POLITECNICO DI MILANO

Software Engineering II

---

# CodeKataBattle

Design Document

Version 1.0

---

Federico Albertini

Aleksandro Aliaj

Leonardo Betti

*January, 2024*

# 1. Introduction

## 1.1 Purpose

This document serves as a comprehensive guide to the architectural design and component structure of the CodeKataBattle (CKB) platform. Its primary purpose is to provide a clear and detailed view of how the platform's components interact to facilitate competitive coding challenges, known as code katas, for students in a gamified environment. The document is intended for a technical audience, including the development team, system architects, and project managers, as well as non-technical stakeholders such as educational institution administrators and potential investors who are interested in understanding how the CKB platform is built and operates. Through this document, all parties will gain a comprehensive understanding of the platform's inner workings, facilitating informed decision-making and effective collaboration.

## 1.2 Scope

The scope of this document is to provide a detailed architectural design and component structure of the CodeKataBattle (CKB) platform, which is engineered to host, manage, and assess competitive coding challenges (code katas) in a tournament-style setting to enhance students' coding skills. It includes a comprehensive breakdown of the platform's individual components such as Tournament Management, Battle Setup, Team Formation, Repository Management, Automated Workflow, Scoring Engine and Badge System. The document also details how users, including students and educators, interact with the platform for activities ranging from tournament creation to code submission and feedback reception, alongside outlining the integration points with external services like GitHub for repository management and code submission workflows. This document aims to serve as a living guide, updated regularly to reflect the evolving nature of the CKB platform, ensuring clarity, scalability, and effectiveness in its design and use. For more in depth detail, suggest viewing the Requirements Analysis and Specification Document (RASD).

## 1.3 Definitions, Acronyms, Abbreviations

### 1.3.1 Definition

- Admin: educator who creates a tournament, and then battles.
- Collaborator: educator who participates in a tournament, to help create new battles.

### 1.3.2 Acronyms

- [RASD] Requirement Analysis and Specification Document.
- [UI] User Interface
- [CK] - code kata( it consists in the brief textual description and a software project with build automation scripts with test cases, but without implementation)
- [CKB] - CodeKataBattle platform
- [TDD] - Test driven development
- [CEB] - Code Evaluation Bridge
- [UX] - User Experience
- [API] - Application Programming Interface
- [VM] - Virtual Machine
- [CRUD] - Create, Read, Update, Delete
- [DB] - Database

### 1.3.3 Abbreviations

- Repo: Repository
- Def: Definition
- E.G.: Example
- RESTful: REpresentational State Transfer

## 1.4 Revision history

- Version 1.0

## 1.5 Reference Documents

This document is strictly based on:

- The specification of the DD assignment of the Software Engineering II course, held by professor Elisabetta Di Nitto A.Y 2023/2024;
- Slides of Software Engineering 2 course on WeBeep;
- Official link of https://www.codewars.com/ to get more information about battle
- https://docs.github.com/en/webhooks GitHub WebHooks documentation

- https://docs.github.com/en/actions GitHub Action documentation
- https://azure.microsoft.com/en-us/products/devops/ Azure DevOps

## 1.7 Document Structure

1. ***Introduction:*** This initial section provides an overview of the document's intent, a recap of key concepts previously detailed in the RASD, and essential information for the reader, including definitions, acronyms, synonyms, and referenced documents.

2. ***Architectural Design:*** This chapter offers a comprehensive description of the system's architecture, presenting a high-level overview of the elements, the software components of CLup, detailed runtime diagrams illustrating various system functionalities, and an exhaustive discussion on the architectural patterns employed.

3. ***User Interface Design:*** This section showcases the mockups for the application's user interfaces, detailing the navigation flow between them to facilitate an understanding of the user's journey through the application.

4. ***Requirements Traceability:*** Here, the document establishes a connection between the requirements outlined in the RASD and the components introduced in this document, serving as validation that the design choices align with and fulfill the specified requirements and goals.

5. ***Implementation, Integration, and Test Plan:*** This chapter delineates the methodologies and procedures for the system's implementation, integration, and testing, guiding developers to ensure the system is developed correctly and efficiently.

6. ***Effort Spent:*** This section documents the time and resources expended during the creation of the document.

7. ***References:*** The final section lists all the documents, literature, and software referenced or utilized in the preparation of this document.

Each section is crafted to provide detailed insights and guidelines, ensuring a comprehensive understanding of the system's design, implementation, and operation.

# 2. Architectural design

## 2.1 Overview

In this section is given an overview of the architectural elements that compose the system, their interaction and a description of the replication mechanism chosen for the system in order to make it distributed.

### 2.1.1 High level view

In alignment with the strategic decision to implement a 3-tier architecture, this section provides a high-level overview of the design and interaction between the Client Tier, Server Tier, and Data Server. This architecture is chosen for its scalability, maintainability, and clear separation of concerns, enhancing the system's overall performance and security.

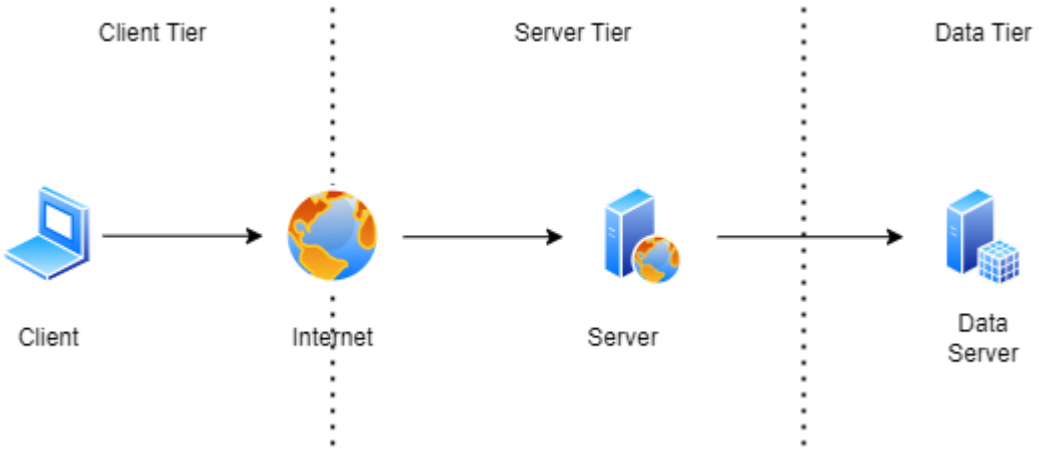



*Figure 2.1 - High level system architecture*

Client Tier: This layer is the user's entry point to the system. It comprises the user interface and client-side logic, enabling users to interact with the application through web browsers. This tier is designed to be lightweight, dynamic, and responsive, providing a seamless user experience while communicating with the Server Tier for data processing. It will contain logic on how and when to display the data that the Server Tier will provide.

Server Tier: The Server Tier acts as the middleman between the Client Tier and Data Server. It is further subdivided into three main components:

Web Server: This server handles HTTP requests from the client, delivering data and other content from the application server to the client. The web

server communicates with the application server using a standardized interface, often referred to as the "Application Programming Interface (API)." This API is designed to ensure that requests and data are passed efficiently and securely, maintaining the integrity and performance of interactions between the two servers.

Application Server: At the heart of the Server Tier, the application server will be a RESTful application that executes the business logic, processes user commands, performs calculations, and makes logical decisions. It interacts directly with the Data Server to retrieve, manipulate, and store data. This section will communicate directly with gitHub to be informed when a new commit is pushed by the students and retrieve the code to be tested and calculate the scores. This procedure will use the github WebHooks interface that allows to know when specific actions on the Repo take place .

Test Server: This server's purpose is to compile, run, and assess the code submissions for correctness, efficiency, and adherence to specified requirements. It provides a controlled environment where student submissions are rigorously tested and evaluated to ensure they meet the desired outcomes of the battles. The Test Server communicates with the Application Server via a custom interface known as the Code Evaluation Bridge (CEB). This proprietary interface facilitates secure and efficient data exchange, ensuring that the code submissions are seamlessly transferred, tested, and the results relayed back to the Application Server for further action.

Main Database : This is the primary database where all current and active data is stored. It handles real-time queries and transactions, ensuring data integrity and security.
Archive Database: To maintain system performance and manage data lifecycle, older data is transferred to an archive database. This database is optimized for storing, retrieving, and analyzing historical data.
The main Database will have some policies on when to move the data to the archive. The aim of this decision is to have a reliable and fast retrieval of the live data but also maintain the history of all the Tournaments and Battles created using less space. This logic is implemented via specific triggers.

The 3-tier architecture ensures that each layer can operate and scale independently, providing a robust framework for the application's growth and evolution. By segregating the system into distinct tiers, the application can better manage user requests, process data efficiently, and store information

securely, all while maintaining a high level of performance and user satisfaction. This design also simplifies maintenance and updates, as changes in one tier do not necessarily impact the others, reducing downtime and enhancing the system's adaptability to future requirements.

