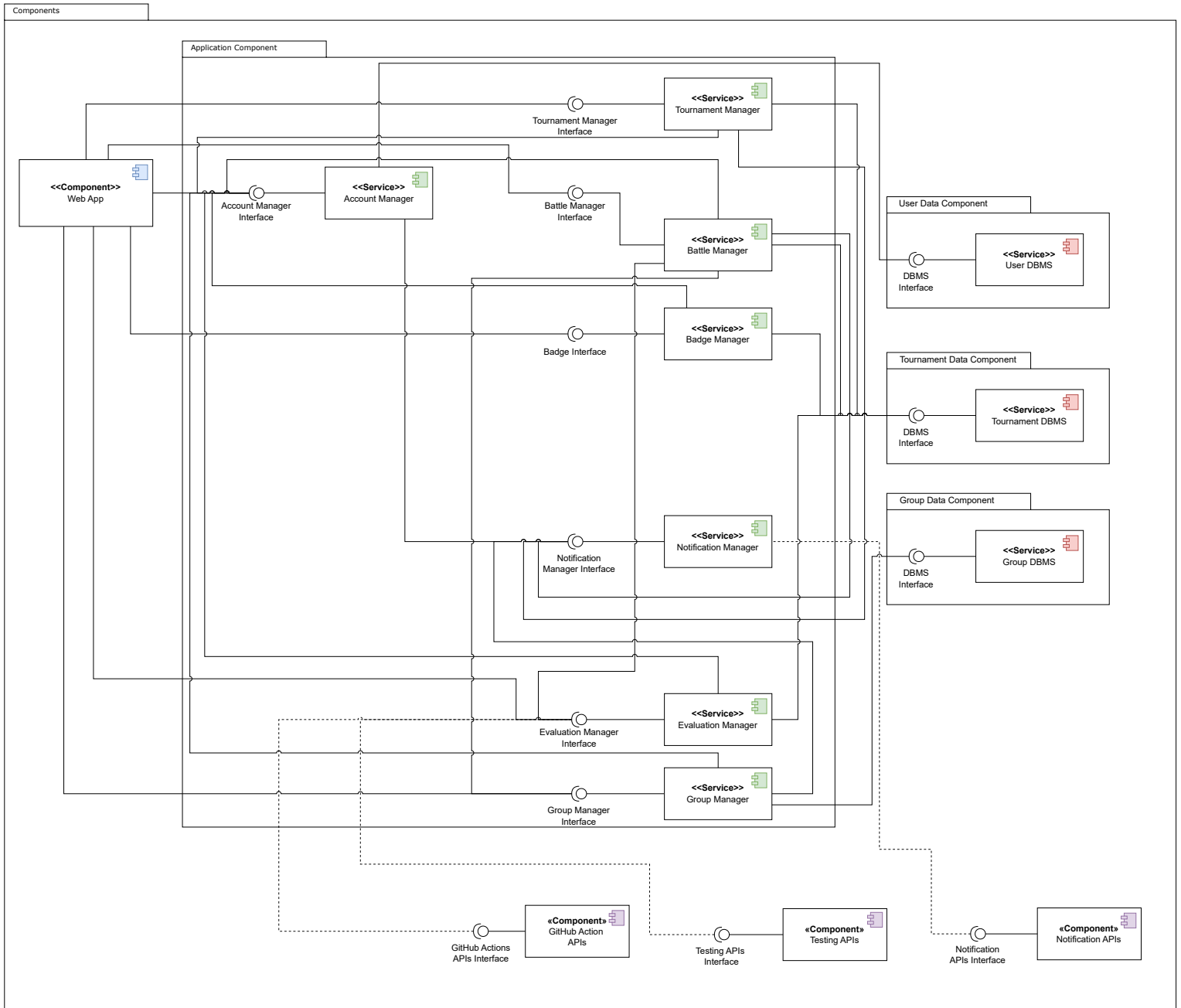
2.2 Component view

Figure 2: Component diagram

In Figure 2 we can see a more detailed diagram, representing all the components previously described. The Web App is what Students and Educators connect to via their web browsers. It collects all of the Students' and Educators' requests, forwards them to the right service or component and sends back different responses.

- **Account Manager:** The Account Manager is the module that handles the client's logins and creates the session. It checks the client's data and permissions and creates a session with the data it collects. This data session is what is used by the other services and interfaces to grant the user different permission levels.
  E.G. A Student can use the Tournament Manager differently from an Educator, since the Student can subscribe to a tournament and see the battles it contains, while an Educator can use the interface to manage the tournament.

- **Tournament Manager:** The Tournament Manager is the module that lets Educators create and manage tournaments and lets them grant permission to create battles in their tournaments to other Educators.
  It lets Students visualise all open tournaments and subscribe to each tournament they want.

- **Battle Manager:** The Battle Manager is the module that lets Educators create and manage battles within their tournaments, or within tournaments for which they have been granted permission to create and manage battles from the tournament's creator.
  It lets Students visualise the battles in tournaments they are subscribed into and lets them subscribe to each battle they want, if they are within the subscription deadline for the battle they want to participate in and if their group respects the battle's requirements.

- **Badges Manager:** The Badges Manager lets Educators create and manage badges for their tournaments. It lets Educators define new rules for obtaining badges in tournaments. It also checks if Students have obtained any badges and updates the Students' pages on rule completion.

- **Notification Manager:** The Notification Manager is the module in charge of notifying Students of new tournaments' creation, new battles' creation in a tournament they are subscribed into and when they receive an invite to join a group.
  The Notification Manager achieves this by interfacing itself with some external Notification APIs, in charge of sending each notification.

- **Evaluation Manager:** The Evaluation Manager is the module responsible for grading each Student's entry as solution for a battle. The Evaluation Manager uses external Testing APIs to test the Students' code for correction checking and the GitHub Actions APIs for downloading the Students' solution they upload on the GitHub platform.
  The Evaluation Manager interacts with the Tournament DBMS to update and upload the Students' grades for each battle, after the Evaluation Process ends.
  The Evaluation Manager also lets Educators manually evaluate each Student's solution and upload the grade to the platform via the Tournament DBMS.

- **Group Manager:** The Group Manager lets Students create new groups to participate in battles with. It lets Students send invites to other Students.

- **DBMS:** The DBMS is the module in charge of interacting directly with the DataBases, translating each query written so that the DataBases understand the requests and responding with the correct data requested.
  In this project, to ensure the MicroServices design, we used three different DBMSs with three different DataBases.

- **GitHub Actions APIs:** The GitHub Actions APIs are external APIs used to have the CKB platform interact with the GitHub platform, since the Students will upload their solutions on the latter.

- **Testing APIs:** The Testing APIs are external APIs used to run tests on the code written by the Students and submitted as a solution for each battle.

- **Notification APIs:** The Notification APIs are external APIs used to notify Students of some events happening on the CKB platform.
  E.G. A new tournament is created, a new battle is created in a tournament they are subscribed into and they received an invite for joining a group.
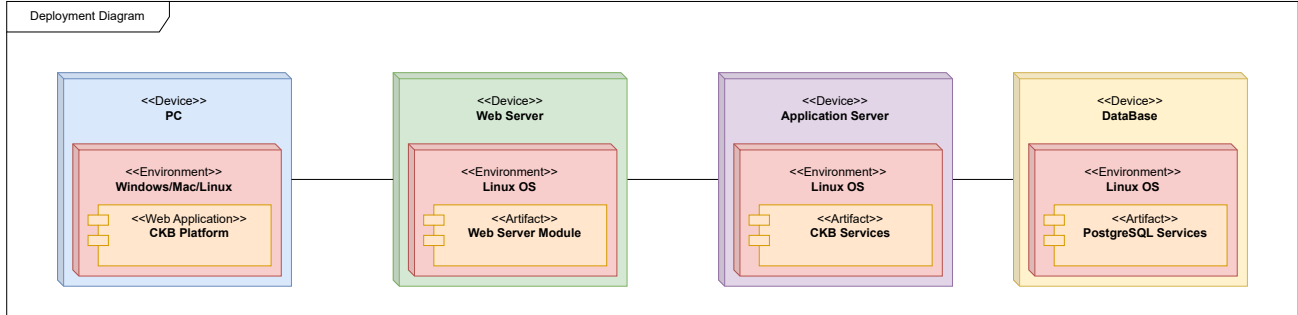
2.3   Deployment view

Figure 3: Component diagram

The Deployment Diagram in figure shows the components needed for the correct system behaviour. The External APIs are not included, since they are already implemented and found online.

Each device has its own operating system downloaded, the different tiers in the figure are:

- **Tier 1:** The First tier is the client tier. On the client's machine can be run any OS (Windows 11 32 bit, for example) and any web browser. The web application needs to be able to correctly run on any web browser downloaded on any OS.

- **Tier 2:** The Second tier is the Web Application tier. Here no logic is found, is where the CKB platform can be accessed. The Web Server is stored here, is contains no logic besides the bases needed for a correct execution of the website (CSS, JavaScript and Bootsrap 5.0).
  This tier simply receives requests from the clients and forwards them to the Third tier, it then receives and shows the answer it received from the Third tier to the requesting client.

- **Tier 3:** The Third tier stores the Application Servers. The servers include all the main logic needed by the CKB platform to correctly perform. The Third tier receives all the forwarded requests from the clients, interacts with the Fourth tier to get data and evaluates a response. The response is then sent up to the Second tier, which will show the answer to the client. Each module is mapped onto this tier. This tier is the one that mainly interacts with all external APIs.

- **Tier 4:** The Fourth tier is composed by the DBMSs needed by the CKB platform for correctly interacting with the DataBases. The DBMSs function as gateways between the Third tier and the actual DataBases. They perform the actions requested by the Third tier and respond with the data requested.

2.4 Component Interfaces

The Component Interfaces here described are the most important ones exposed by the components. Only the most relevant method parameters and methods are shown.

- **Account Manager**

  - **registerUser(userData):** This method takes a data struct containing all the user's data and contacts the DBMS in order to register the new user in the system. A reply is then sent to the caller.

  - **loginUser(userData):** This method takes a data struct conntaining all the user's data, contacts the DMBS, runs a query that checks for the data matching the user's data provided and, if found, creates a valid session with the user's data, then logs the user in the platform.

  - **checkUser(sessionData):** This method takes the session data and checks whether the user is a Student or an Educator. This method is needed and will be used by all the methods that need to show the created battles and tournaments by an Educator.

- **Tournament Manager**

  - **getCreatedTournaments(userId):** This method takes the user's ID from the session data, contacts the DBMS which will check if the user has created any tournaments and returns the json containing the new page to display, containing all the user's created tournaments. If no tournaments are found, the json returned will be an empty page.

  - **createTournament(userId, tournamentData):** This method takes the user's ID and the tournament's data, which will be contained in an ad-hoc created struct. The method will contact the DBMS and create a new tournament with the data contained in the tournamentData. The method will then return a json file containing a confirmation page. If any check returns negative, or the process incurs in any errors, an error message will be returned, and nothing will be inserted in the DB.

  - **closeTournament(tournamentId, userId):** This method takes the user's ID and a tournament's ID, contacts the DBMS which will check if the tournament corresponding to the tournament's ID provided is created by the user with the provided user's ID, then, if

the check returns a positive result, updates the tournament in the database and sets its state to "closed", then returns a json containing a confirmation page. If the check provides a negative result, an error message will be displayed.

– **getSubscribedTournaments(userId):** This methods takes the user's ID, checks if the user is a Student by interacting with the DBMS and, if the user is indeed a Student, returns the page containing all the tournaments he is subscribed to in json format. If the user is not a Student or an error occurs while interacting with the DBMS, an error page will be displayed.

– **enterTournament(userId, tournamentId):** This method takes the user's ID and tournament's ID and checks whether the user is a Student and if the tournament is still open. If the user is a Student and the tournament is open, the Tournament Manager will interact with the DBMS and subscribe the provided Student to the provided tournament by updating the database. A confirmation page will appear after the update. If an error occurs while interacting with the DBMS, the user is not a Student or if the tournament is closed, an error page will be displayed.

- **Battle Manager**

    – **getCreatedBattles(userId):** This method takes the user's ID from the session data, contacts the DBMS which will check if the user has created any battles and returns the json containing the new page to display, containing all the user's created battles. If no battles are found, the json returned will be an empty page.

    – **createBattle(userId, battleData, tournamentId):** This method takes the user's ID, the tournament's ID and the battle's data, which will be contained in an ad-hoc created struct. The Battle Manager will check if the user is an Educator through the checkUser method and if the battle will be contained in a tournament in which the user's has permission to create one. If all the checks pass then the method will contact the DBMS and create a new battle with the data contained in battleData. The method will then return a json file containing a confirmation page. If any check returns negative, or the process incurs in any errors, an error message will be returned, and nothing will be inserted in the DB.

    – **getSubscribedBattles(userId, tournamentId):** This methods takes the user's ID and the tournament ID, checks if the user is a Student and if the user is subscribed to the tournament corresponding to the given tournament ID by interacting with the DBMS and returns a page containing all the battles the user is subscribed to, contained in the tournament provided. If the user is not a Student or an error occurs while interacting with the DBMS, an error page will be displayed.

– **getBattleData(battleId, userId):** This method takes the battle's ID and the user's ID and performs a check to understand whether the user is a Student or an Educator. If he's a Student, the returned page will contain data corresponding to the battle provided, which are deemed interesting to a Student, if he's an Educator, different types of data will be provided, also relative to the battle provided. If an error occurs while interacting with the DBMS, an error page will be provided.

– **enterBattle(groupId, battleId):** This method takes the group's ID and battle's ID and checks whether the group exists, if the battle is still open, if each Student in the provided group is subscribed to the tournament containing the provided battle and if the group is not already subscribed to the specified battle. If all checks pass, the Battle Manager will interact with the Tournament DBMS and subscribe the provided group to the provided battle by updating the database. A confirmation page will appear after the update. If an error occurs while interacting with the DBMS or any of the checks are not passed an error page will be displayed.

- **Badge Manager**

  – **getCreatedBadges(userId):** This method takes the user's ID from the session data, contacts the DBMS which will check if the user has created any badges and returns the json containing the new page to display, containing all the user's created badges. If no badges are found, the json returned will be an empty page.

  – **createBadge(userId, badgeData):** This method takes the user's ID and the badge's data, which will be contained in an ad-hoc created struct. The Badge Manager will check if the user is an Educator through the checkUser method and if the badge will be contained in a tournament in which the user's has permission to create one. If all the checks pass then the method will contact the DBMS and create a new badge with the data contained in badgeData. The method will then return a json file containing a confirmation page. If any check returns negative, or the process incurs in any errors, an error message will be returned, and nothing will be inserted in the DB.