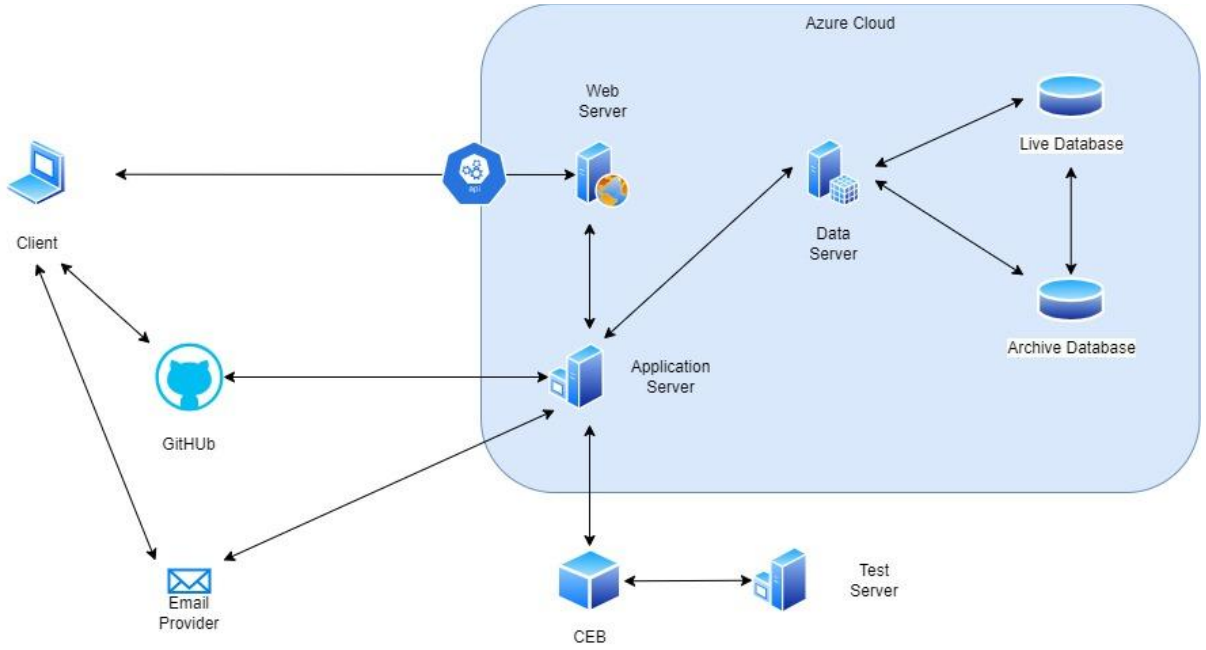



*Figure 2.2 - High level system architecture 2*

### 2.1.2 Distributed view

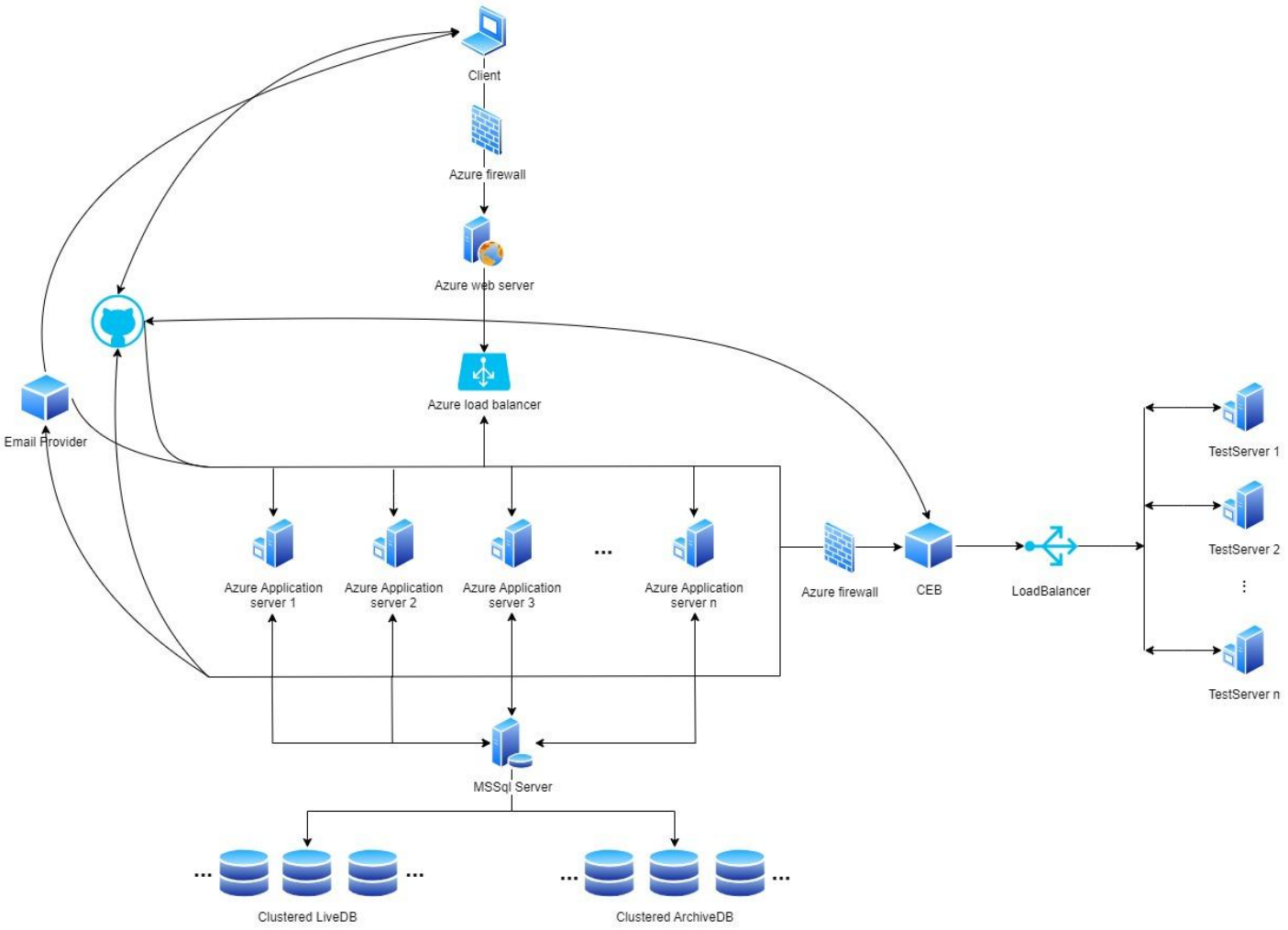



*Figure 2.3 - Distributed CKB architecture*

- **Application Server Cluster:** This cluster comprises several Azure Application Servers, indicated as 'server 1' to 'server n'. These servers are designed to handle the application's business logic and process user requests. The clustering indicates a high availability and load resilience setup, ensuring that the application can handle a high volume of requests and distribute the load efficiently. These servers are interconnected and managed by an Azure Load Balancer, which directs user requests to the servers based on current load and availability, thus optimizing resource utilization and response times.

- **Database Cluster:** The architecture includes two clusters of databases: the LiveDB cluster and the ArchiveDB cluster. The LiveDB cluster is focused on handling real-time data transactions and is designed for high-performance access to current data, such as ongoing tournaments and battles. The ArchiveDB cluster, in contrast, stores historical data, such as completed tournaments and past battle results. Both clusters are managed by an MS SQL Server, indicating that Microsoft SQL Server technology is used for database management. The clustered configuration suggests that the databases are set up for redundancy and reliability, ensuring data persistence and stability even in the case of individual node failures.

- **Test Server Cluster:** TestServers in this diagram are specifically purposed for evaluating the code submitted by students in coding battles. Each TestServer, labeled from 'server 1' to 'server n', represents a scalable and parallelized environment where submitted code can be run and tested. This setup allows multiple submissions to be evaluated simultaneously, facilitating a quick and efficient turnaround in providing feedback to students. These servers are behind a LoadBalancer, ensuring that no single server is overwhelmed with requests and that code submissions are evenly distributed for testing. The CEB (Code Evaluation Bridge) acts as a communication gateway between the application server cluster and the test servers, signifying a specialized protocol or set of APIs designed to manage the transfer of code to be tested and the retrieval of test results.

The architecture delineates a robust and scalable system capable of handling both the operational demands of the application and the specific requirement of testing student-submitted code in an efficient, parallelized manner. The clustered approach not only provides resilience and high availability but also the flexibility to scale resources according to the system's demands.

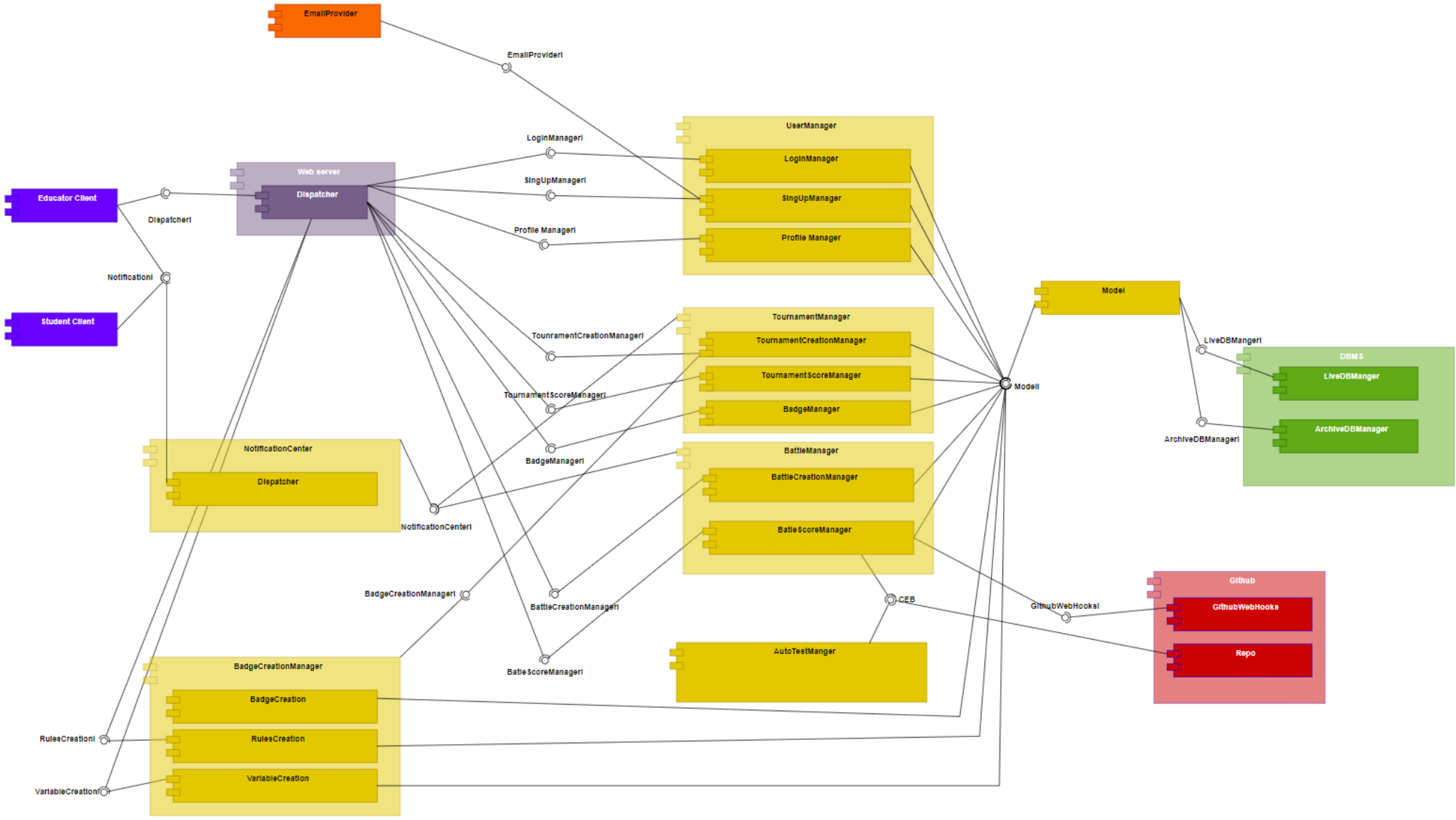
## 2.2 Component view

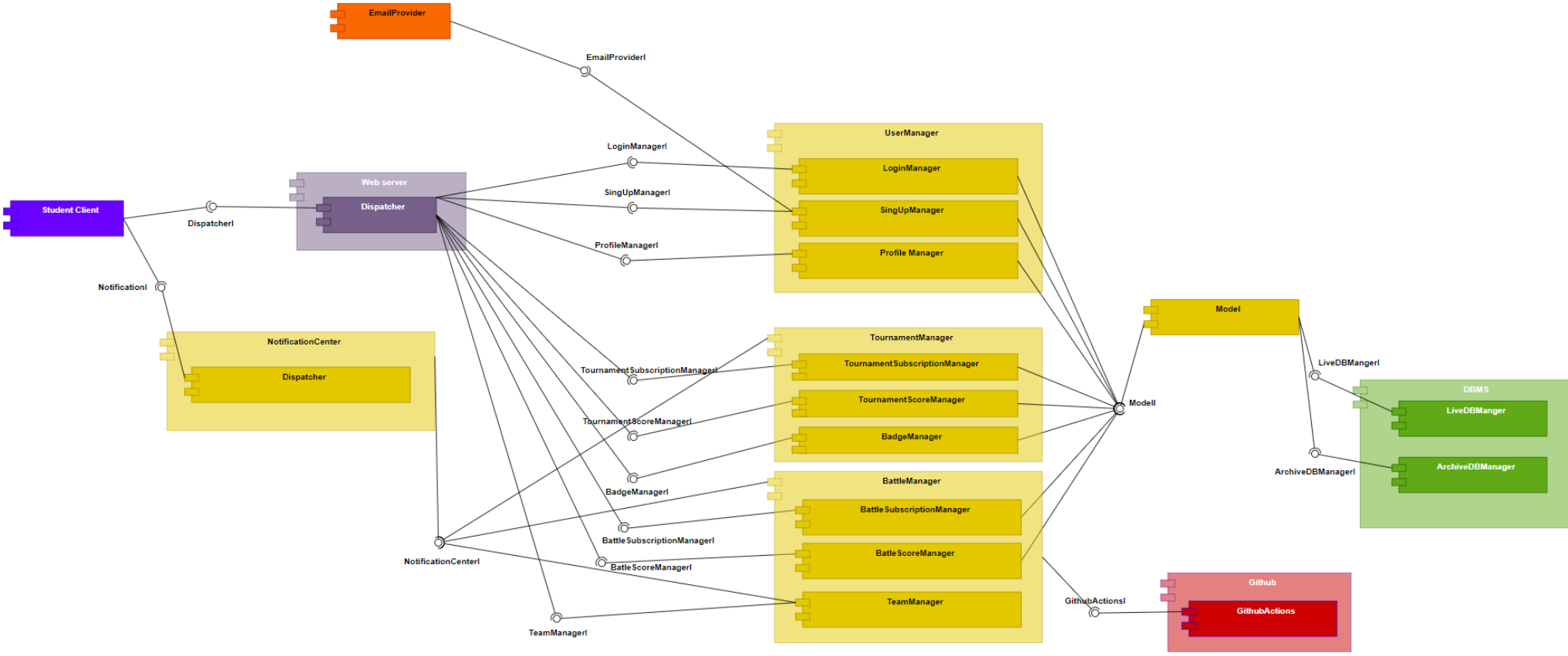
The next figures represent the component view of our system. We divided it two to separate diagrams to show the two types of user that can access the system and what component they have access to. The yellow elements are the system's component, the purple ones are the client of our system, the email provider being a third party service is orange and finally the GitHub interfaces and components are in red.

The following pages show:

*Figure 2.5 - CKB Educator component diagram*

*Figure 2.6 - CKB Student component diagram*


EmailProvider
EmailProviderI
LoginManagerI
SingUpManagerI
Profile ManagerI
UserManager
LoginManager
SingUpManager
Profile Manager
Web server
Dispatcher
Educator Client
DispatcherI
NotificationI
Student Client
TounramentCreationManagerI
TournamentScoreManagerI
BadgeManagerI
TournamentManager
TournamentCreationManager
TournamentScoreManager
BadgeManager
Model
ModelI
LiveDBMangerI
ArchiveDBManagerI
DBMS
LiveDBManger
ArchiveDBManager
NotificationCenter
Dispatcher
NotificationCenterI
BattleManager
BattleCreationManager
BattleScoreManager
CEB
BadgeCreationManagerI
BattleCreationManagerI
BattleScoreManagerI
AutoTestManger
GithubWebHookI
Github
GithubWebHooks
Repo
BadgeCreationManager
BadgeCreation
RulesCreation
VariableCreation
RulesCreationI
VariableCreationI


EmailProvider
EmailProviderI
UserManager
LoginManagerI
LoginManager
Web server
Dispatcher
SingUpManagerI
SingUpManager
Student Client
DispatcherI
ProfileManagerI
Profile Manager
NotificationI
Model
NotificationCenter
Dispatcher
TournamentManager
TournamentSubscriptionManagerI
TournamentSubscriptionManager
LiveDBMangerI
DBMS
TournamentScoreManagerI
TournamentScoreManager
ModelI
LiveDBManger
BadgeManager
ArchiveDBManagerI
ArchiveDBManager
BattleManager
BadgeManagerI
BattleSubscriptionManager
BattleSubscriptionManagerI
BatleScoreManager
NotificationCenterI
BatleScoreManagerI
Github
TeamManager
GithubActionsI
GithubActions
TeamManagerI

Educator components view:

- The **Educator Client** serves as the front-end interface for educators to interact with the CodeKataBattle (CKB) website. It establishes communication with the Web Server using HTTP calls, requesting and retrieving data. The client then renders this data for the educator in accordance with predefined logic rules set by the system developers, ensuring a user-friendly and intuitive experience.
- The **Web Server** is a critical component tasked with managing all HTTPS requests. It acts as the initial receiver of these secure requests and is responsible for routing them to the appropriate controllers on the Application Server. Integral to its function is the **Dispatcher**, a sub-component designed to interface directly with incoming requests, ensuring they are processed accurately and efficiently within the server's architecture.
- The **UserManager** component is an integral part of the system that oversees all activities related to user profiles. Within this component, the **LoginManager** is tasked with all login-related functionalities, ensuring secure user authentication and session management. Concurrently, the **SignUpManager** handles new user registrations by verifying the input data with the **EmailProvider** to dispatch verification emails. Complementing these two, the **ProfileManager** takes charge of processing requests about the student's profile, managing the viewing and editing of profile information to maintain up-to-date and accurate user data.
- The **EmailProvider** operates as an external service that integrates with the **SignUpManager**. Upon successful validation of user data during the registration process, the **EmailProvider** is triggered to dispatch a verification email to the user, thereby completing the signup procedure and ensuring the user's email address is confirmed for secure communications.
- The **TournamentManager** is a comprehensive component tasked with overseeing all aspects of tournament functionality on the platform. It is segmented into three focused sub-components, each with a distinct responsibility. The **TournamentCreationManager** is the manager for tournament setup, springing into action when an educator initiates a tournament through a POST request. This sub-component ensures that the tournament is created with all the specified details in place. Meanwhile, the **TournamentScoreManager** is responsible for the meticulous task of managing and updating the scores of each student participating in the tournament. Finally, the **BadgeManager** is charged

with the distribution of badges to students at the end of the tournament, recognizing their achievements. While the **TournamentCreationManager** is reserved for use by educators, ensuring they have exclusive control over the initiation of tournaments, the other components work together to manage the tournament lifecycle and student engagement.