  – **editBadge(badgeId, userId, badgeData):** This method takes the user's ID, the badge's ID and the new badge data, which will be contained in an ad-hoc made struct. The method then contacts the DBMS, which will check if the badge is contained in a tournament created by the user corresponding to the user provided, if the badge exists and, if the badge is not a tournament the user created, if the user has permission to edit the tournament. If each check passes, then the old badge's data will be overwritten by the new data and a json containing a confirmation page will be returned, else an error will be displayed.

- **deleteBadge(badgeId, userId, tournamentId):** This method takes the user's ID, the badge's ID and the tournament's ID and, after having run checks to verify that the user has permission to delete the specified badge in the specified tournament, contacts the DBMS which will delete the badge from the database. If everything in the procedure is correct, a confirmation page will be returned, else an error page will be displayed.
- **checkBadges():** This method, once per day, at random intervals, cycles each group subscribed to each open battle in the platform and checks for each Student if they achieved any of the badges that were assigned to that battle. If the check returns positive, the Badge Manager will grant the Student his newly achieved badge which will be visible on his profile. This method will also be invoked once when a battle is being closed, so that badges will be granted also on the last commits made by each group.
- **getBadges(userId):** This methods takes the user's ID, checks if the user is a Student by interacting with the DBMS and, if the user is indeed a Student, returns all the badges he achieved while using the platform. If the user is not a Student or an error occurs while interacting with the DBMS, an error page will be displayed.

- **Notification Manager**

  - **sendGroupInvite(senderId, receiverId):** This method takes the sender's and receiver's ID, checks whether both are indeed Students, then by interfacing with external Notification APIs sends a notification to the receiving Student via e-mail.
  - **receiveInvitationAnswer(senderId, receiverId, answer):** This method takes the sender's and receiver's ID, checks whether both are indeed Students, then, after the Student who previously received a group invitation answers it, sends the answer to the Student who asked him to be in a group toghether via e-mail, using external Notification APIs.
  - **sendTournamentNot(tournamentId):** This method takes the newly created tournament's ID and automatically sends each Student subscribed to the platform an e-mail, notifying them of the creation of the new tournament, using external Notification APIs.
  - **sendClosedTournamentNot(tournamentId):** This method takes the closed tournament's ID and automatically sends each Student subscribed to that tournament an e-mail, notifying them of the closure of the tournament, using external Notification APIs.
  - **sendBattleNot(battleId):** This method takes the newly created battle's ID and automatically sends each Student subscribed to the tournament containing the new battle a notification, notifying them

of the creation of this battle via e-mail, using external Notification APIs.

– **evalBattleNot(battleId, groupId):** This method is called after the computation of the "evaluate" and "editScore" methods. It takes the battle's ID and the evaluated group's ID, then sends each group member a notification via e-mail, letting them know that a new score was assigned to them for their newly uploaded solution. This is achieved using external Notification APIs.

– **sendRegistrationNot(userId):** This method is called after a user's registration procedure is completed. Once the user's data is saved in the user's database, an e-mail, containing a confirmation message for the registration procedure, is sent to the user. This is achieved using external Notification APIs.

- **Evaluation Manager**

– **editScore(battleId, userId, score, studentID, description):** This method takes the user's ID, the battle's ID, the Student's ID and the new score to assign to the corresponding Student for the corresponding battle. The method checks if the user has permission to edit the Student's grade in the corresponding battle and, if he has permission, will contact the DBMS and update the Student's old grade with the new one provided for the specified battle. A description parameter is not mandatory, but if the Evaluation Manager finds it the grade is updated with the description provided. A confirmation page will be sent back to the user. If the platform runs into any error in the process, or if the user doesn't have the necessary permissions, an error page will be displayed, and the database will not be updated.

– **evaluate(battleId, groupId):** This method takes the battle's ID and the group's ID, then performs a search to find the tests written to automatically evaluate the battle provided, runs them on the group's provided solution and assigns each group member a grade on a scale from 0 to 100. The grade will be automatically generated based on the test creator previously specified data.

– **getScore(battleId, groupId):** This method takes the battle's ID and the group's ID and returns the latest score assigned to the provided group for the specified battle. If no score is found, a score of 0 will be automatically returned.

- **Group Manager**

– **createGroup(studentId, invitedIds):** This method takes the provided Student's ID and an array of the invited Students' IDs, checks if each ID corresponds to a Student's ID saved in the platform and, if so, creates the group, saving it in the group's database.
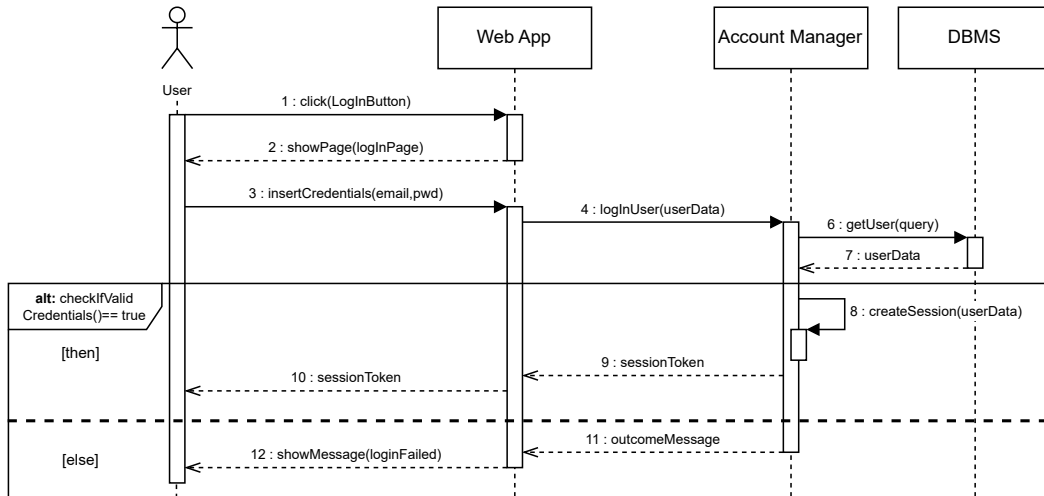
2.5 Runtime view

Figure 4: User login sequence diagram

Most of the actions described in the following Sequence Diagrams require the User (either a Student or an Educator) to be logged in the platform to actually perform the actions. The diagram above represents the User's login process.
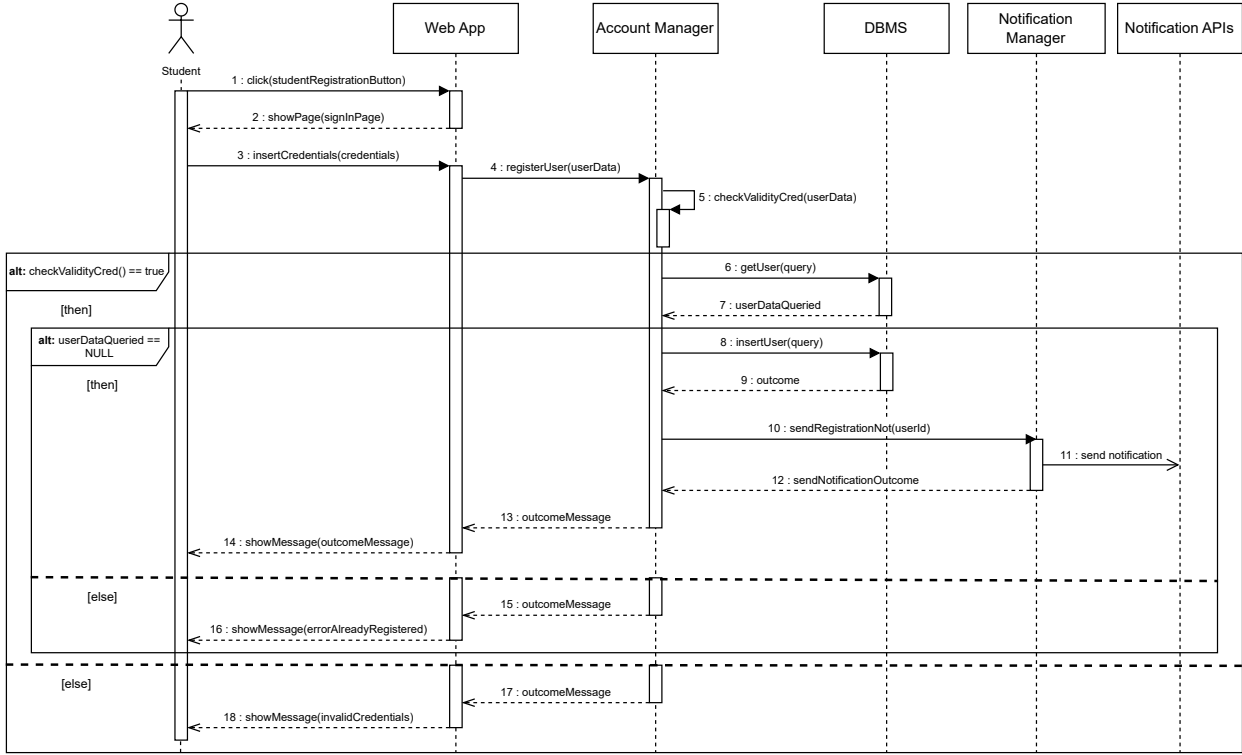
2.5.1 Student sign-up to the platform

Figure 5: Student sign-up sequence diagram

The above diagram represents a Student's signing up process. The Student accesses the CKB platform WebApp Homepage through their browser (omitted for simplicity). He then clicks on the "Student Registration" button as shown in figure 18. The WebApp will show a form which the Student has to fill with their informations such as name, surname, username that want to use in the platform, email, password, attended school, . . .

Once the Student confirms the registration the WebApp contacts the Account Manager which evaluates the request. The Account Manager is in charge of checking the validity of the Student's credentials, i.e. checking whether the name, surname or username inserted by the Student contain some not acceptable characters, the password doesn't respects security standards, . . .
Other than that the Account Manager takes care verifing if the Student is already registered, i.e. if the used email already appears in the DB. This is done by interacting with the DBMS.

At this point we can have three possible situations:

- **Credentials validation went wrong:** In this case the WebApp will show the Student an "Invalid credentials" error message

- **Student has already registered with the same email:** In this case the WebApp will show the Student an error message stating they are already registered.

- **Valid credentials and email used for the first time:** In this case the Account Manager proceeds to communicate to the DBMS asking for the insertion of the Student's data in the DB. It proceeds to inform the Notification Manager asking it to send an email of confirmation for the registration. It returns the positive outcome of the operation to the WebApp, which in turn will show a confirmation message to the Student.

2.5.2 Educator sign-up to the platform
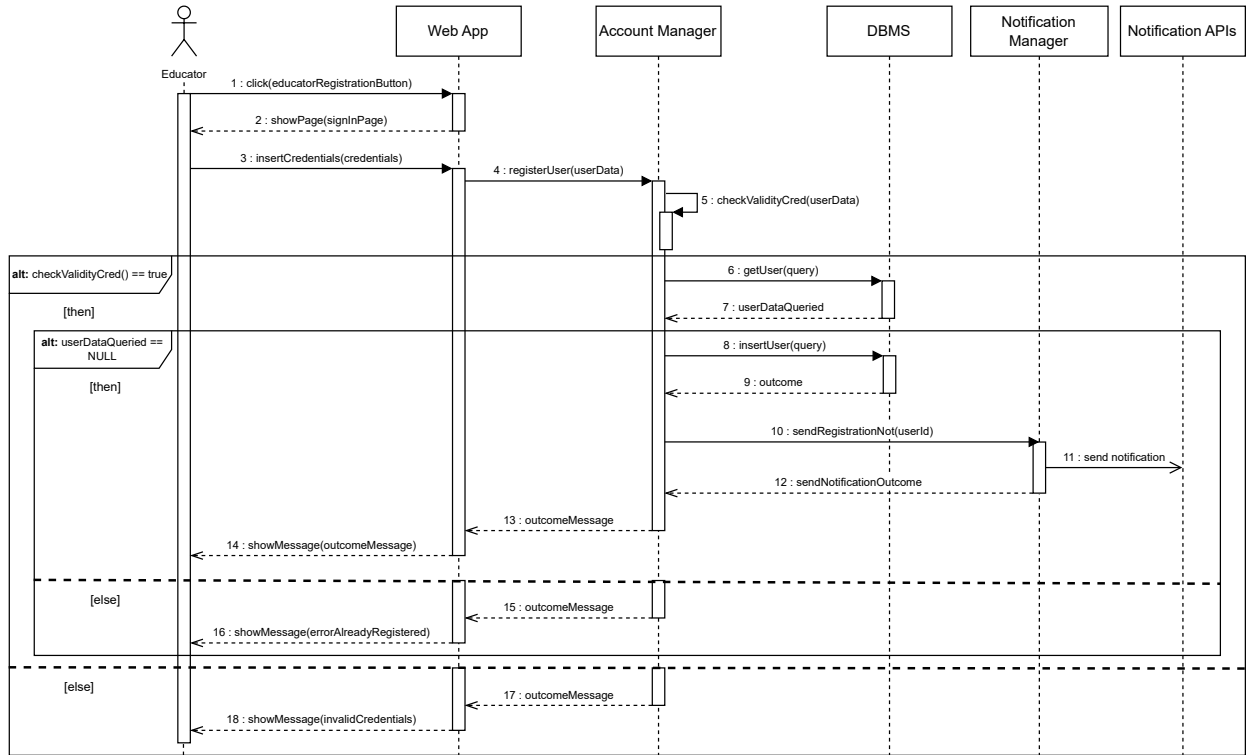
Figure 6: Educator sign-up sequence diagram

The above diagram represents an Educator's signing up process. The process is pratically identical to the Student's one, with the exception of the data that the Educator is requested to enter in the form, such as their name, surname username, school in which they teach, istitutional email, password, . . .

2.5.3 Educator creates a new tournament
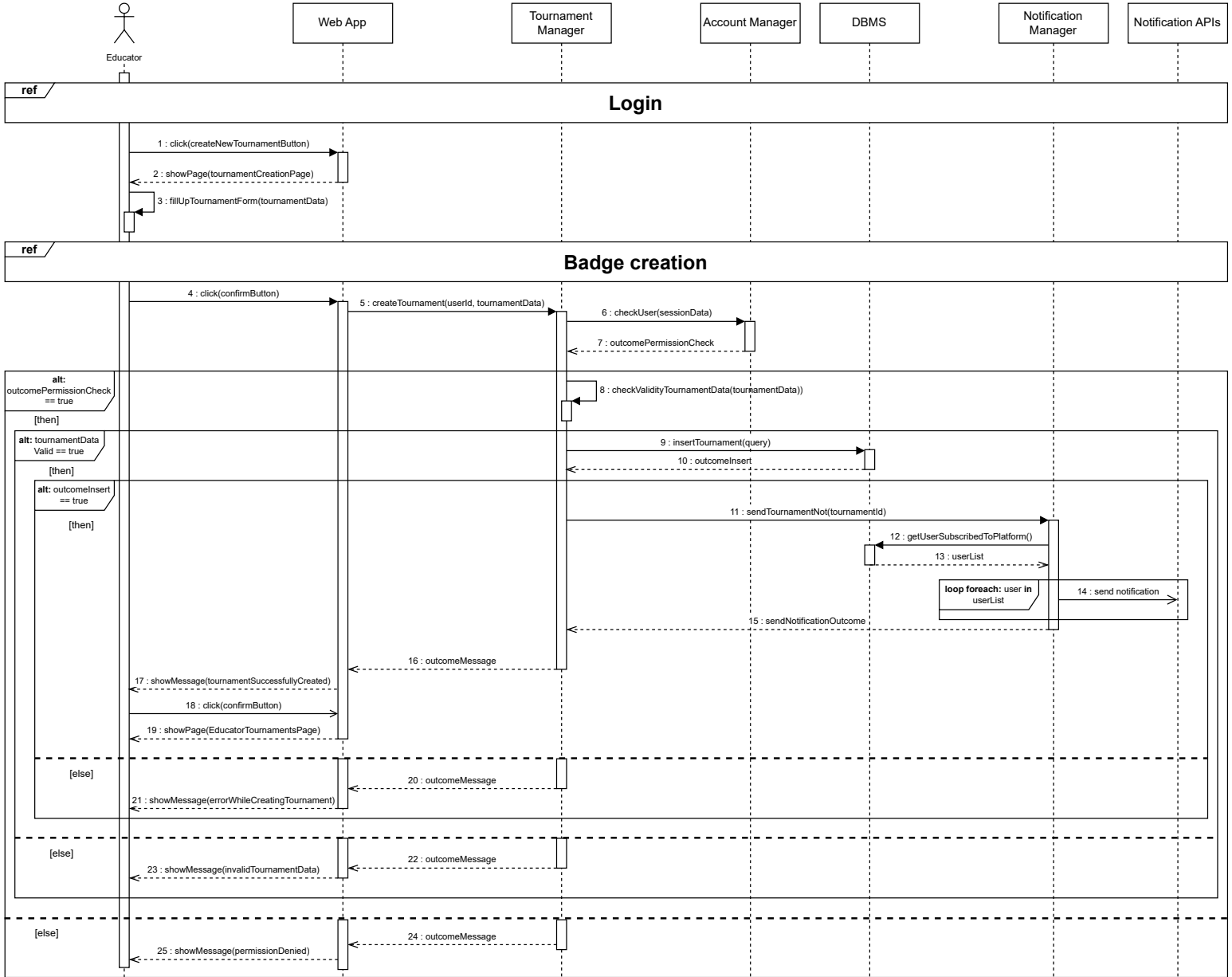
Figure 7: Tournament creation sequence diagram

The above diagram represents the creation of a tournament process. In order to start this process the Educator must have completed the login procedure, as shown in 4 figure. The Educator has to access the riepilogative page of their tournaments (this part of the process was omitted for simplicity) and here he has to click on the "Create new Tournament" button (as shown in 21). The WebApp will show the Tournament creation page (Figure 22) in which there is a form that the Educator has to fill. At this point the Educator can create new badges to insert in the tournament. This process is displayed in the Sequence Diagram 2.5.4 in figure 8.

The Educator then clicks on the "Confirm creation" button as shown in figure 22, in this way the tournament's data will be sent to the Tournament Manager which will check if the user has the permission to use the API by contacting the Account Manager. It will then check if the tournament's inserted data are valid. If all the checks were successful the Tournament Manager proceeds to contact the DBMS which will insert the new tournament in the DB.

If everything went well, the Tournament Manager contacts the Notification Manager which will send a tournament's creation notification to all the Students subscribed to the platform. Finally the Tournament Manager will return to the WebApp a message of success, which in turn will return it to the Educator who will be redirected to the riepilogative page of their tournaments.

There are several cases in which this process could go wrong. In order we have:

- **User does not have permissions:** In case the logged User is not an Educator, or isn't logged. The Tournament Manager will return an error message to the WebApp, which will display it to the User. The user will be redirected to the login page (omitted for simplicity).

- **Tournament's data are invalid** In this case after the check, the Tournament Manager component returns an error message to the WebApp, stating that the data inserted is not valid. The message will then be shown to the Educator by the WebApp.

- **Tournament's DB insertion went wrong:** In this case no notification will be sent to the Students and the Tournament Manager will return to the WebApp an error message stating that there was an error while creating the tournament. The message will then be shown by the WebApp to the Educator.
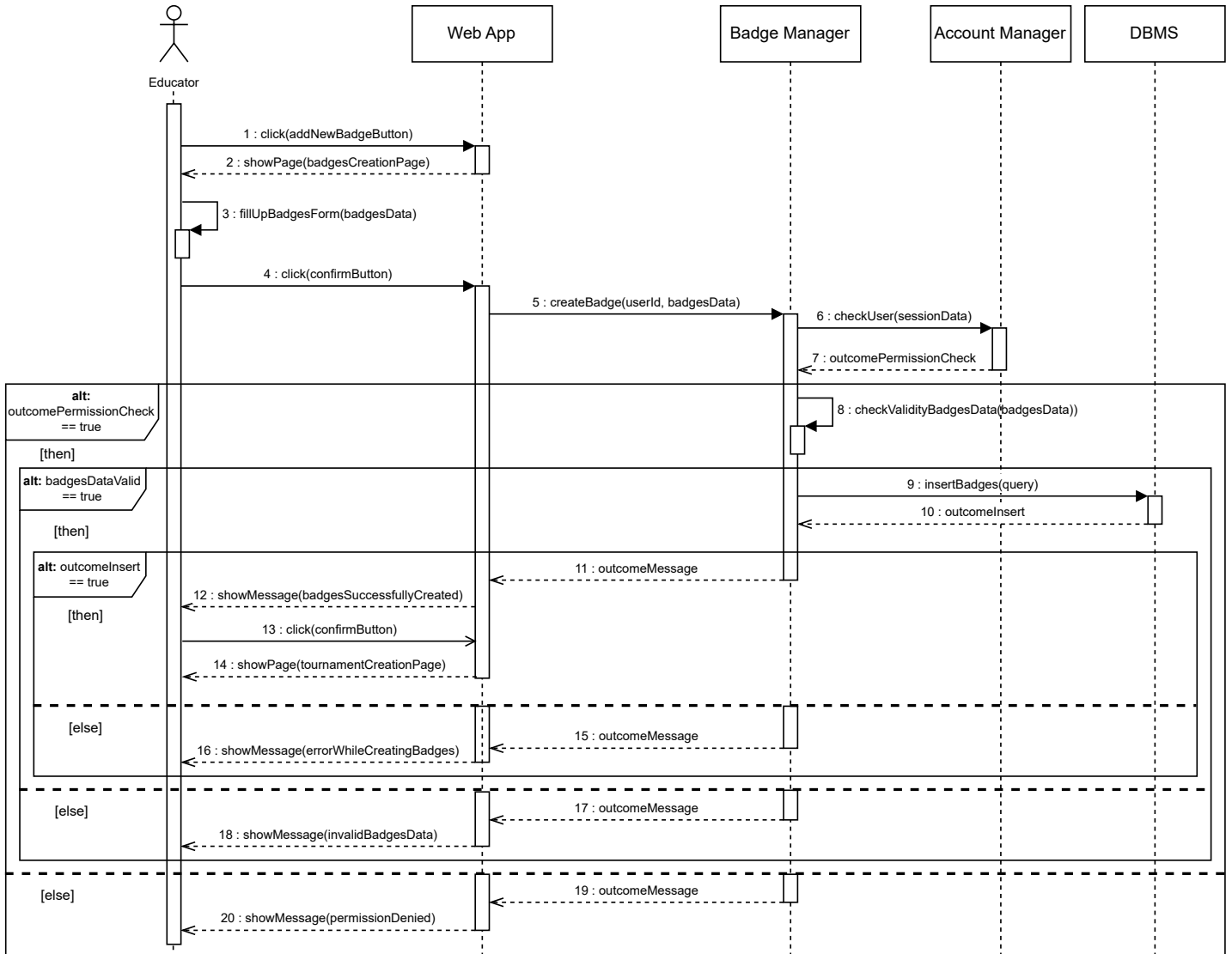
2.5.4   Educator creates new badges

Figure 8: Badges creation sequence diagram

The diagram above represents the process of a badge's creation. In order to start this process the Educator must start creating a new tournament as first step and as shown in figure 7.

To create new badges during a tournament's creation the Educator clicks on

the "Add new badge" button (as shown in figure 22). The WebApp will show a form that the Educator has to fill with the badge's data.

After clicking on the "Confirm creation" button (as shown in figure 23) the WebApp contacts the Badge Manager component in order to start the badge's creation.

The Badge Manager communicates with the Account Manager, which will check whether the logged User has permission to create a new badge. The Badge Manager will also check the validity of the data inserted by the Educator in the form.

If all the checks go well, the Badge Manager contacts the DBMS in order to insert the badge's data into the DB. If the insertion goes well, the Badge Manager sends back to the WebApp a confirmation message stating that the creation of the badge was successful and the Educator is led back to the page dedicated to the tournaments' creation.

The process can go wrong in several situations. In order we have:

- **User does not have permissions:** In case the logged User is not an Educator, or isn't correctly logged, the Badge Manager will return an error message to the WebApp. The WebApp will then display it to the User. Finally the user will be redirected to the login page (omitted for simplicity).

- **Badge's data are invalid:** In this case, the Badge Manager component returns an error message to the WebApp stating that the inserted data is not valid. The message will then be shown to the Educator by the WebApp.

- **Badge's DB insertion went wrong:** In this case the Badge Manager will return to the WebApp an error message stating that there was an error while creating the badge. The message will then be shown by the WebApp to the Educator.

As said at the beginning the Educator can also delete badges just by pressing the "Delete" button as shown in Figure 22.

2.5.5 Educator deletes and/or updates a badge
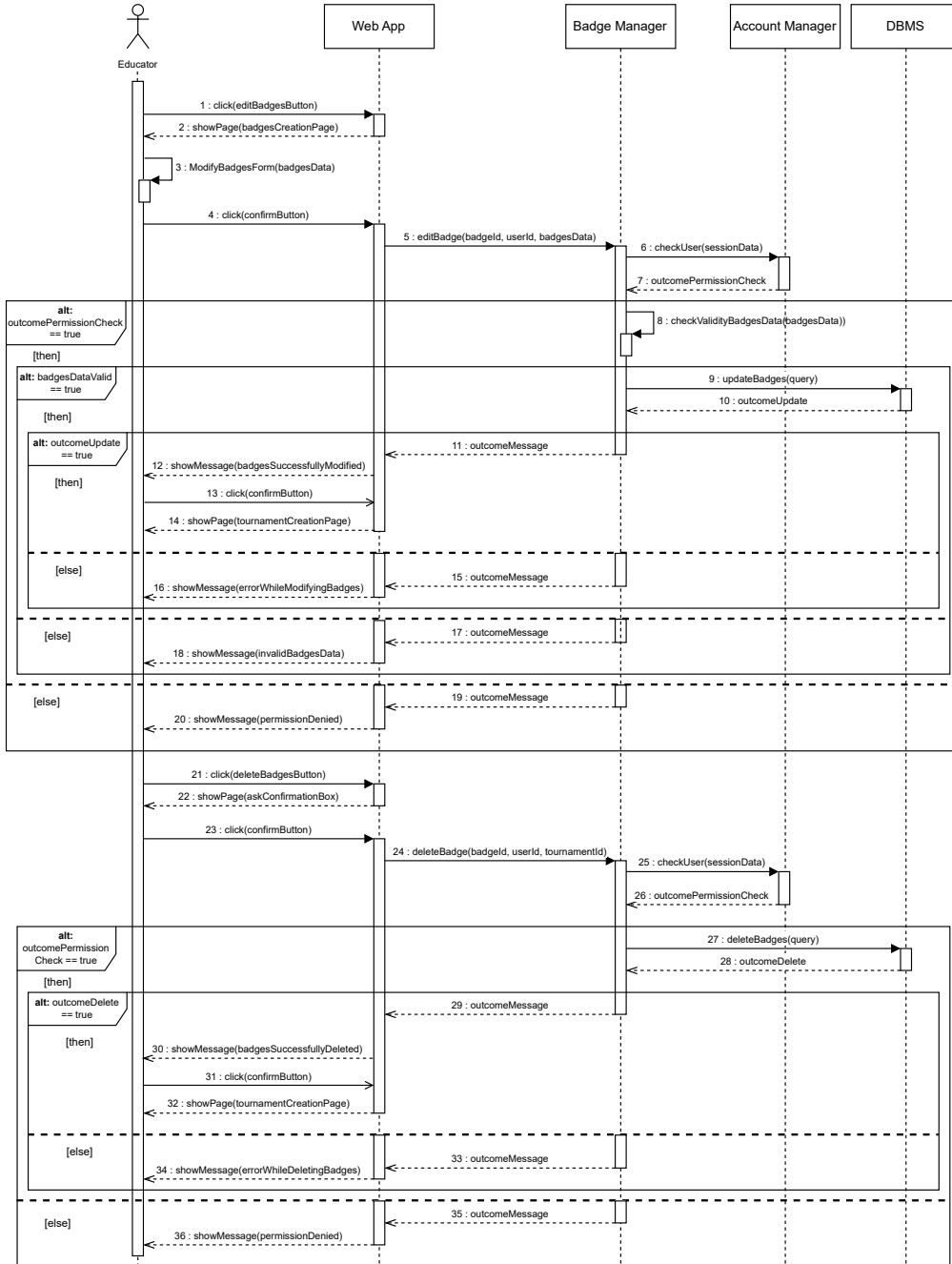

Figure 9: Badges deletion/update sequence diagram

The diagram above represents the process of badges' modification and/or deletion. In order to start the process the Educator must have completed the login procedure first, as shown in figure 4.

The Educator, during the creation of a tournament, has the possibility of modifying and/or deleting badges that he has precedently created during the same tournament creation procedure. In order to modify a badge, the Educator can click on the "Edit" button (as shown in figure 22). The WebApp will show a form containing all the informations related to the badge that the Educator wants to modify. At this point the Educator can modify the badge's data to their liking. After the modification the Educator has press the "Confirm creation" button (as shown in figure 23) to confirm the changes.

The WebApp contacts the Badge Manager in order to start the badge's edit. The Badge Manager communicates with the Account Manager which will check whether the logged User has permission to modify the badge. The Badge Manager will also check the validity of the data inserted by the Educator in the form.