- Similar to the **TournamentManager**, the **BattleManager** is a dedicated component designed to handle all features related to the individual battles within the platform. It encompasses two specific sub-components that streamline the process of battle management and creation. The **BattleScoreManager** is responsible for the crucial task of managing the scores for each team engaged in the current battle, ensuring that points are accurately tallied and reflected. On the other hand, the **BattleCreationManager**, which is accessible exclusively to educators, comes into play when there's a need to initiate a new battle. This sub-component validates the format of the data provided and, upon confirmation, proceeds to create a new battle, setting the stage for another round of competitive coding challenges.

- The **AutoTestManager** is a specialized component within the system, designed to autonomously evaluate student-submitted code. Its primary role kicks in once the **BattleScoreManager** receives a notification from the **GitHubWebHooks** indicating that new code has been submitted to the repository. Upon receiving this trigger, the **AutoTestManager** retrieves the submitted code from the repository and prepares it for evaluation. To facilitate this process, it employs the **Code Evaluation Bridge** (**CEB**), a bespoke interface that serves two main functions. Firstly, it communicates to the **AutoTestManager** when a new build of the battle's solution is ready, and secondly, it provides the mechanism through which the actual executable code is obtained for testing. This ensures that the **AutoTestManager** has all the necessary tools and information to accurately assess the code, determining its correctness, efficiency, and adherence to the battle's specifications.

- The **GitHub** supercomponent represents an external but integral part of the system, serving as the platform where students submit their code. While this component is not under direct control of the system, it's crucial for the operational workflow, particularly through the use of **GitHubWebHooks**. These WebHooks are essentially API calls

configured to notify our system of specific actions taken on the repositories, such as new code submissions or changes. When these events occur, the **GitHubWebHooks** send signals to the appropriate components within our system, triggering subsequent processes like code retrieval and evaluation.

- Our system incorporates a **Database Management System** (DBMS) to efficiently handle all database-related operations, particularly the Create, Read, Update, and Delete (CRUD) functionalities. To optimize performance and organization, the DBMS oversees two distinct databases, each managed by its own dedicated sub-component. The **LiveDBManager** is responsible for the live database, handling real-time data that is actively being used and modified within the platform. This includes current user information, ongoing battle scores, and other data that require immediate access and frequent updates. Conversely, the **ArchiveDBManager** oversees the archive database, which is designated for older data that is no longer in active use but still holds value for historical reference or analytical purposes.

- The **Model** component in our system plays a pivotal role as the central structure representing the application's data model. It functions as the core framework where all data-related interactions converge. Each component within the system interfaces with the Model to exchange and manipulate data. This interaction involves retrieving data from, or submitting data to, the Model based on the specific operations performed by different components like user management, tournament handling, and battle management. The Model ensures that data is properly formatted and adheres to the predefined schemas before passing it on to the DBMS for storage.

- The **NotificationCenter** component is a dedicated module within our system designed to manage and route both internal and external notifications effectively. Its primary function is to serve as a centralized hub that receives various types of notifications from different components of the system and then directs them to the appropriate destinations. This may include user alerts, system updates, error messages, or any other significant information that needs to be communicated .For internal notifications, the NotificationCenter ensures that messages and alerts are promptly delivered to the relevant components within the system, facilitating smooth and coordinated

operations. For external notifications, such as those intended for users or third-party services, it handles the dispatch and delivery, ensuring that each message reaches its intended recipient accurately and securely.

- The **BadgeCreationManager** is an overarching supercomponent within our system, specifically designed to oversee the entire process of badge creation and management. This module is further divided into three distinct sub-components. The first sub-component, **BadgeCreation**, is solely focused on the actual construction and design of the badges. Next, the **RulesCreation** sub-component is tasked with establishing the criteria and conditions under which badges are awarded. It is responsible for creating a set of rules that dictate how and when badges are earned. Lastly, the **VariableCreation** sub-component works in tandem with **RulesCreation**. It is responsible for defining and managing the variables that the rules utilize. These variables might represent specific actions, scores, or other quantifiable data that are relevant to earning badges.

Student Component view:

- **StudentClient** represents the front-end client that students use to interact with the CodeKataBattle website, facilitating various user actions and displaying information received from the server.
- The **TournamentManager** is a key component within the system, equipped with three sub-components, each dedicated to a specific aspect of tournament functionality: The **TournamentSubscriptionManager** is responsible for overseeing the enrollment process of tournaments. It handles all aspects of subscriptions and ensures that their entries are properly recorded and managed within the system. The **TournamentScoreManager** plays a crucial role in calculating and maintaining the scores of students throughout the duration of a tournament. This involves the continuous updating of scores as students progress through the various stages of the tournament. Lastly, the **BadgeManager** is tasked with the distribution of badges upon the conclusion of a tournament. It operates by applying the established badge rules to determine eligibility and award badges to students based on their achievements.

- The **BattleManager** is a central component designed to streamline and manage the processes of the battle aspects within the platform. It comprises three subcomponents: The **BattleSubmissionManager** is the gateway for student or team submissions. It meticulously handles the logistics of accepting and organizing battle's submissions. The **BattleScoreManager** is tasked with the crucial role of scoring. For each team or individual student, this subcomponent calculates the scores based on their performance in the battle, applying the rules and criteria set out for the competition. The **TeamManager** is dedicated to the administration of student teams. It manages team formation, membership, and potentially team-specific data. This subcomponent is essential for team-based battles.

- **GitHubActions** operates as an external automation component that integrates with the system, providing students with the capability to define a series of operations in a .yml (YAML) file. These predefined operations are automatically executed by **GitHub** after each push to a repository, which is a crucial step in the workflow of the platform. The .yml file, crafted by students, specifies a sequence of tasks that **GitHubActions** will perform. The successful execution of these tasks is essential as it ensures that the code is in the correct state and environment for the **TestServer** to evaluate.

**Data Model Description**

This section will provide readers with a deeper understanding of the underlying data structures that support the platform's functionality, highlighting the relational nature of the system's core entities and their interdependencies. Our system's data model is intricately designed to support the complex functionalities of the CodeKataBattle platform. Central to this model is a structured relational database schema that ensures the integrity and accessibility of data across various components of the system.

At the heart of the data model are several key tables:

**Users**: Stores essential information about each user, including their unique identifier, name, last name, email, and role within the platform. This table is fundamental for user authentication and profile management.

**Tournaments**: Captures details of each tournament, such as the title, language of implementation, type, and a description. It also includes

timestamps for the subscription deadline and the tournament end date, ensuring the timely management of each event.

**TournamentSubscribers**: Keeps track of which users are subscribed to which tournaments, along with their scores. This table is pivotal for managing tournament participation and performance tracking.

**Teams**: Maintains records of teams, linking user accounts to specific teams within the platform, facilitating team-based participation in battles and tournaments.

**Battles**: Records individual battles, their titles, descriptions, and participation constraints such as the minimum and maximum number of participants. It also includes timestamps for subscription and submission deadlines, integral for battle lifecycle management.

**BattleSubscribers**: Connects teams to the battles they've subscribed to and records their scores, crucial for monitoring battle progress and outcomes.

**TournamentBattles**: Associates battles with the tournaments they belong to, allowing for a multi-layered structure where battles contribute to the broader context of a tournament.

**Badges**: Lists the badges that can be earned by users, along with their names and definitions, thus underpinning the gamification aspects of the platform.

**UserBadges**: Associates badges with the users who have earned them, serving as a recognition log for user achievements.

**BadgesRules**: Links badges to the rules that define how they are awarded, ensuring that badge allocation is driven by clear, predefined criteria.

**Rules**: Defines the rules used across the platform, including their names and definitions. This table is crucial for establishing the logic that governs various platform actions and rewards.

**VariablesRules**: Associates rules with variables, which are defined in a separate Variables table. This association is key for dynamic rule evaluation based on varying data points.

**Variables**: Contains the variables that are used within rules, storing their names and definitions. These variables allow for flexible and complex rule definitions that can adapt to the changing state of user interactions and achievements.

The relationships between these tables are carefully constructed to allow complex queries and operations that support the platform's functionality. Foreign keys and unique identifiers ensure referential integrity, while various attributes capture the state and properties of each entity within the system. This data model is a vital component of the system's backend, enabling robust data management that supports a rich feature set and a dynamic user experience.