If all the checks go well, the Badge Manager contacts the DBMS in order to update the badge's data stored in the DB. If the update goes well, the Badge Manager sends to the WebApp a confirmation message stating that the modification of the badge was successful and the Educator is led back to the page dedicated to the tournaments' creation.

The process can go wrong in several situations. In order we have:

- **User does not have permissions:** In case the logged User is not an Educator. The Badge Manager will return an error message to the WebApp, which will be displayed it to the User. Finally the user will be redirected to the login page (omitted for simplicity).

- **Badge's data are invalid:** In this case the Badge Manager returns an error message to the WebApp stating that the inserted data were not valid. The message will then be shown to the Educator by the WebApp.

- **Badge's DB insertion went wrong:** In this case the Badge Manager will return to the WebApp an error message stating that there was an error while updating the badge. The message will then be shown by the WebApp to the Educator.
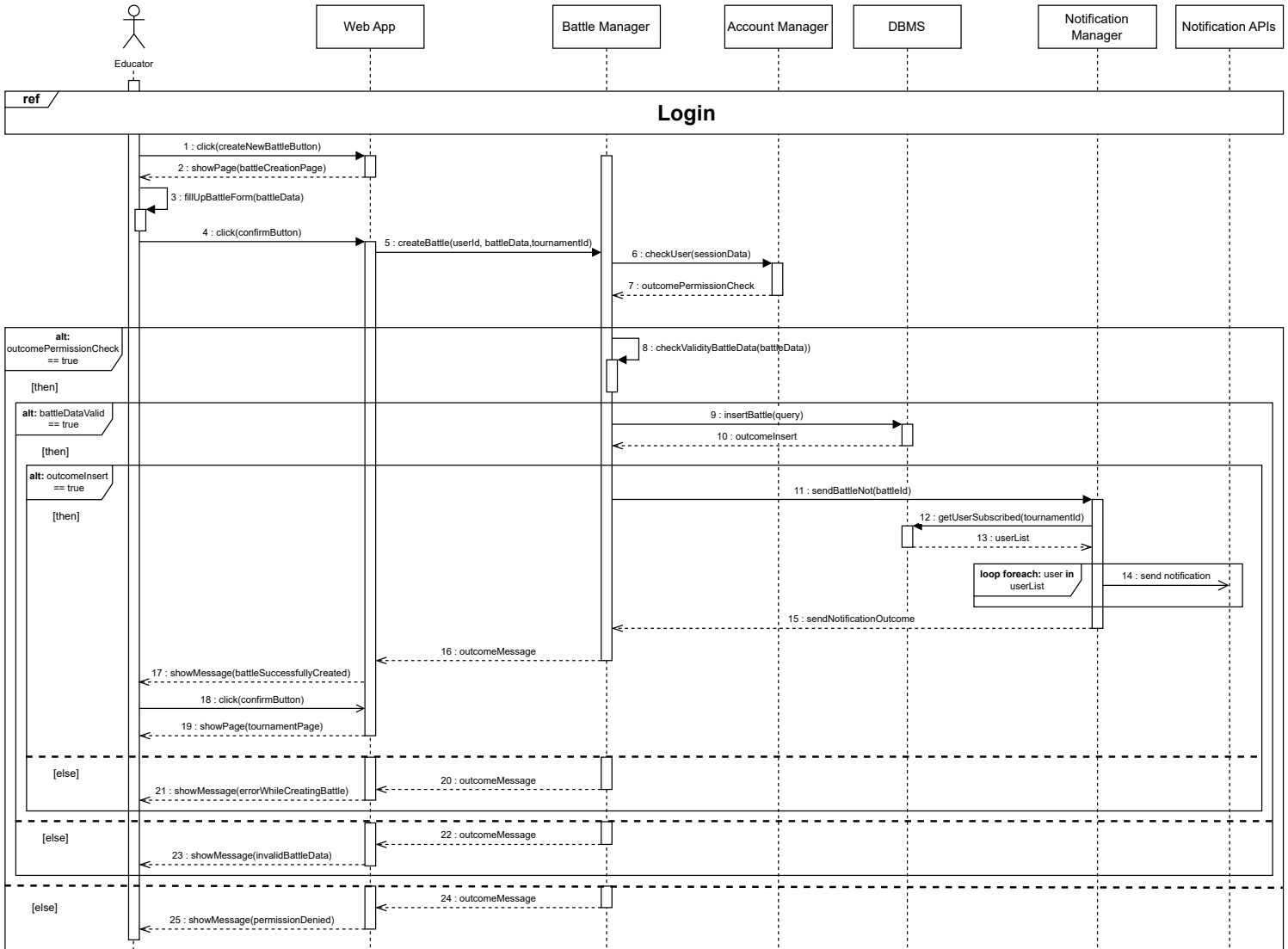
2.5.6 Educator creates a new battle

Figure 10: Battle creation sequence diagram

The above diagram represents the process of creating a new battle within a tournament. In order to start the process, an Educator must have completed the login procedure first, as shown in figure 4.

The Educator accesses the CKB platform WebApp Homepage and subsequently the page of one of the tournaments were they have permission to create a bat-

tle into (omitted for simplicity). They then click on the "Create new Battle" button as shown in figure 24.

The WebApp will show a page which contains a form that the Educator can fill with the battle's data, such as its name, maximum and minumum number of member per group, registration and submission deadlines, programming language accepted, ...

Once the Educator presses the "Confirm" button, the battle's data will be sent to the Battle Manager.
The Battle Manager will contact the Account Manager to verify, through the session's data, whether the logged User has permission to create a new battle within that tournament. If that's the case then the Battle Manager will check the validity of the battle's data inserted, i.e. checking whether one or more data don't respect some standards.

If the check succeeds then the Battle Manager will try to insert the battle in the Tournament DB by contacting the appropriate DBMS.
If the insertion succeeds then the Battle Manager will communicate with the Notification Manager, which will send the notification of the battle's creation to all Students subscribed to the tournament the newly created battle is contained into. The Battle Manager will return a message of success to the Educator through the WebApp.

There are several cases in which this process won't succeed. In order we have:

- **Educator does not have permissions:** In case the Educator is not the tournament's creator or has not been granted access to it from another collegue, the Battle Manager will return an error message stating that the the procedure was denied. The message will be shown by the WebApp to the Educator and finally the User will be redirected to the login page (omitted for simplicity).

- **Battle's data are invalid:** In this case, after the check, the Battle Manager returns an error message to the WebApp stating that the inserted data are not valid. The message will then be shown to the Educator by the WebApp.

- **Battle's DB insertion went wrong:** In this case no notification will be sent to the Students and the Battle Manager will return to the WebApp an error message stating that there was an error while creating the battle. The message will then be shown by the WebApp to the Educator.
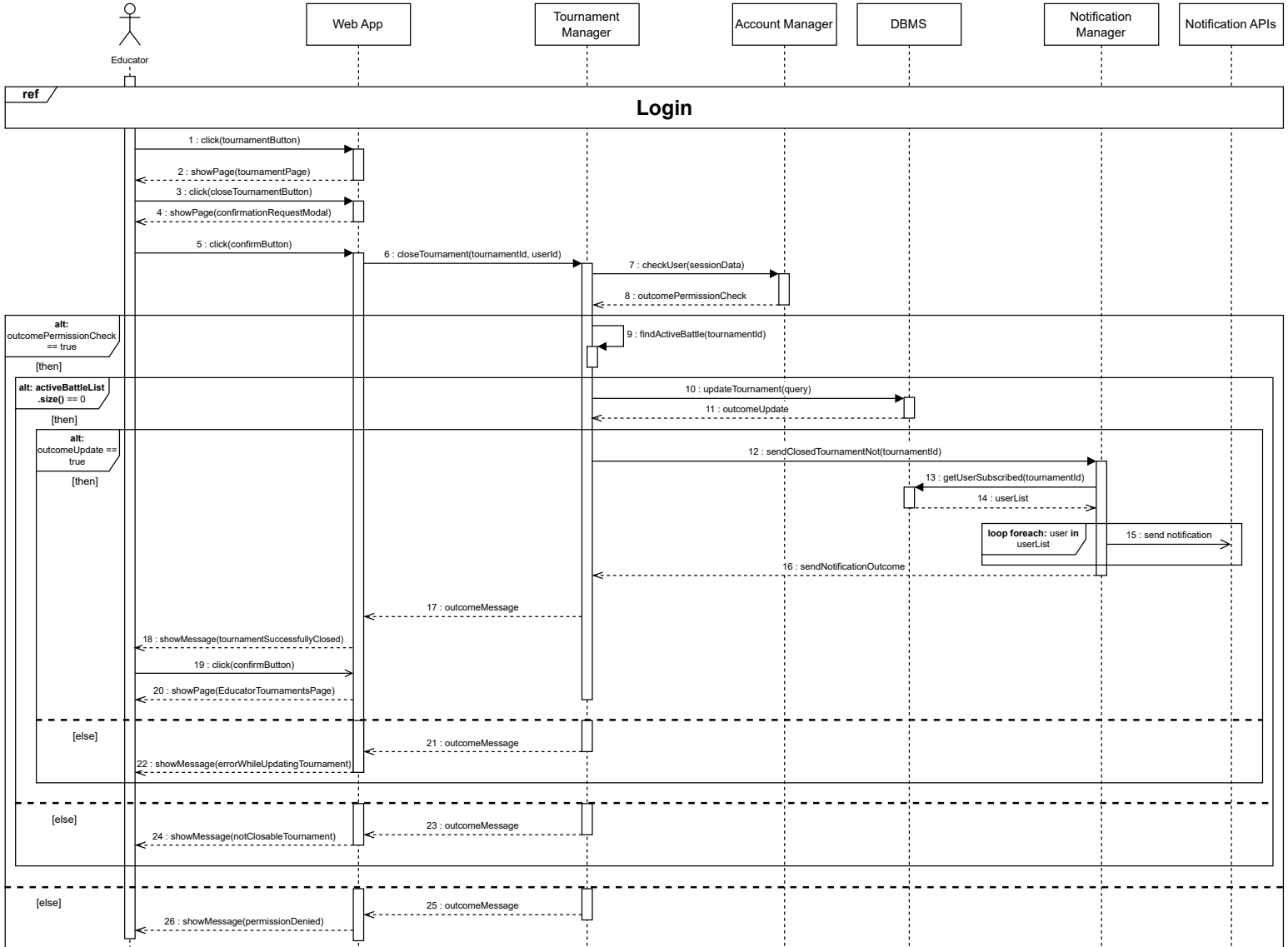
2.5.7 Educator closes a tournament

Figure 11: Tournament closure sequence diagram

The above diagram represents the process of tournament closing.
In order to start this process the Educator must have completed the login procedure, as shown in 4 figure. The Educator has to access the riepilogative page of their tournaments (this part of the process was omitted for simplicity) and

here they have to click on a tournament in order to access it. Subsequently they have to click the "Close tournament" button in order to close it (as shown in 21). The WebApp will show a confirmation modal to the Educator, who can click the "Confirm" button starting in this way the tournament's closure.

At this point the WebApp contacts the Tournament Manager which will rely on the Account Manager to check if the user has permission to use the APIs. The Tournament Manager will check if there is at least one active battle within the tournament. If that is not the case then it will contact the DBMS in order to update the tournament's information and close it.

If everything goes well, the Tournament Manager contacts the Notification Manager which will send a notification for the closure of the tournament to all the Students subscribed to that tournament. Finally the Tournament Manager returns to the WebApp a message of success which in turn will return it to the Educator, who will be redirected to the riepilogative page of their tournaments.

There are several cases in which this process may go wrong. In order we have:

- **User does not have permissions:** In case the logged User is not an Educator. The Tournament Manager will return an error message to the WebApp, which will display it to the User. Finally the user will be redirected to the login page (omitted for simplicity).

- **Tournament contains active battles** In this case after the check, the Tournament Manager returns an error message to the WebApp stating that the closure of the tournament has been denied since there is still at least one battle active within it. The message will be shown then to the Educator by the WebApp.

- **Tournament's DB insertion went wrong:** In this case no notification will be sent to the Students, and the Tournament Manager component will return to the WebApp an error message stating that there was an error while updating the tournament. The message will then be shown by the WebApp to the Educator.
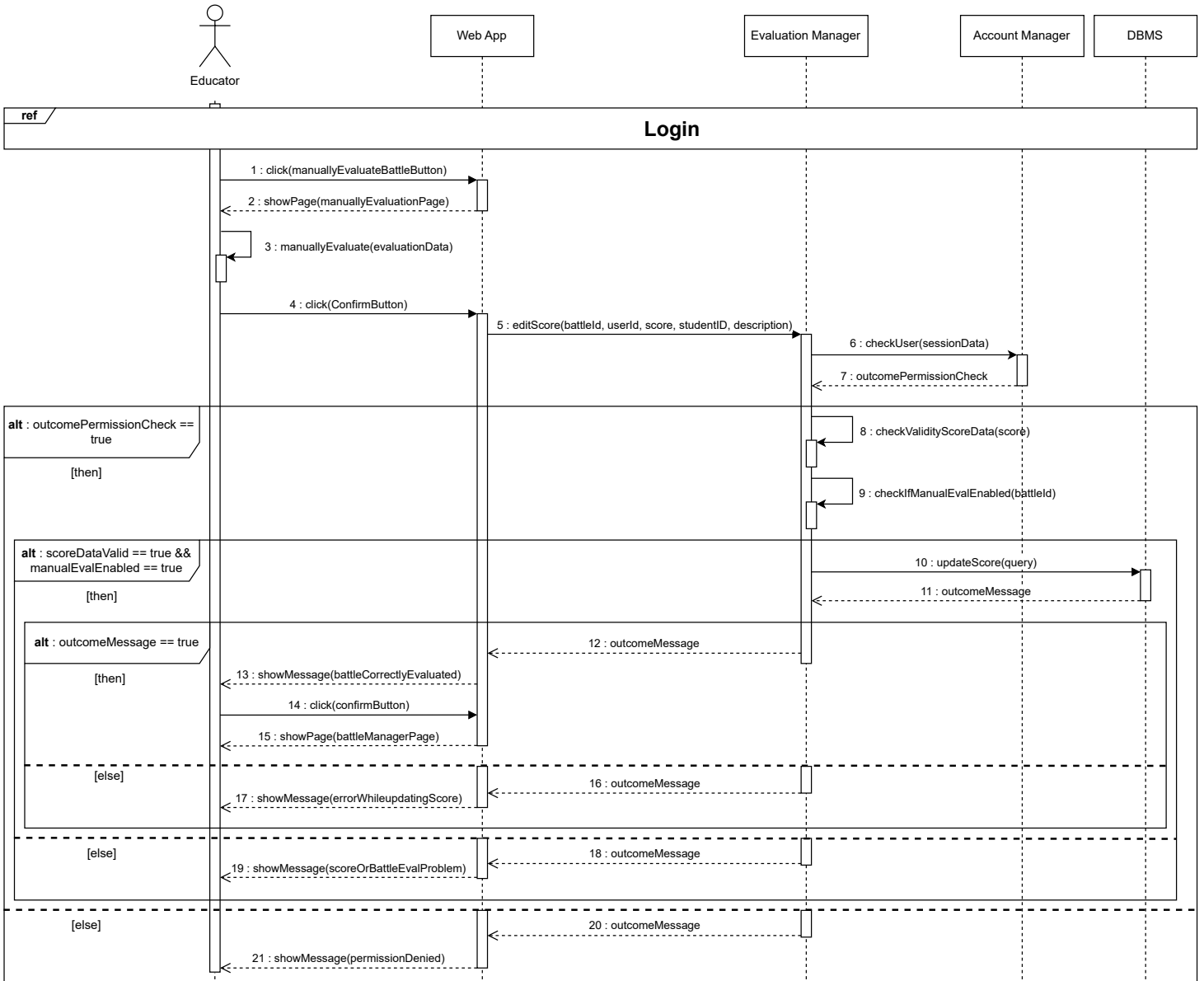
2.5.8   Educator evaluates a battle's results

Figure 12: Manual solution's evaluation sequence diagram

The above diagram represents the manual evaluation of a battle. In order to start this process the Educator must have completed the login procedure first, as shown in 4 figure.
The Educator has to access their tournaments' page, then has to click on the tournament containing the battle that needs a manual evaluation (omitted for simplicity).
The Educator, after having selected a battle, clicks on the "Manually Evaluate Battle" button.