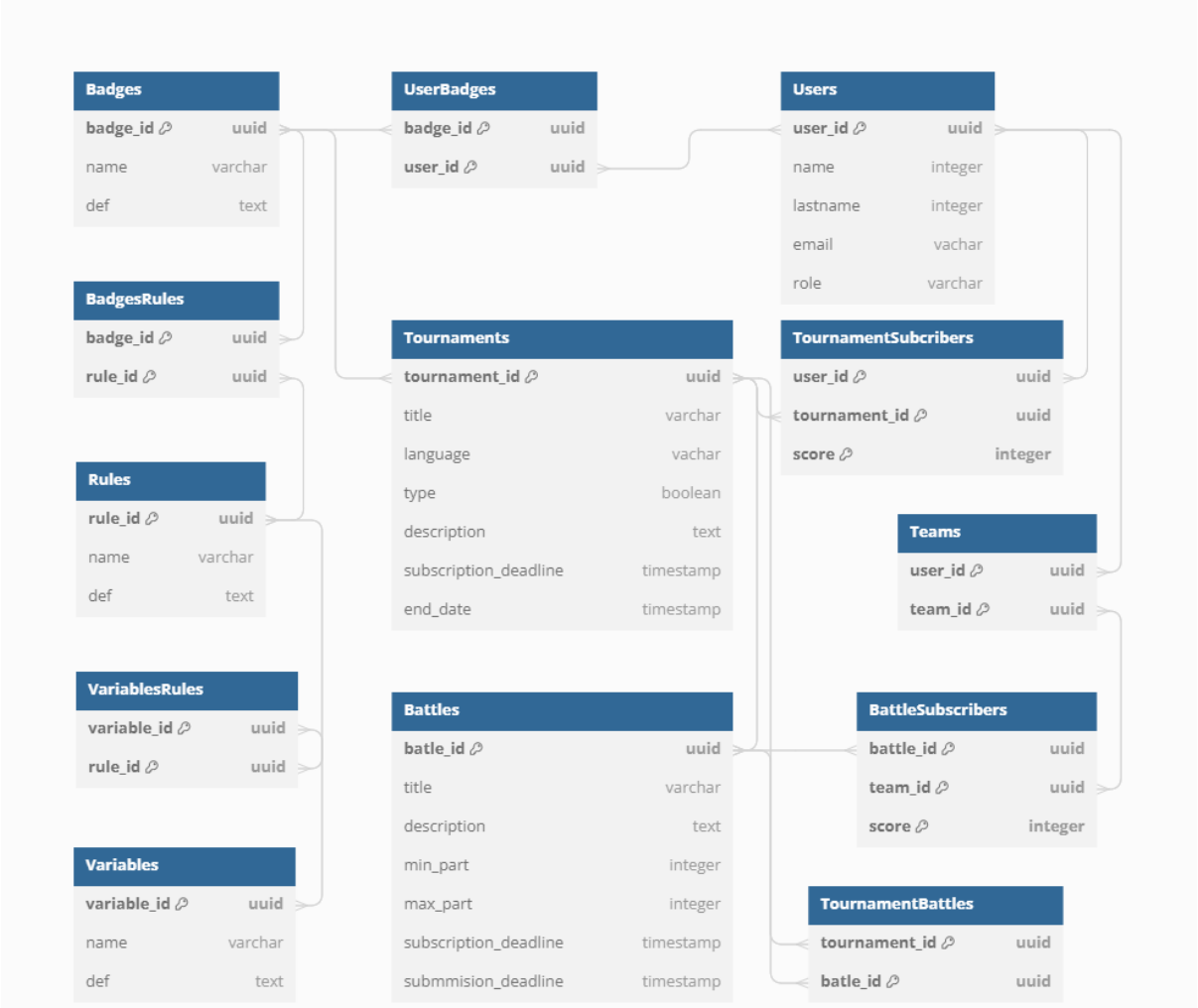



*Figure 2.6 - CKB Data Model Diagram*

## 2.3 Deployment view

The deployment diagram for the CodeKataBattle platform delineates a multi-tiered architecture that separates client interfaces, server operations, and data management across different environments to optimize performance, security, and scalability.

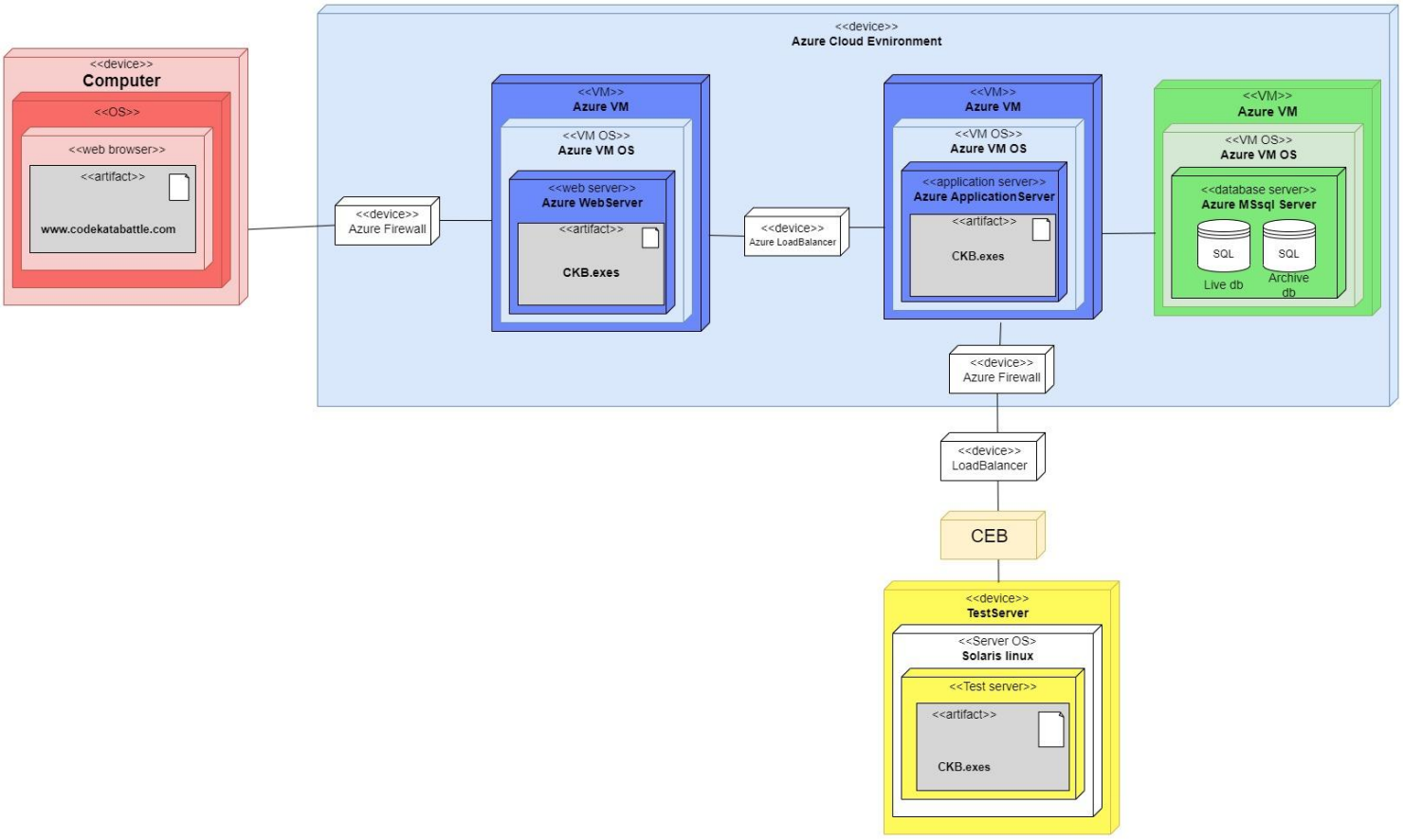



*Figure 2.7 - CKB Deployment view*

- **Client**
  Starting at the client level, users interact with the system through a standard computing device equipped with an operating system and web browser. The browser acts as the portal to the CodeKataBattle platform, accessed via the URL 'www.codekatabattle.com'. This setup represents the typical end-user's gateway to web-based applications, allowing for a familiar and accessible user experience.

- **Azure Cloud Environment**
  Within the Azure Cloud Environment, the platform's backend infrastructure is supported by a suite of Azure services, taking advantage of Azure's Platform as a Service (PaaS) offerings. The deployment utilizes Azure Firewalls at the ingress points to ensure secure and controlled access to the web and application servers, safeguarding the system against potential cyber threats and unauthorized access. To efficiently manage network traffic, an Azure LoadBalancer is employed, distributing incoming requests across available servers. This not only prevents overloading but also ensures high availability and fault tolerance of the web services.

- **Web Server**
  The web server is encapsulated within an Azure Virtual Machine, which operates on an Azure-specific operating system. This server hosts the web-facing components of the CodeKataBattle platform, and it's likely that the 'CKB.ear' artifact housed here is the deployable package of the web application, containing all necessary web resources.

- **Application server**
  The application server, also residing on an Azure Virtual Machine, is tasked with executing the business logic of the platform. It interacts with both the web server and the data layer, processing user actions and serving the dynamic content accordingly. The 'CKB.ear' artifact on this server signifies the enterprise application archive, encapsulating the business logic and backend operations of the platform.
- **Database server**
  For data storage and management, a dedicated Azure Virtual Machine runs the Azure SQL Server service. It maintains two distinct databases: the 'Live db' for handling ongoing transactions and operations, and the 'Archive db' for storing historical data, ensuring efficient data retrieval and performance optimization.
- **CEB & Test Server**
  A Custom Evaluation Bridge (CEB) serves as the communication liaison between the Azure-hosted components and the proprietary Test Server, which is not hosted on Azure but on a separate, secure server environment running Solaris Linux OS. This server's primary function is to evaluate and test the code submissions from the battles. It operates a version of the application encapsulated in the 'CKB.ear' artifact, which is

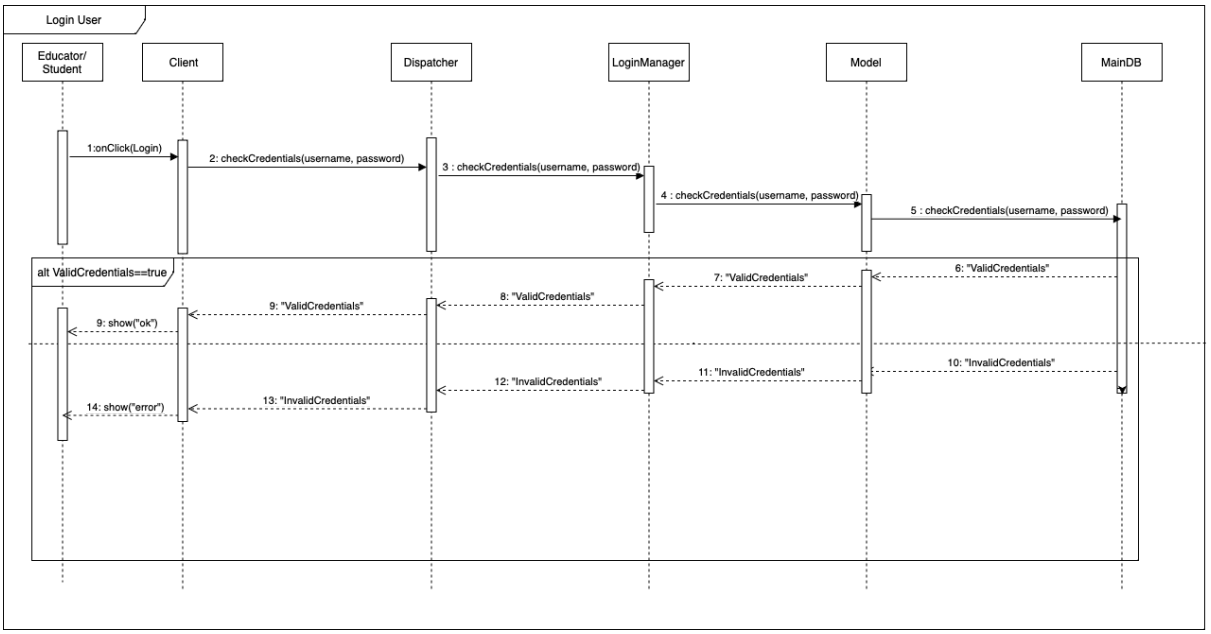
likely tailored for testing purposes, maintaining the fidelity of the evaluations while operating independently from the main application's runtime environment.

This structured deployment model employs Azure's robust and scalable infrastructure for the core application and data management while utilizing a specialized and secured environment for the testing and evaluation of user-submitted code, ensuring that the platform's performance is optimized across all operations.

## 2.4 Runtime views

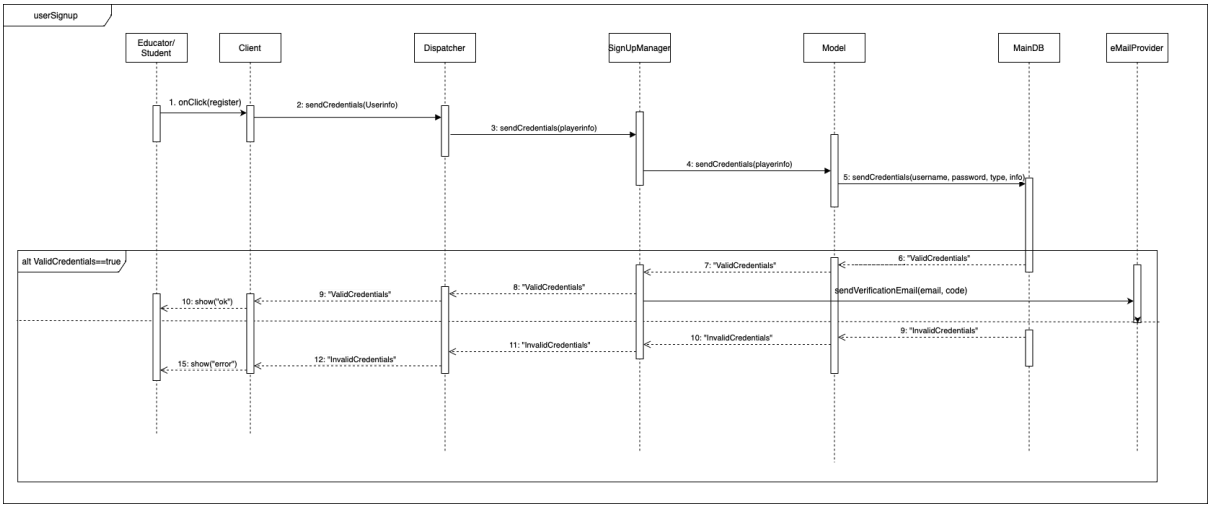
In this chapter are presented all the runtime views associated with the use cases described in the RASD relative to CKB. Runtime views show the interactions between the various components to carry out the functionalities offered by CKB platform.

[UC1]- User login to the platform



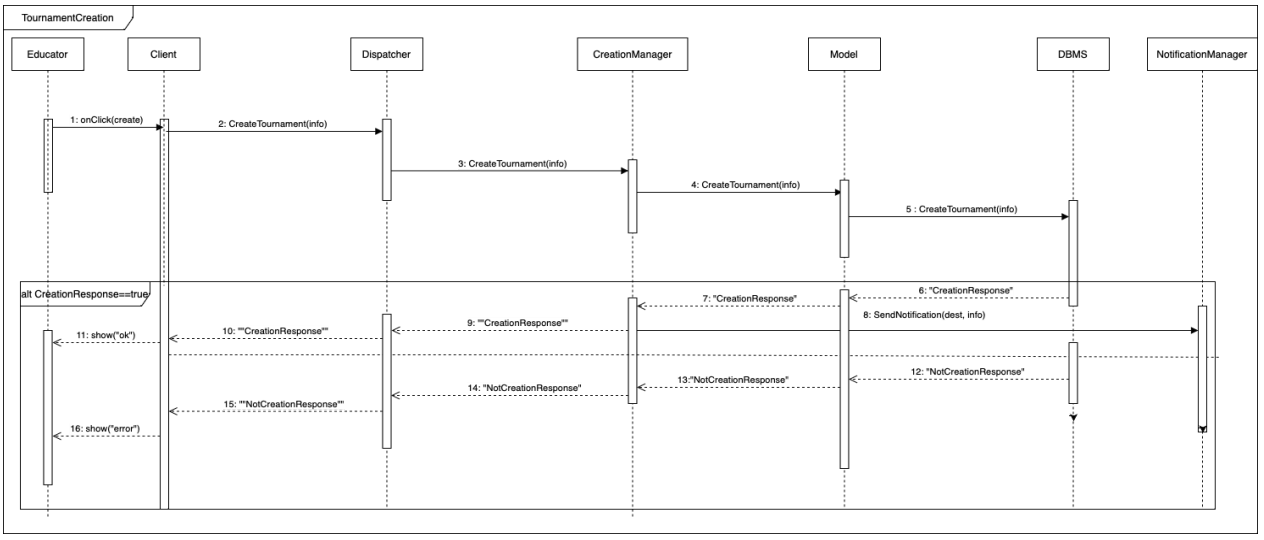

When a user clicks the login button, the credentials goes through the dispatcher, which forwards the request to the loginManager, then goes through the model to arrive at the Database where are checked.
At the end the message of invalid or invalid credential goes back to the client, that have different behaviors, depending on the response.

[UC2]- User Sign up to the platform




When a user wants to register to the platform, it inserts the credentials, chooses the type of user and inserts the relative info. Then The credentials goes through the dispatcher, which forwards the request to the SignUpManager, then goes through the model to arrive at the Database to check the uniqueness.
At the end the message of valid or invalid credential goes back to the client, which has different behaviors, depending on the response.
in the same the SignupManager call the email provider to send the verification email

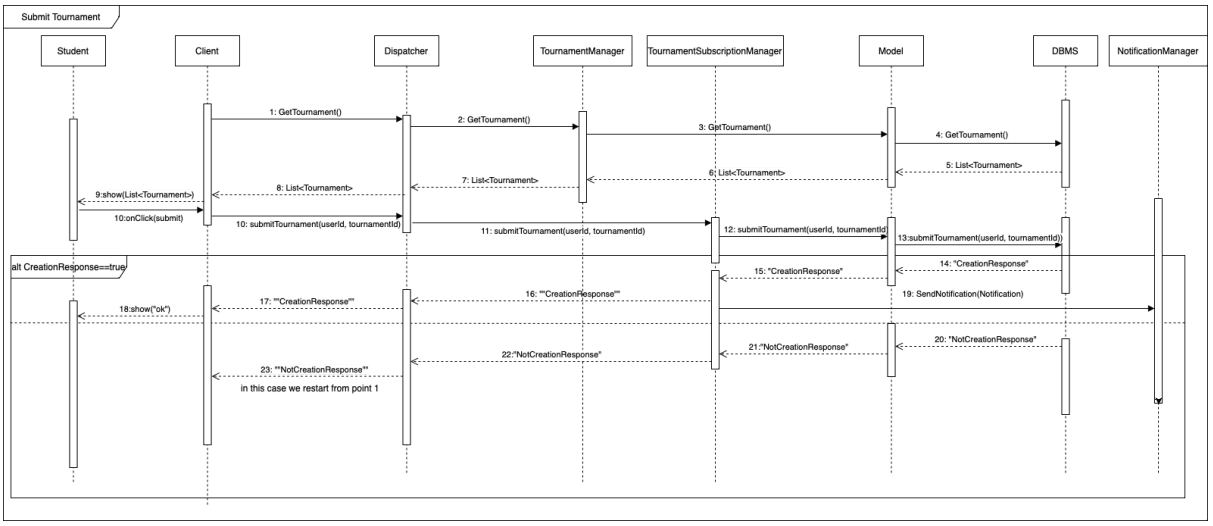
[UC3]- Educator Create a Tournament




When an Educator wants to create a Tournament, it inserts the relative info and then clicks the create button. Then The info goes through the dispatcher, that forwards the request to the CreationManager, subcomponent of