The WebApp will show a page which contains the groups' current scores and their last uploaded solutions. The Educator, after having inspected the groups' solutions, can update the groups' score and, if he wants, can write some comments.

Once the Educator presses the "Confirm" button, the score's data will be sent to the Evaluation Manager.
The Evaluation Manager will contact the Account Manager, to verify, through session's data, whether the logged User has permission to manually evaluate a battle.
If that's the case then the Evaluation Manager will check the validity of the inserted score data.
If the check succeeds then the Evaluation Manager will try to update the score in the Tournament DB by contacting the Tournament DBMS.
If the update succeeds, the Evaluation Manager will return a message of success to the Educator through the WebApp.

There are several cases in which this process may go wrong. In order we have:

- **User does not have permissions:** In case the User is not and Educator or is not the battle's creator, the Evaluation Manager will return an error message stating that the the procedure was denied. The message will be shown by the WebApp to the Educator, and finally the User will be redirected to the login page (omitted for simplicity).

- **Score's data are invalid:** In this case, after the check, the Evaluation Manager returns an error message to the WebApp stating that the inserted data were not valid. The message will then be shown to the Educator by the WebApp.

- **Battle is not enabled to be manually evaluated:** In this case after the check, the Evaluation Manager returns an error message to the WebApp stating that the battle was not enabled to be manually evaluated.

- **Score's DB insertion went wrong:** In this case no update will be done to the previous score and the Evaluation Manager returns to the WebApp an error message stating that there was an error while updating the score. The message will then be shown by the WebApp to the Educator.

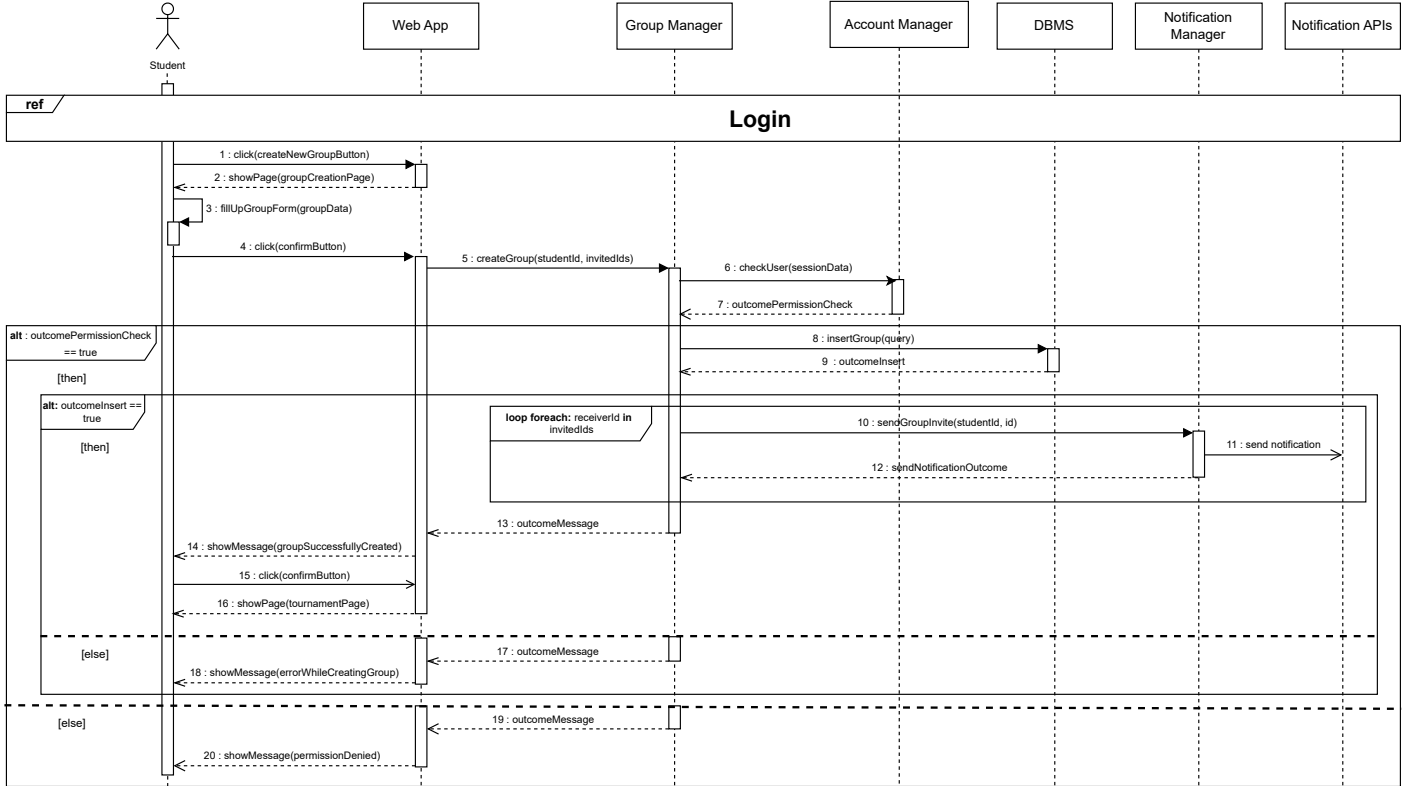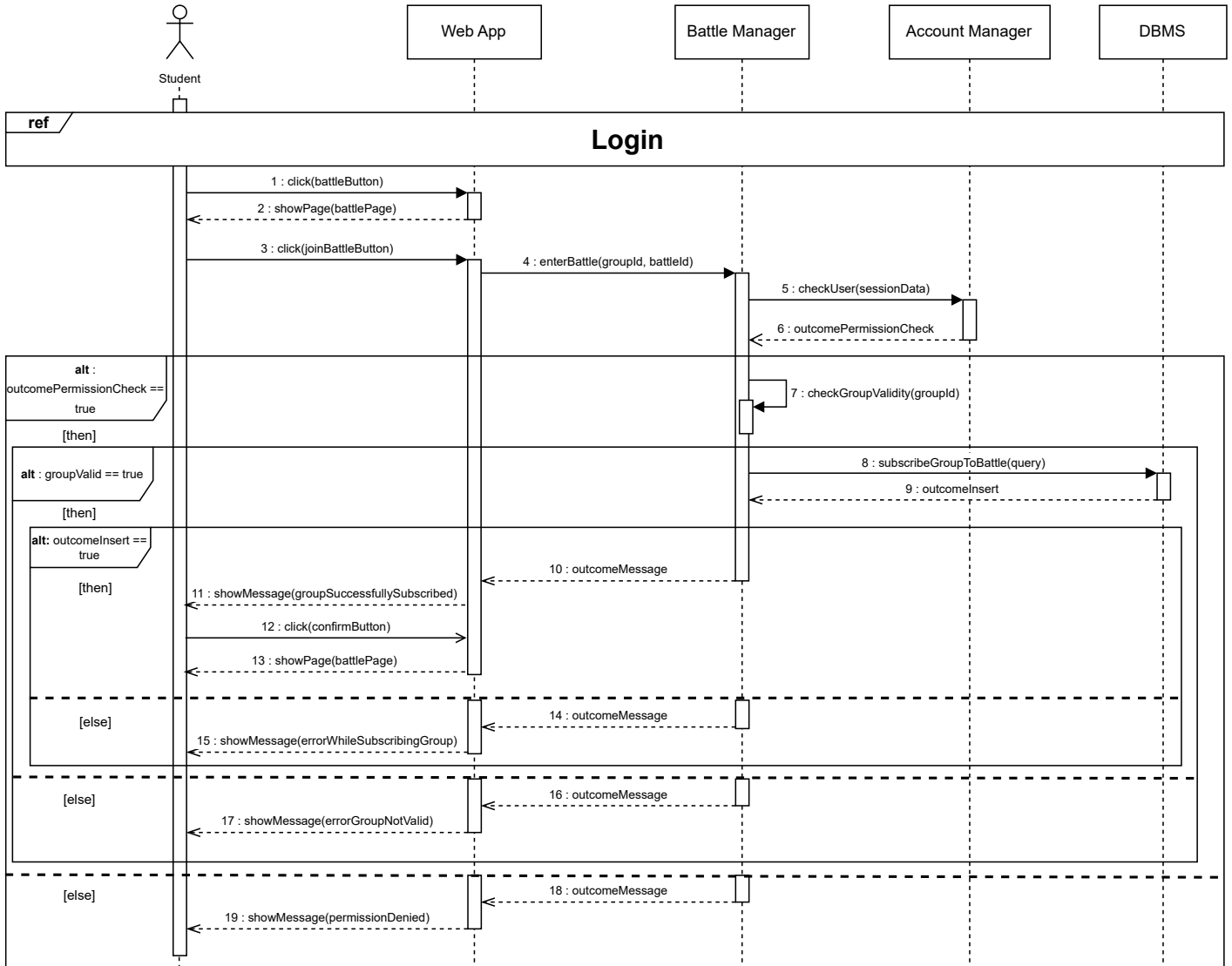## 2.5.9   Student forms a group

Figure 13: Group creation sequence diagram

The above diagram represents the formation of a group. In order to start this process the Student must have completed the login procedure first, as shown in 4 figure.

The Student has to access the tournaments' page, access tournament's page where he wants to create the group then click on the "Create new group" button (as shown in figure 27).

The WebApp then shows a page containing a fillable form where the Student can decide the Students to invite in the group. After choosing the Students to invite, the Student clicks the "Confirm creation" Button (as shown in figure 28) and the group's data will be sent to the Group Manager.

The Group Manager contacts the Account Manager to verify, through the session's data, whether the logged User has the permissions to create a group. If that's the case then the Group Manager will effectively create the group by

contacting the DBMS.

If the check succeeds then the Group Manager will try to insert the group in the Group DB, by contacting the Group DBMS. If the insertion succeeds then the Group Manager interacts with the Notification Manager, which in turn will send a notification to all the invited Students.

At this point the Group Manager returns to the WebApp a message of success. That message will be showed to the Student by the WebApp itself.

There are several cases in which this process may go wrong. In order we have:

- **User does not have permissions:** In case the User is not a Student or isn't subscribed to the tournament the Group Manager will return an error message stating that the the procedure was denied. The message will be shown by the WebApp to the User and finally they will be redirected to their profile page (omitted for simplicity).

- **Group's DB insertion went wrong** In this case no insertion will be done, and the Group Manager will return to the WebApp an error message, stating that there was an error while creating the group. The message will then be shown by the WebApp to the Student.
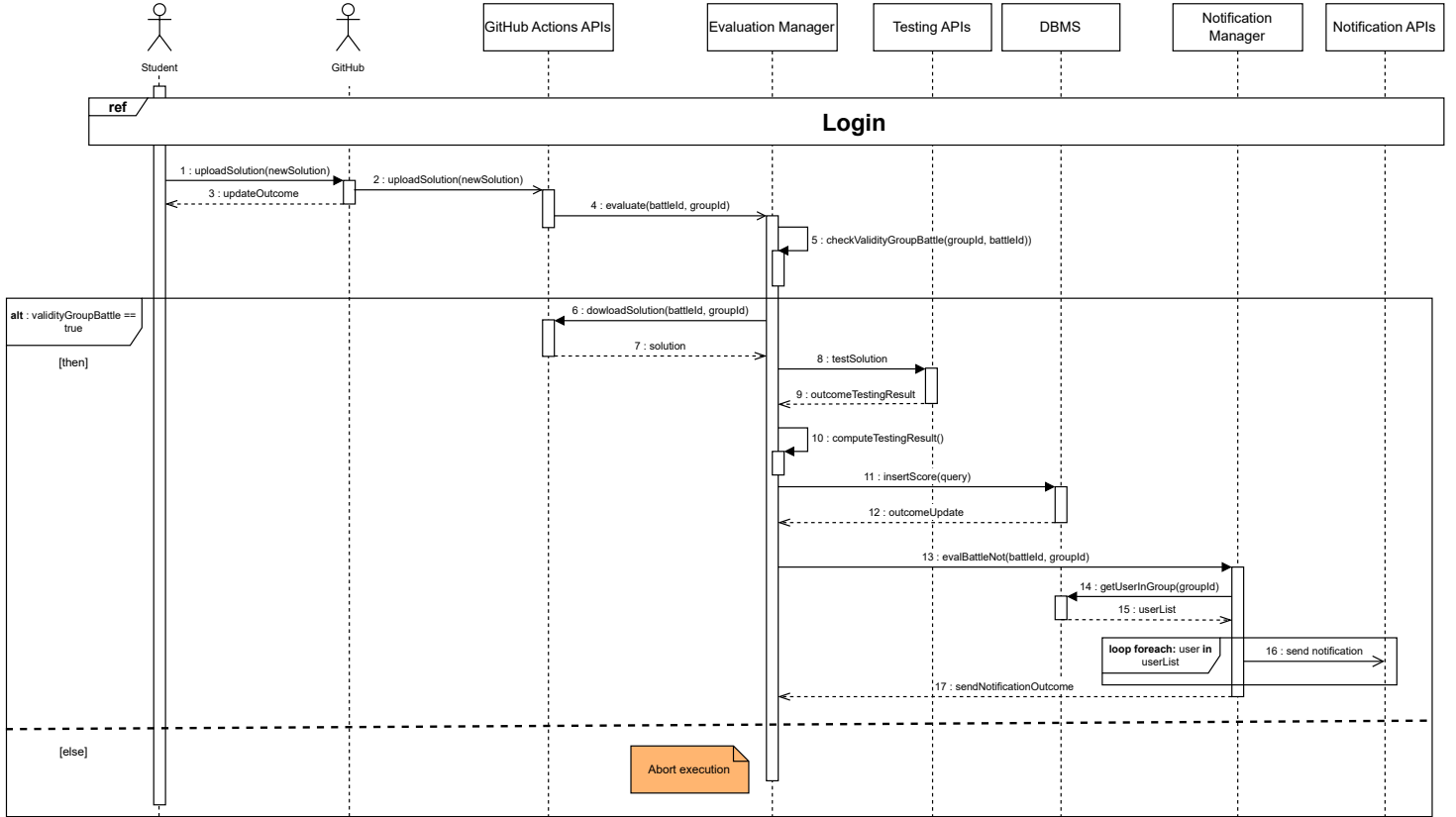
2.5.10   Student joins a battle

Figure 14: Battle joining sequence diagram

The above diagram represents the joining of a battle. In order to start this process the Student must have completed the login procedure first (as shown in 4 figure) and the group creation procedure (as shown in 13 figure). The Student has to access a tournament, access the page of the battle that he wants to join then click on the "Enroll" button (as shown in figure 27).