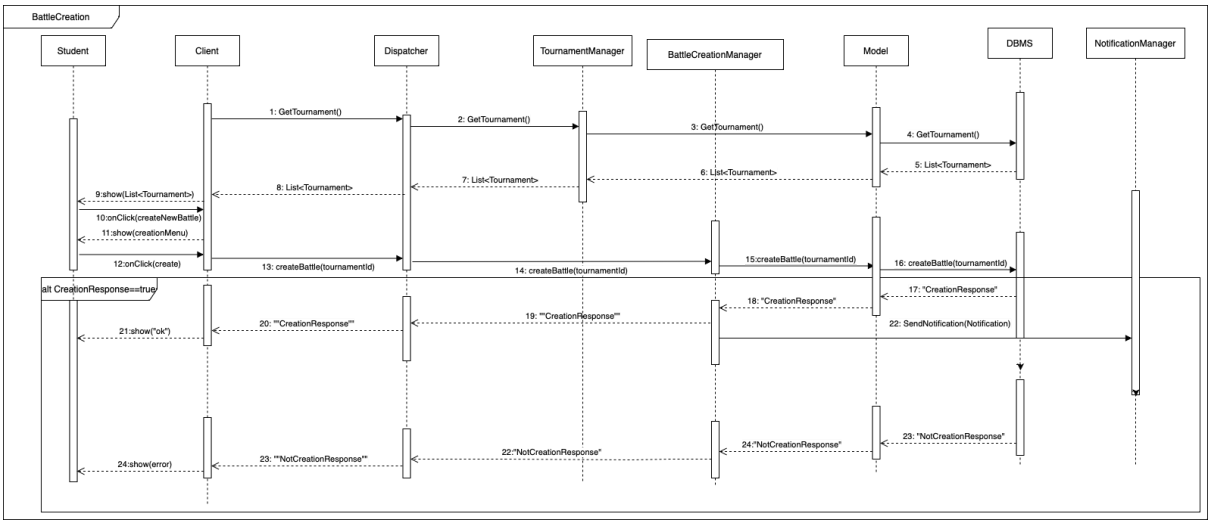
TournamentManager then goes through the model to arrive at the Database, where the new tournament is inserted.
At the end the message to ensure the creation of credentials goes back to the client, which displays the outcome.
in the same the CreationManager calls the NotificationManager  to send the notification about the new tournament.

[UC4]- student submit to a Tournament



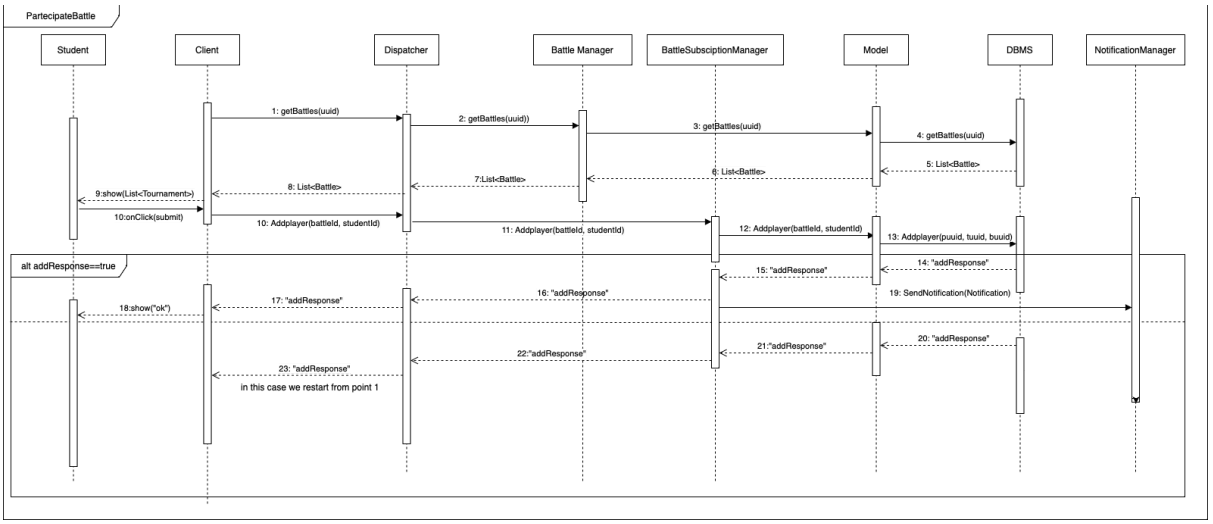

When the student goes to the homepage, it sees the list of  tournaments(the list was previously requested to the databases and sent back to the client by the Tournamentmanager).
With the list the student can choose the tournament he wants to participate in and then click on it to participate. The request goes through the dispatcher, that forwards the request to the TournamentSubsciptionManager, subcomponent of TournamentManager then goes through the model to arrive at the Database, where the participation is inserted.
At the end the message to ensure the insertion in the tournament o goes back to the client, that displays the outcome.
in the same the CreationManager calls the NotificationManager  to send the notification about the new participant.

[UC5]- Educator Create a Battle



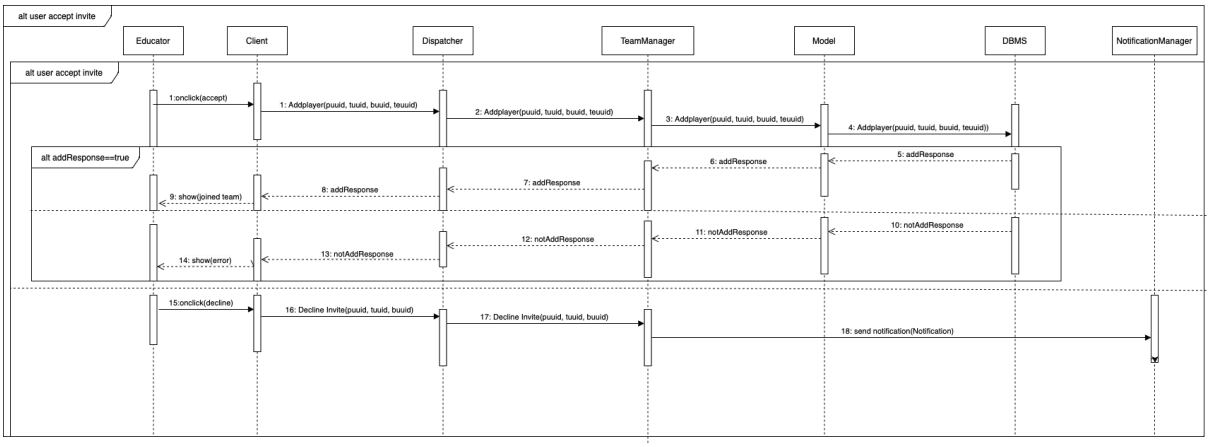

When an Educator wants to create a Battle, at the start, it needs to choose the tournament to put the battle in(the list was previously requested to the databases and sent back to the client by the Tournamentmanager).
Then it inserts the relative info and then clicks the create button. Then The info goes through the dispatcher, which forwards the request to the CreationManager, subcomponent of BattleManager then goes through the model to arrive at the Database, where the new Battle is inserted.
At the end the message to ensure the creation of credentials goes back to the client, which displays the outcome.
In the same the CreationManager calls the NotificationManager  to send the notification about the new tournament.

[UC6]- Student joins a battle



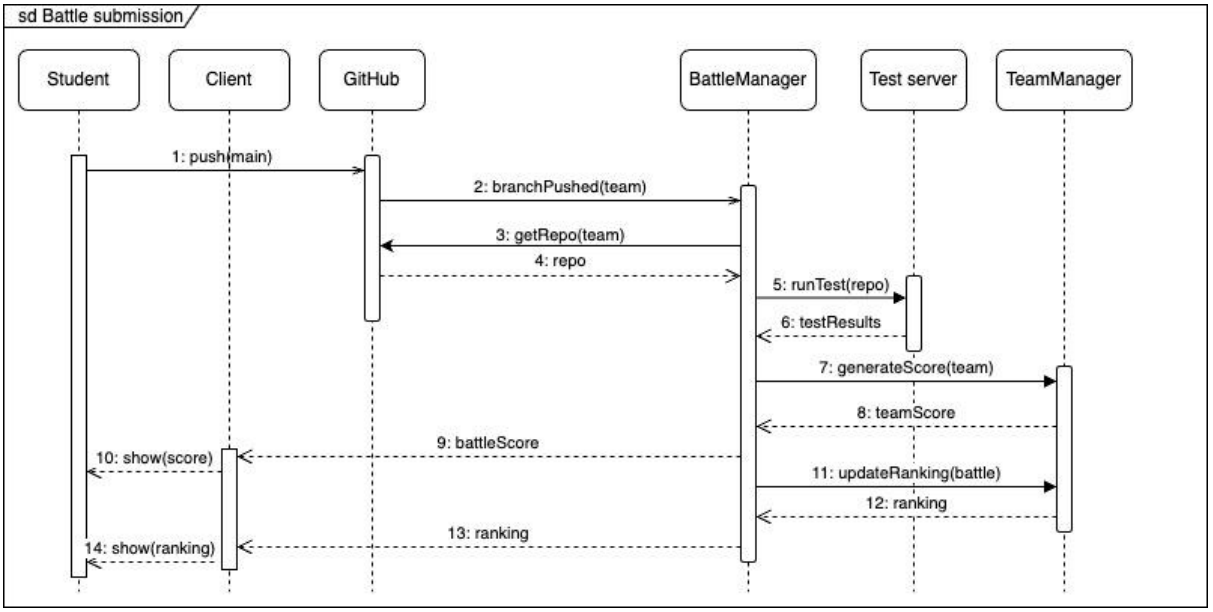
When the student goes to the page of a tournament, it sees the list of  Battles (the list was previously requested to the databases and sent back to the client by the BattleManager).
With the list the student can choose the battle he wants to participate in and then click on it to participate. The request containing the info of tournament and the player  goes through the dispatcher, that forwards the request to the TeamManager, subcomponent of BattleManager then goes through the model to arrive at the Database, where the participation is inserted.
At the end the message to ensure the insertion in the tournament goes back to the client, which displays the outcome.
in the same the CreationManager calls the NotificationManager  to send the notification about the new participant.