The Battle Manager contacts the Account Manager to verify, through the session's data, whether the logged User has the permissions to join the battle. If that's the case then the Battle Manager will check the validity of the group, i.e. checking whether the group violates some dimension constraints.

If the check succeeds then the Battle Manager will try to subscribe the group in the battle, by contacting the DBMS. If the insertion succeeds then the Battle Manager will return a message of success to the Student through the WebApp.

There are several cases in which this process may go wrong. In order we have:

- **User does not have permissions:** In case the User in not a Student or has not permission to join a battle, the Battle Manager will return an error message stating that the the procedure was denied. The message will be shown by the WebApp to the Student and finally the User will be redirected to the profile page (omitted for simplicity).

- **Group's data are invalid** In this case after the check, the Battle Manager returns an error message to the WebApp stating that the group don't respect the battle's rules. The message will then be shown to the Student by the WebApp.

- **Group's DB insertion went wrong:** In this case no subscription will be done and the Battle Manager will return to the WebApp an error message stating that there was an error while inserting the group. The message will then be shown by the WebApp to the Student.

2.5.11 Student uploads a new solution

Figure 15: Upload of a new solution sequence diagram

The above diagram represents the uploading of a new battle's solution. In order to start this process the Student must have completed the login procedure first (as shown in figure 4). The Student uploads a new solution on the GitHub platform, that will activate the GitHub Actions APIs. The GitHub Actions APIs will communicate to the Evaluation Manager that a new solution has been uploaded and needs to be evaluated.

The Evaluation Manager will check the validity of the group and battle, i.e. checking whether the group is actually subscribed to that battle.

If the check succeeds then the Evaluation Manager will download the solution through the GitHub Actions APIs. The Evaluation Manager tests the deadline, omitted because part of a subsequent diagram (shown in figure 16). The Evaluation Manager tests the solution by passing it to the Testing APIs. Af-

ter receiving the test's result, the Evaluation Manager will try to update the group's score in the battle by interacting with the DBMS.

Finally the Evaluation Manager will interact with the Notification Manager in order to send a notification to all the group's member that their solution has been evaluated.

There is one case in which this process won't succeed:

- **Group isn't subscribed to the battle:** In this case the Evaluation Manager will abort the execution, nothing changes. (Shown for simplicity with a comment)

2.5.12 Student uploads a solution after submission deadline
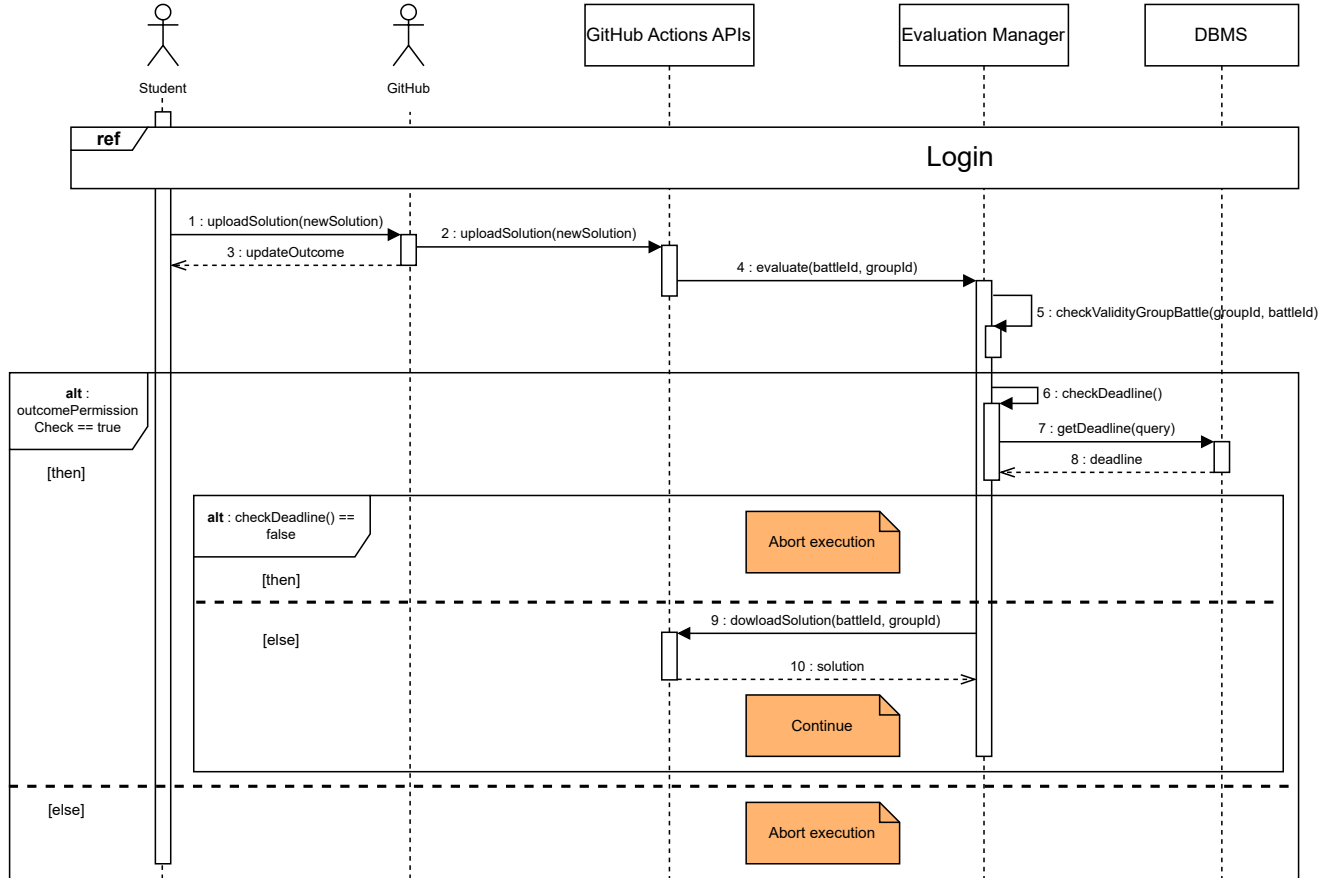
Figure 16: Upload of a new solution after deadline sequence diagram

The above diagram represents the uploading of a new solution for a battle by a Student, after the expiration of the battle's deadline. If the check goes well, then the Evaluation Manager component will download the solution by interacting with the GitHub APIs and it will have the same behaviour as in the previous diagram.

If the check highlights some problems, i.e. if the battle's deadline has already expired, then the execution is aborted.

2.5.13 Student visualises his tournament's results and badges
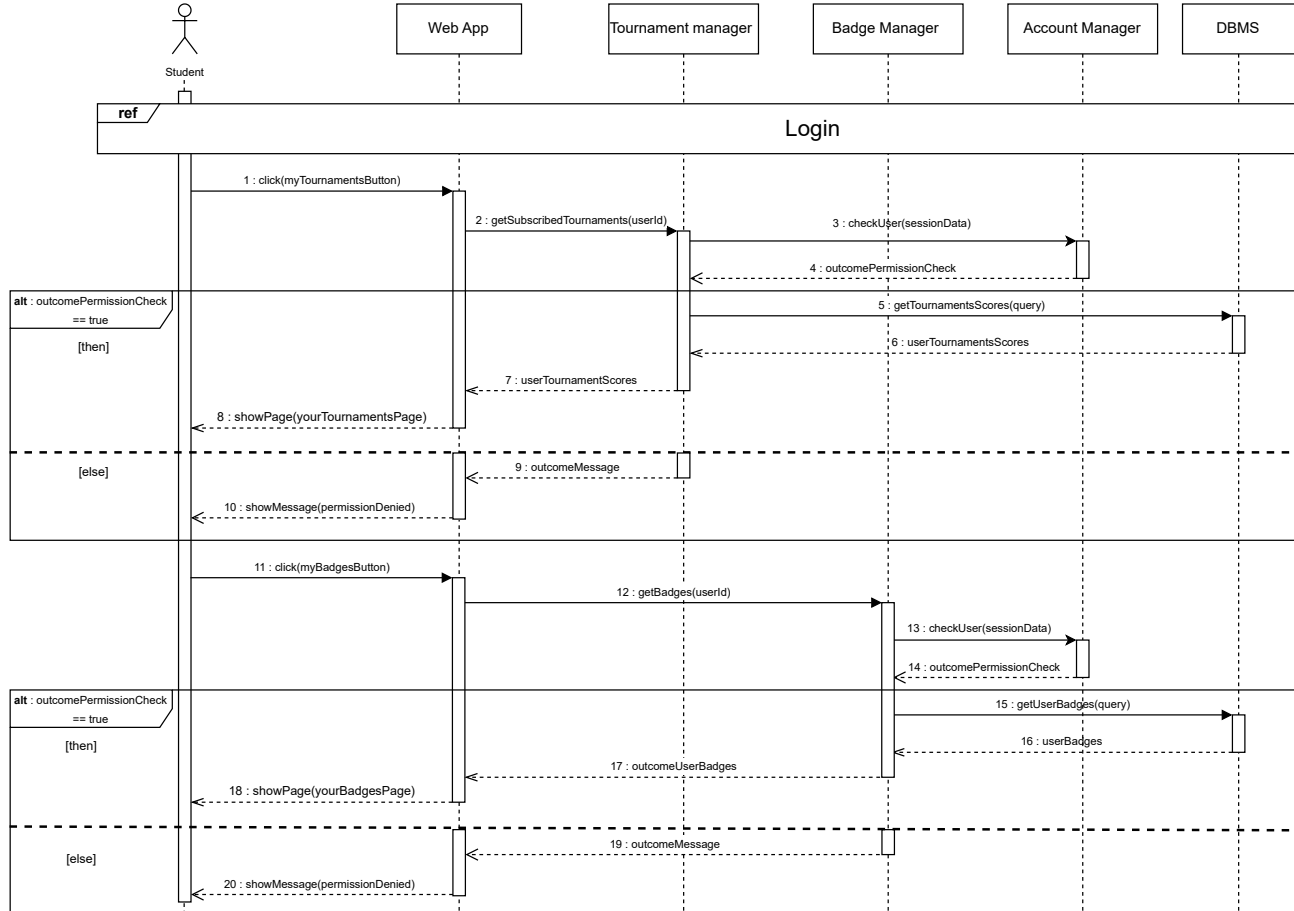
Figure 17: Badges and result visualization sequence diagram

The above diagram represents the visualization of a Student's results and badges. In order to start this process the Student must have completed the login procedure first (as shown in fig. 4). The Student has to access their profile page (omitted for simplicity), then has to click on the "My Tournaments" button (as shown in figure 26).

The WebApp will send the user's data to the Tournament Manager.
The Tournament Manager contacts the Account Manager to verify, through the session's data, whether the logged User has permission to see the tournaments'

result. If that's the case then the Tournament Manager will ask the DBMS to get all the scores obtained by the Student in the tournaments that they have partecipated to.

After receiving the list of scores, the Tournament Manager component will return the page with the tournaments' result to the Student through the WebApp.

The Student now wants to see their badges, so they click on the "My Badges" button (as shown in figure 26).
The WebApp sends the user's data to the Badge Manager.

The Badge Manager contacts the Account Manager, to verify, through the session's data, whether the logged User has the permissions to see the badges. If that's the case then the Badge Manager will ask the DBMS all the badges obtained by the Student. After receiving the badges, the Badge Manager will return the page with the badges to the Student through the WebApp.

There is a case in which this process may go wrong. In order we have:

- **User does not have permissions:** In case the User isn't a Student or doesn't have permission to see tournaments' result, the Tournament Manager will return an error message stating that the the procedure was denied. The message will be shown by the WebApp to the Student and finally the User will be redirected to his profile page (omitted for simplicity).

2.6 Selected architectural styles and patterns

2.6.1 Four-layered architecture:

We chose this architecture for many reasons, mainly because:

- **Maintainability:** By having four different layers, with separated logic and data, once the platform's structure is fully defined each layer's interior logic will be fully defined. By having separated logic and data it will be easier in the future to access and solve possible problems that can occur in different layers.

- **Security:** By separating the web, application and data servers in different internet partitions divided by firewalls we create multiple DMZs, resulting in a more secure architecture. Before accessing the data layer, a malicious user would need to surpass three different firewalls.

2.6.2 MicroServices architecture:

The MicroServices architecture is what we chose since the platform will be developed as a collection of services. MicroServices are needed for a fast, modern and reliable system, since what we want to achieve here is dividing each task in multiple, smaller ones, that are easier to work with. We chose the MicroServices architecture to ensure these properties in the system:

- **Load Distribution:** The presence of multiple application servers and databases assures us that the platform will be fast and reliable in all its elaborations. If we didn't have load balancing or any sort of redundancy we would have found ourselves with multiple single nodes having to serve multiple requests, causing overloading and possible deadly failures.

- **Scalability:** The result of load distribution ensures that redundancy is provided only for the most important parts of the platform. This ensures maximum scalability with minimal cost.

2.6.3 REST architecture:

The REST architecture is a set of rules to follow while building a web application, to ensure that the web application being built is stateless. This means that the server does not care about the client's status information, but those informations are dealt with by the client.

- **Cacheability:** For a web application to be RESTFul, it needs to be cacheable. Having a web cache makes the web application much faster when handling requests, since the application servers will not be accessed

each time a request is made. Instead, cache proxies present in the network will parse the requests and answer for them, if the data they have is not too old.

- **Uniform interface:** Having uniform interfaces thoughout all its users, the web application ensures that it can be accessed from any browser, downloaded on any operating system. This is an important constraint, as any user can access and use the CKB platform.

2.7 Other design decisions

2.7.1 Relational Database:

We chose to implement our databases in a relational way, since relational databases are the go-to choice when implementing a web application. Relational databases are needed when we want to ensure that data will not be lost in time and we want data to be kept sure. Since we want to implement our web application following the MicroServices architectural style, we chose to separate our data in three different databases. We have a database dedicated only to the users' data, containing all the different users. This database is accessed on login, registration and when the session is created. We have a database dedicated to the tournaments' data, containing all battles, badges and tournaments. This database is accessed each time a request is made that needs to access or modify some data related to tournaments. We have a database dedicated to Students' groups, accessed each time a new group is created. This database will also contain all the useful information regarding groups' work on their repositories, gathered via the GitHub APIs. Each database is developed using the PostgreSQL language, since it has a better trigger architecture than MySQL or MariaDB.

3   User Interface Design
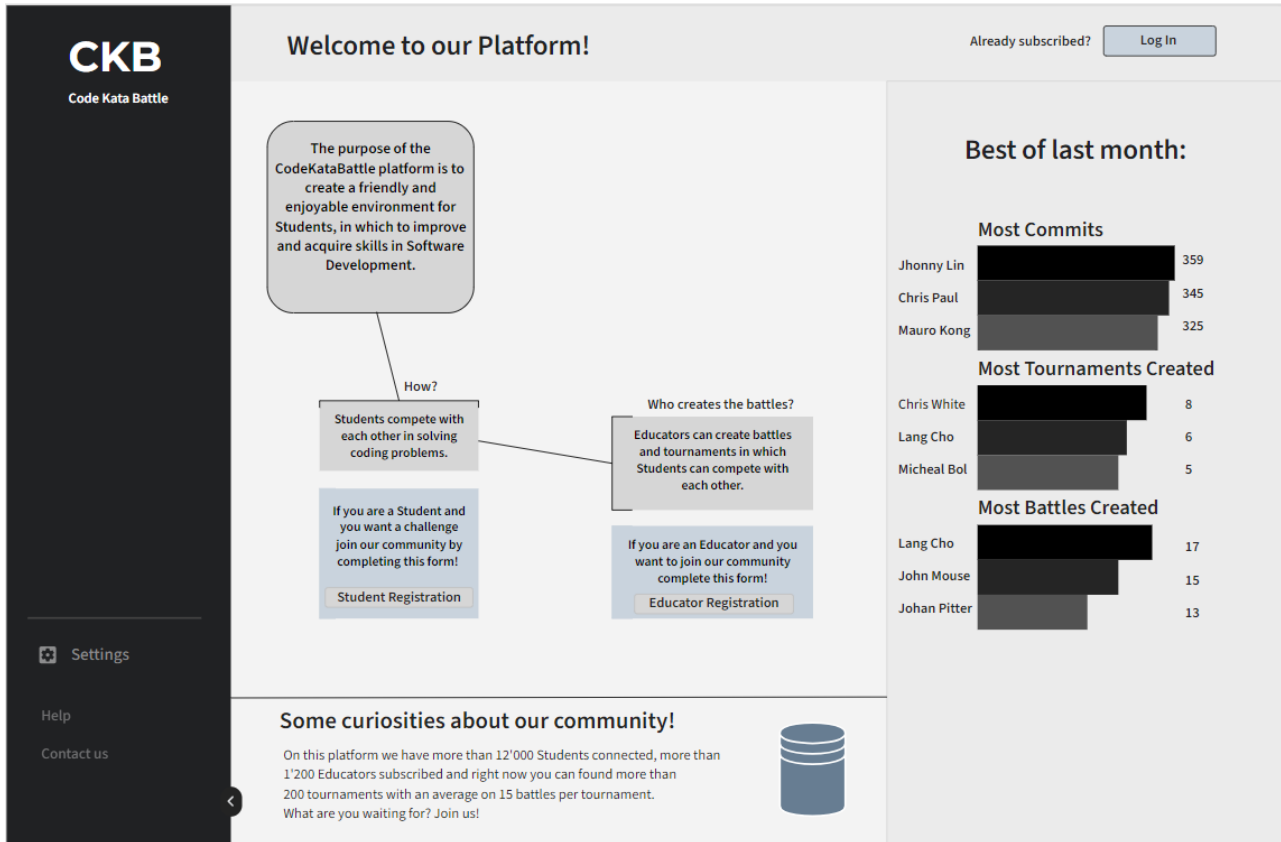
Figure 18: Homepage mockup

Figure 19: Student registration form, mockup

Figure 20: Educator registration form, mockup
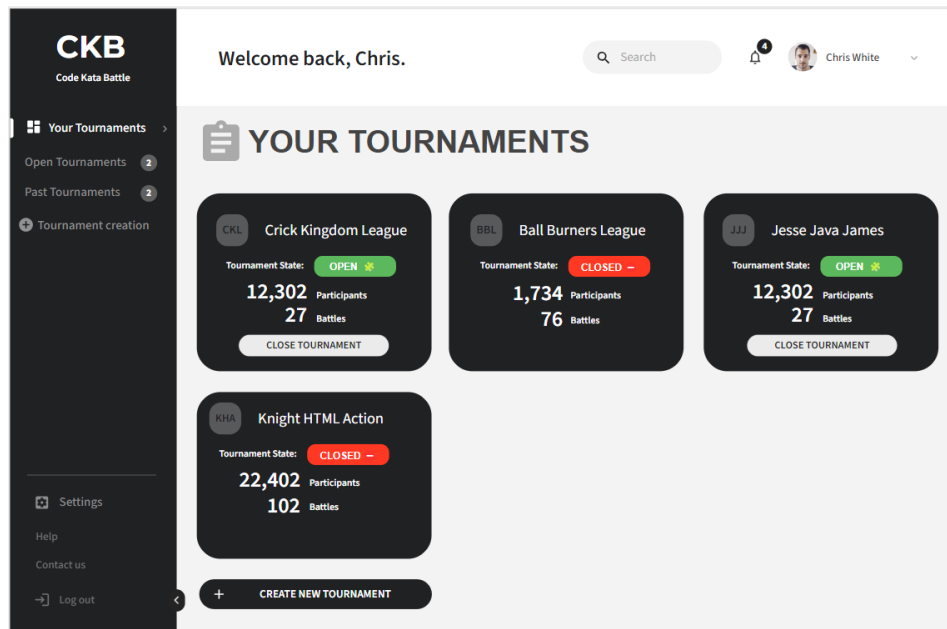
Figure 21: Page of the Educator's tournaments, mockup
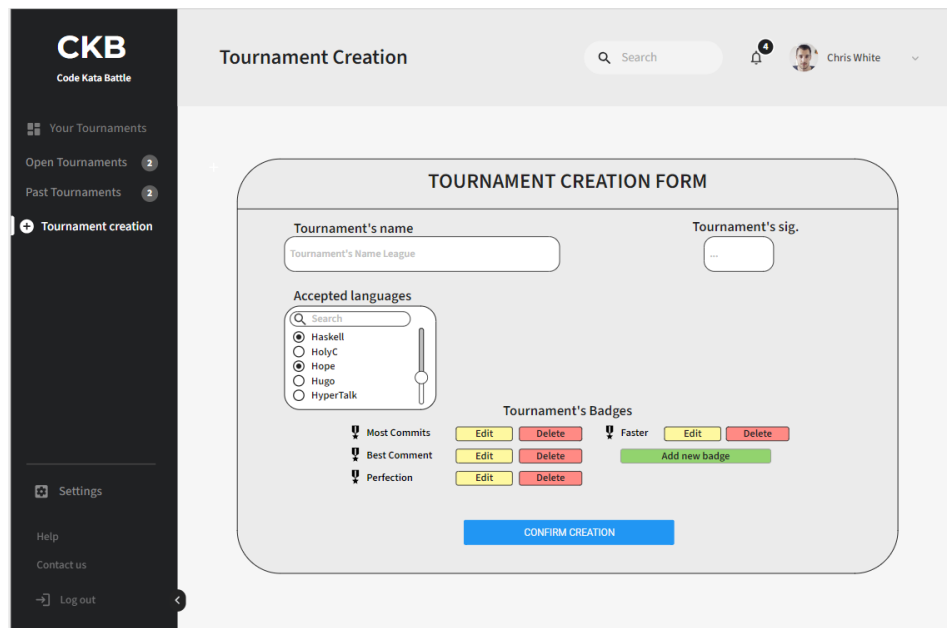
Figure 22: Tournament creation form, mockup

Figure 23: Badge creation form, mockup

Figure 24: Page of tournament's battles created by an Educator, mockup
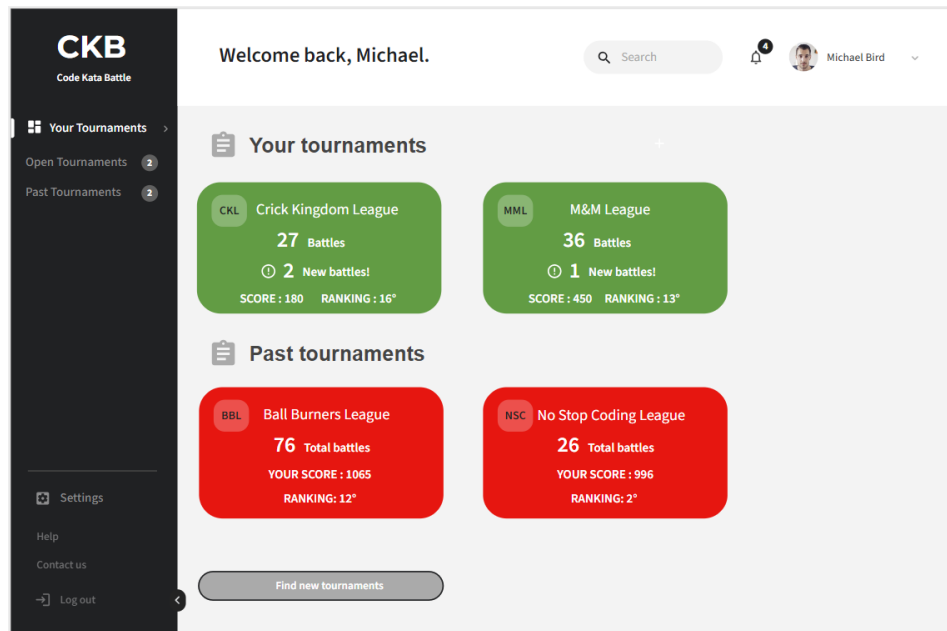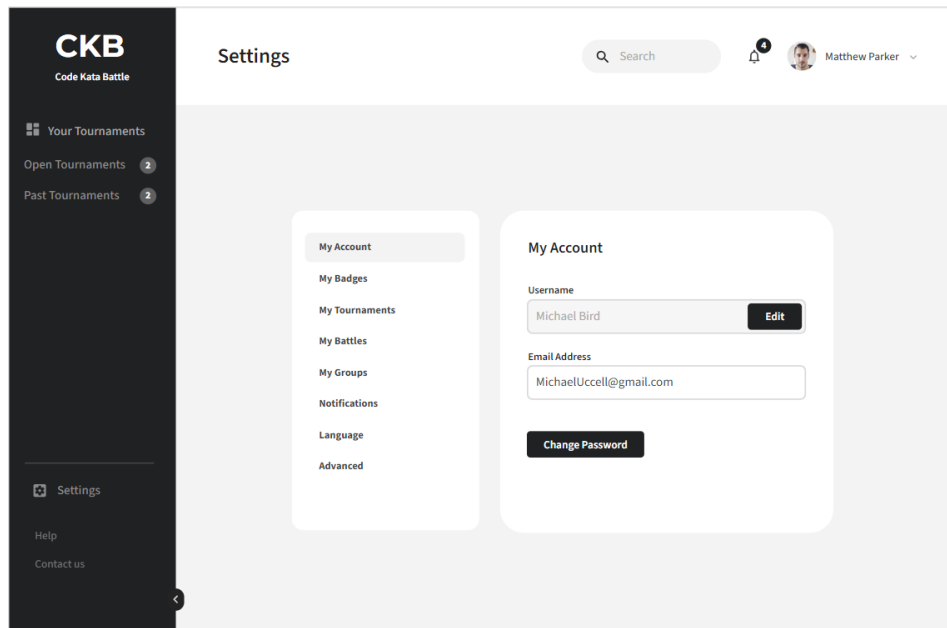
Figure 25: Page of Student's tournaments, mockup

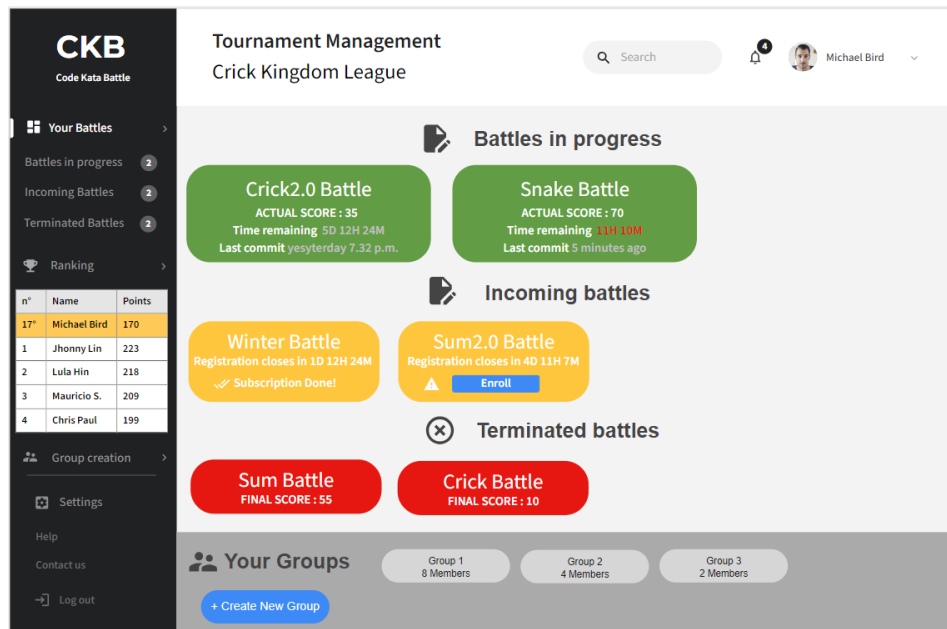Figure 26: Student's profile page, mockup

Figure 27: Page of tournament's battle the Student partecipate to, mockup
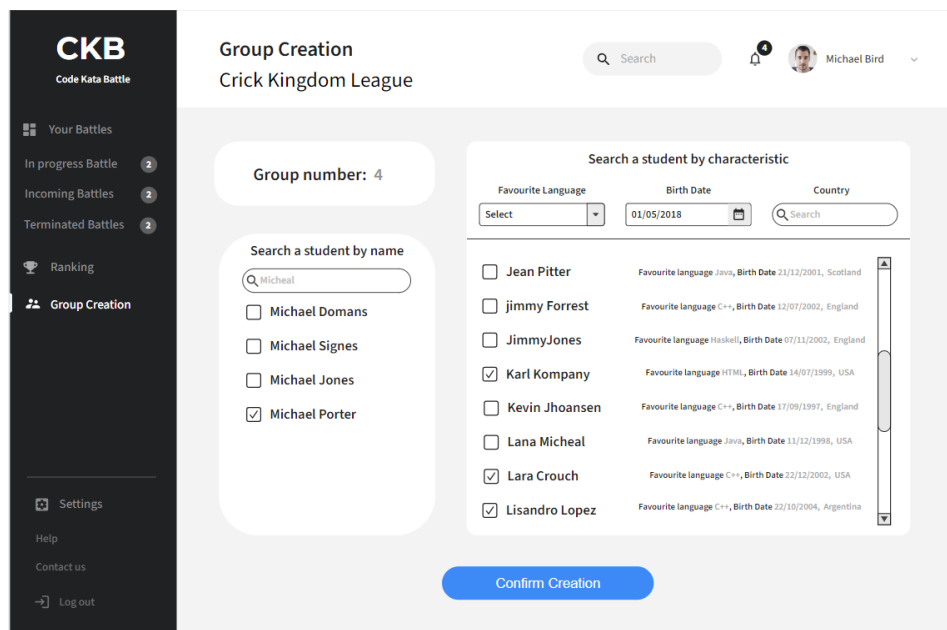
Figure 28: Group creation page, mockup

4 Requirements traceability

- **R.1:** The CKB platform should allow an unregistered Student to create a new account.

  - **Account Manager:** Used to handle the registration requests.
  - **User Database:** Used to store Students' informations after registration.

- **R.2:** The CKB platform should allow an unregistered Educator to create a new account.

  - **Account Manager:** Used to handle the registration requests.
  - **User Database:** Used to store Educators' informations after registration.

- **R.3:** The CKB platform must allow access to its pages only if the used credentials are correct.

  - **Account Manager:** Used to check whether the credentials used are correct and to retrieve users' data.

- **R.4:** The CKB platform must not allow a Student to register more than once in the system.

  - **Account Manager:** Used to check if the Student has already registered to the platform and retrieve Student's data.

- **R.5:** The CKB platform must not allow an Educator to register more than once in the system.

  - **Account Manager:** Used to check if the Educator has already registered to the platform and retrieve Educator's data.

- **R.6:** Educators can access the platform's services only if they are registered to it.

  - **Account Manager:** Used to check if the Educator is registered to the platform and retrieve Educator's data.

- **R.7:** Students can access the platform's services only if they are registered to it.

  - **Account Manager:** Used to check if the Student is registered to the platform and retrieve Student's data.

- **R.8:** The CKB platform should not allow Students to create tournaments and/or battles.

- **Account Manager:** Used to check if the user is a Student and prevent him to access Tournament Manager APIs that should not have access to. Also used to retrieve Student's data.

- **R.9:** The CKB platform should allow Educators to create battles within a tournament only to the tournament creator and to any other Educator that has been granted permission to do so by the tournament's creator.

  - **Account Manager:** Used to check whether the logged Educator can create a battle within a certain tournament and to retrieve Educator's data.
  - **Battle Manager:** Used to create the new battle within the tournament.
  - **Tournament Database:** Used to store the newly created battle's data.

- **R.10:** The CKB platform must allow Educators to personalise the tournaments they create.

  - **Tournament Manager:** Used to create and personalize new tournaments.
  - **Tournament Database:** Used to store tournament's data.

- **R.11:** The CKB platform must allow Educators to personalise the battles they create.

  - **Battle Manager:** Used to create and personalize new battles.
  - **Tournament Database:** Used to store battle's data.

- **R.12:** The CKB platform must allow Educators to define new obtainable badges for each tournament they create.

  - **Badges Manager:** Used to create new badges for a new tournament.
  - **Tournament Database:** Used to store badges data.

- **R.13:** The CKB platform must allow Educators to manually evaluate the solutions uploaded by the Students for the battles that the Educators created.

  - **Evaluation Manager:** Used to allow the Educator to manually evaluate Students' work and check whether the battle is enabled to be manually evaluated.
  - **Account Manager:** Used to retrieve Educator's data and check whether the User has the permission to manually evaluate a battle.
  - **Tournament Database:** Used to store battle and scores' data. The scores are both the automatically generated ones and the ones given by the Educator through the manual evaluation process.

- **R.14:** The CKB platform must allow Educators to delete or update badges before finalizing a tournament's creation.

    - **Badges Manager:** Used to update or delete some badges during tournament creation.
    - **Tournament Database:** Used to store the badge's data.

- **R.15:** The CKB platform must allow Educators to define rules to obtain badges in tournaments created by them.

    - **Badges Manager:** Used to define the rules to obtain the badges.
    - **Tournament Database:** Used to store badges's obtaining rules.

- **R.16:** The CKB platform must ensure that badges' characteristics respect guidelines regarding their name, icon format and rules to obtain them.

    - **Badges Manager:** Used to check if the characteristics of the new badges that the Educator wants to create, or the modifications made to the ones already created, respect some constraints.

- **R.17:** The CKB platform must allow Educators to create new tournaments.

    - **Tournament Manager:** Used to create new tournaments.
    - **Tournament Database:** Used to store tournament's data.

- **R.18:** The CKB platform must ensure that tournaments' characteristics respect guidelines regarding their name, programming language.

    - **Tournament Manager:** Used to check if the characteristics of the tournaments that the Educator wants to create respect some constraints.

- **R.19:** The CKB platform must allow Educators to close tournaments they have created.

    - **Tournament Manager:** Used to allow Educators to close the tournaments that they have created and retrieve old tournament's data.
    - **Tournament Database:** Used to store tournaments' data.

- **R.20:** The CKB platform must ensure that when a tournament is closed, Educators cannot create new battles within it.

    - **Tournament Manager:** Used to prevent Educators to create new battles within closed tournaments and to retrieve the data of the tournament in which the Educator wants to create a new battle.
    - **Battle Manager:** Used to try to create a new battle.

- **R.21:** The CKB platform must ensure that if a group uploads a solution to a battle after the submission's deadline, that solution will not be considered in the score computation by preventing its upload.

    - **Battle Manager:** Used to check if the submission phase of the battle has ended and retrieve battle's data.

- **R.22:** The CKB platform must ensure that the score given to a group in a tournament is coherent with scores given to the same group in the battles they have partecipated in.

    - **Evaluation Manager:** Used to automatically evaluate Students' uploads, give them a score and subsequently update Students' tournament score.
    - **Tournament Database:** Used to store Students's scores.

- **R.23:** The CKB platform must ensure fair competition between group scores. In the tournament's evaluation, the final group score should be the average score of all the battles in the tournament for each group. Any battle with no solution submitted will count as 0 points.

    - **Evaluation Manager:** Used to automatically evaluate Students' uploads. In case of no upload by a group, assigns to them 0 points.
    - **Tournament Database:** Used to store Students's scores.

- **R.24:** The CKB platform must allow Students to subscribe to a tournament.

    - **Tournament Manager:** Used to allow Students to register to a tournament.
    - **Tournament Database:** Used to store registrations' data.

- **R.25:** The CKB platform must allow Students to subscribe to a tournament's battle within the registration deadline.

    - **Battle Manager:** Used to allow Students to register to a battle if the registration deadline has not expired yet.
    - **Tournament Database:** Used to store Student's registration to the battle and retrieve battle's data.

- **R.26:** The CKB platform must allow Students to submit solutions to a tournament's battle within the battle's deadline relying on the external GitHub service.

    - **Evaluation Manager:** Used to retrieve group's solution from GitHub.
    - **Tournament Database:** Used to store solution's data.

- **R.27:** The CKB platform must allow Students to send and receive group invitations to and from other Students in order to form groups.

- **Group Manager:** Used to search Students and send them invitations to groups. Also used to create such groups.
- **Notification Manager:** Takes care of sending emails to the invited Students.
- **Group Database:** Used to store invitation and group's data.

- **R.28:** The CKB platform should allow Students to join a battle only if the group composition rules for that battle are complied with.

  - **Battle Manager:** Used to check whether the group that is trying to access the battle is violating some battle's rules, if not, allows it to register to the battle.
  - **Tournament Database:** Used to store battle registration's data.

- **R.29:** The CKB platform must ensure that solutions uploaded by a Student for a battle are evaluated.

  - **Evaluation Manager:** Used to evaluate solutions uploaded on GitHub by Students.
  - **Tournament Database:** Used to store Students' scores.

- **R.30:** The CKB platform must ensure that only the latest solution uploaded by a Student for a battle he is subscribed to will be taken into consideration for the final score.

  - **Evaluation Manager:** Used to evaluate the last solution uploaded and overwrite Students' score according to it.
  - **Tournament Database:** Used to store Students' scores and new solutions' data.

- **R.31:** The CKB platform must allow groups partecipating in a battle to change their solution, if the battle's submission deadline hasn't expired yet.

  - **Evaluation Manager:** Used to retrieve latest group's solution from GitHub and evaluate it.
  - **Tournament Database:** Used to store Students' scores and new solutions' data.

- **R.32:** The CKB platform must allow an Educator to modify the score for a Student's solution.

  - **Evaluation Manager:** Used to allow an Educator to manually evaluate Students' work and change their score.
  - **Tournament Database:** Used to and store Students' scores and retrieve Students' work and score.

- **R.33:** The CKB platform must ensure that when a new tournament is created, all Students subscribed to the platform are going to receive a notification.

  - **Notification Manager:** Used to send a notification to all the Students registered to the platform.

  - **Account Manager:** Used to retrieve Students' data.

- **R.34:** The CKB platform must ensure that when a new battle is created in a tournament, all Students subscribed to that tournament are going to receive a notification.

  - **Notification Manager:** Used to send a notification to all the Students registered to the tournament in which the battle will take place.

  - **Account Manager:** Used to retrieve Students' data.

- **R.35:** The CKB platform must allow Students to visualise the score they obtained in a battle they partecipated in.

  - **Battle Manager:** Used to allow Students access the score obtained in a battle they partecipated to.

  - **Tournament Manager:** Used to retrieve battle's data and specifically Student's results in it.

- **R.36:** The CKB platform must allow Students to visualise the score they obtained in a tournament they partecipated in.

  - **Tournament Manager:** Used to allow Students access the score obtained in a tournament they partecipated to. Also used to retrieve tournament's data and specifically Student's results in it.

- **R.37:** The CKB platform must allow Students to visualise the badges they obtained.

  - **Account Manager:** Used to allow Students to access their, or others' profile pages and visualize the badges in it. Also used to retrieve Student's data.

- **R.38:** The CKB platform must ensures that battles' characteristics respect guidelines regarding their name, deadlines, programming language, number of member per group.

  - **Tournament Manager:** Used to check if the characteristics of the battles that the Educator want to create, respect some constraints.

5   Implementation, Integration & Test plan

The CKB platform will be divided in:

- **Client;**

- **Web Server;**

- **Application Server;**

- **Databases;**

- **External APIs.**

These elements will be implemented by exploiting the microservices architecture feature that exploits the fact that each microservice is separated from the others and works independently. Knowing this, we can implement and test each microservice through several small development teams, reducing in this way the huge synchronization overhead that would be necessary in case of a monolithic platform. Other key factors are the reusability and composability of the functionalities offered by the implemented microservices and the smaller codebase provided by this architecture, which makes debugging easier and maintenance cheaper. The most important approaches used are the Bottom-Up approach and the Thread approach. The Bottom-Up approach consists in starting building the system from its foundations and from there going up to the most in-depth components. The Thread approach focuses more on developing in parallel specific functionalities across multiple components providing in this user-visible program progress.

5.1   Implementation Plan

5.1.1   Backend

Step 1 (Bottom-Up): The implementation should start with a bottom-up approach, which will allow to develop first the Account Manager and Notification Manager components. These are, within all the components, the ones that will interact more with the others and for this reason they should be developed first.

    Step 2 (Thread): After having fully developed the two components mentioned in the precedent paragraph, the rest of the services should be developed in parallel by exploiting fully the Microservices Architecture's characteristics. In particular the Thread development approach should initially focus on the major functionalities that the services have to offer, especially the ones that

---

require the communication (such as the creation of a battle in a tournament, the evaluation of the battle's uploads, . . .) through lightweight protocols with other components and later moving to the simplier/less important ones.

5.1.2   Frontend

In parallel with the development of the platform's backend, the frontend can also be developed, possibly (not mandatory) following the development of the various functionalities.

5.2   Integration Plan

The integration plan follows the implementation plan explained above. The main part of the platform is developed through the Thread approach. This allows to directly integrate the components while they are being developed. In our case the integration will mostly happen during the early/mid stages of the development, during which the several teams will focus on the major functionalities that the components will offer.

5.3   Test Plan

The test plan includes several types of testing, such as:

- **Unit testing:** Testing used in order to check whether the written code does what is supposed to. Should be performed immediately after the completition of a component's feature or of the component itself, in order to immediatly spot eventual implementative bugs.

- **Functional testing:** Used to check if the CKB platform meets the functional requirements established in the RASD document. The functional testing is very important and will be done in parallel with the unit testing.

- **Performance testing:** Used to detect platform's bottlenecks that can potentially affect the platform's response time, utilization and throughput, which are very important requirements specified in the RASD document. It's also used to detect inefficiencies, optimization opportinities and HW/Network issues.

- **Load testing:** Used in order to identify memory leaks, buffer overflows and other memory related problems.

- **Stress testing:** Used to test the capability of the platform to recover after a failure.

6 Effort Spent

Group member		Effort spent
Francesco Spangaro	Introduction Architectural Design User Interface Design Requirement Traceability Implementation, Integration and Test plan	*2h* *14h* *3h* *3h* *3h*
Luca Tosetti	Introduction Architectural Design User Interface Design Requirement Traceability Implementation, Integration and Test plan	*3h* *10h* *3h* *3h* *4h*
Francesco Riccardi	Introduction Architectural Design User Interface Design Requirement Traceability Implementation, Integration and Test plan	*1h* *7h* *14h* *1h* *1h*

7 References

7.1 Paper references

- Specification document Assignment RDD AY 2023-2024.pdf

7.2 Used tools

- GitHub for project versioning.

- Diagrams.net for UML diagrams.

- Visual Studio Code as LaTeX editor.

- Mockups.com for the mockup's creation.