[UC7]- Student Manage invitation for a team




When a student receives an invitation to a team, it can decide if to accept or not.
If it accepts the request containing the info of tournament, the team, the player and the battle goes through the dispatcher, that forwards the request to the TeamManager, subcomponent of BattleManager then goes through the model to arrive at the Database, where the participation is inserted.
At the end the message to ensure the insertion in the tournament goes back to the client, which displays if an error occurs.
Otherwise if the invitation is declined, the battle manager is called through the dispatcher by the client to send the notification about the rejection.

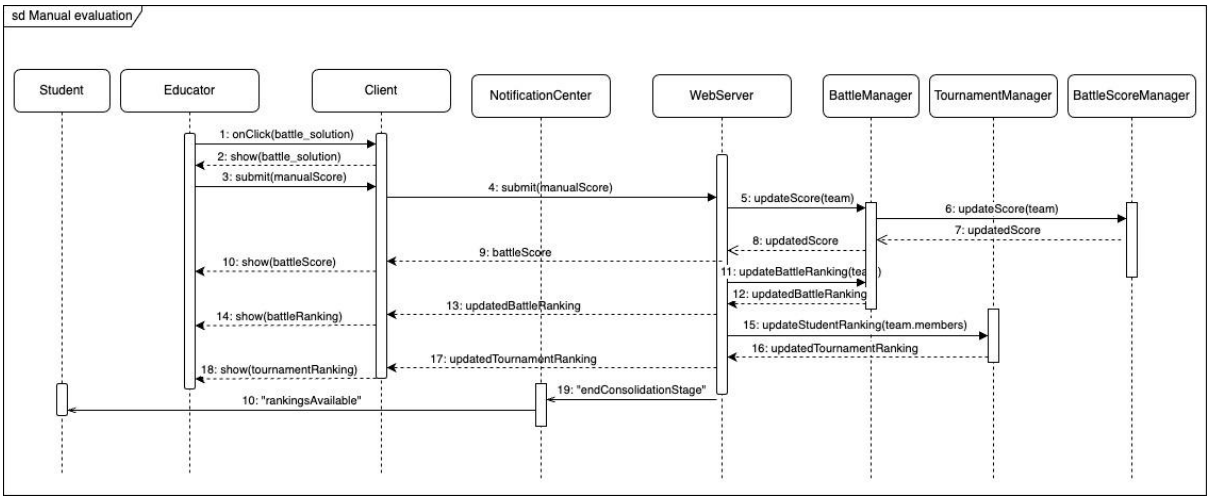
[UC8] - Student submits solution to a battle




In the process of a student submitting a solution to a battle, the student initiates the action by pushing the solution to GitHub. GitHub, in response, sends a notification (branchPushed) to the Battle Manager. The Battle Manager then communicates with GitHub to obtain the repository (getRepo). Subsequently, the Battle Manager triggers the testing process by sending a runTest request to the Test Server.

Once the testing is completed, the Battle Manager sends a generateScore request to the Team Manager. The Team Manager calculates the battle score and updates the ranking. Finally, the Battle Manager informs the client of the student by sending both the battle score and ranking, providing a comprehensive overview of the submission's performance in the battle.

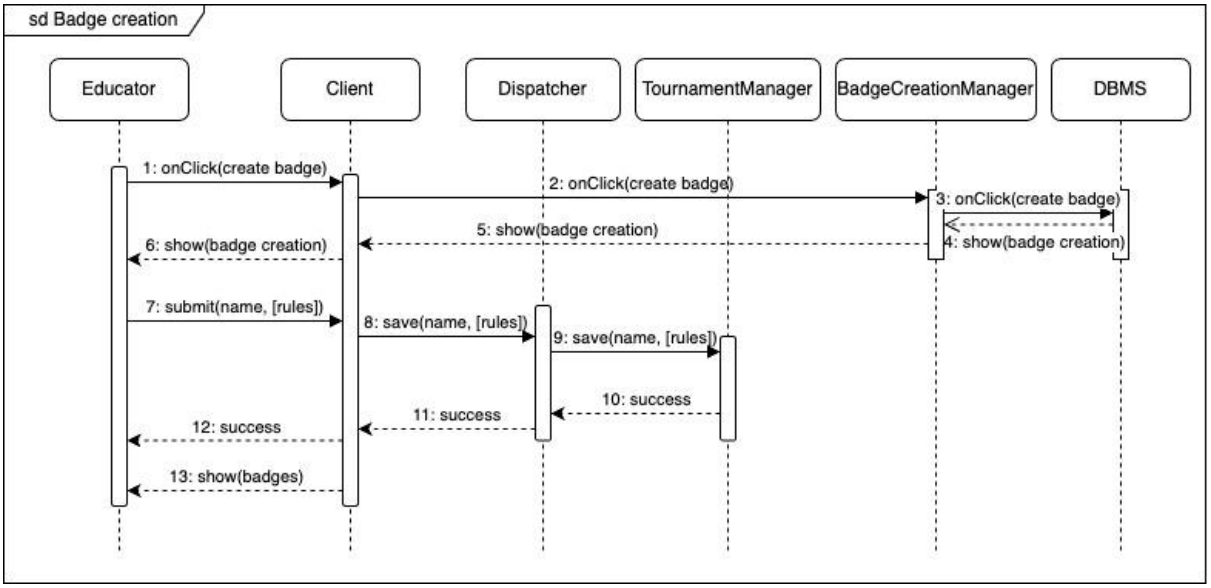
[UC9] - Educator manually evaluates student submissions




In the process of manually evaluating student submissions, the Educator initiates the sequence by clicking on the battle solution in the Client. Subsequently, the Educator submits a manual score through the Client. This manual score submission triggers a sequence of events: the Client communicates with the Dispatcher of the WebServer, conveying the manual score, and the WebServer updates the score for the corresponding team in the BattleManager. The updated score is then sent to the BattleScoreManager, which acknowledges the update and communicates it back to the BattleManager. The BattleManager, in turn, informs the WebServer about the updated score.
Simultaneously, the BattleScore is presented to the Educator through the Client. The WebServer proceeds to update the battle ranking for the team in the BattleManager. Upon receiving the updated battle ranking, the WebServer notifies the Educator through the Client.
Additionally, the WebServer updates the student ranking for the team members in the TournamentManager. The TournamentManager, in response, communicates the updated tournament ranking back to the WebServer. This updated tournament ranking is then sent to the Educator through the Client.

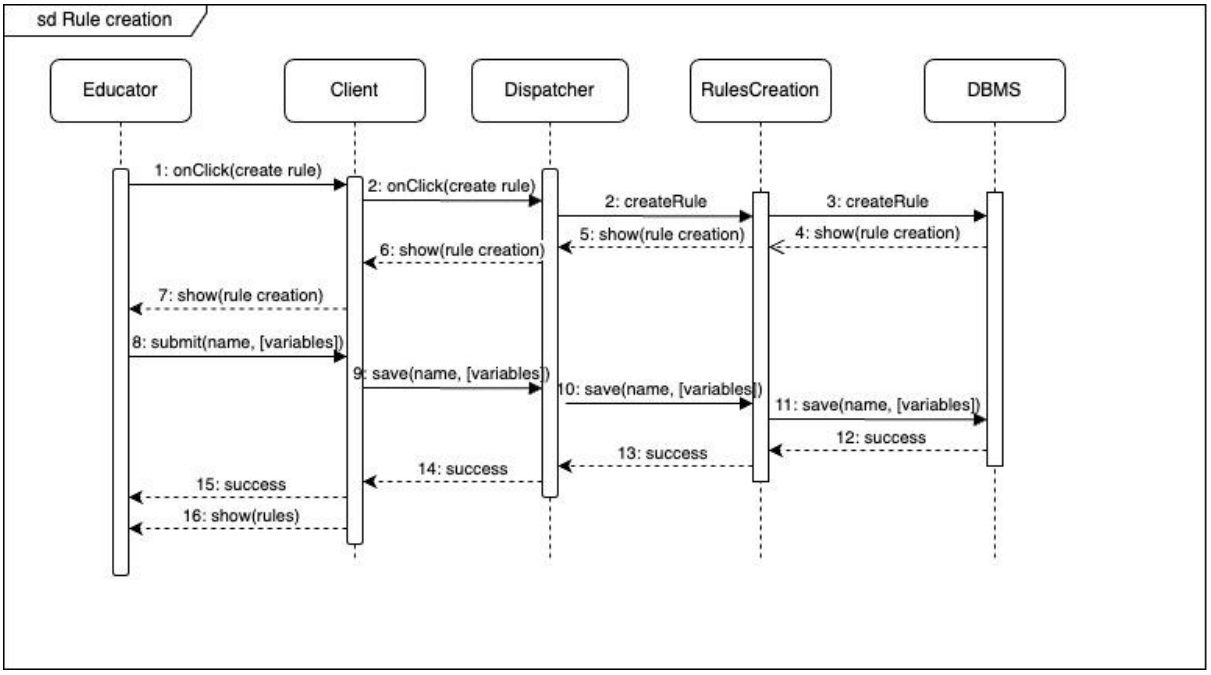
To conclude this process, the WebServer signals the end of the consolidation stage to the NotificationCenter. The NotificationCenter, in response, sends a notification of available rankings to the students involved in the battle.

[UC10] - Educator creates badges



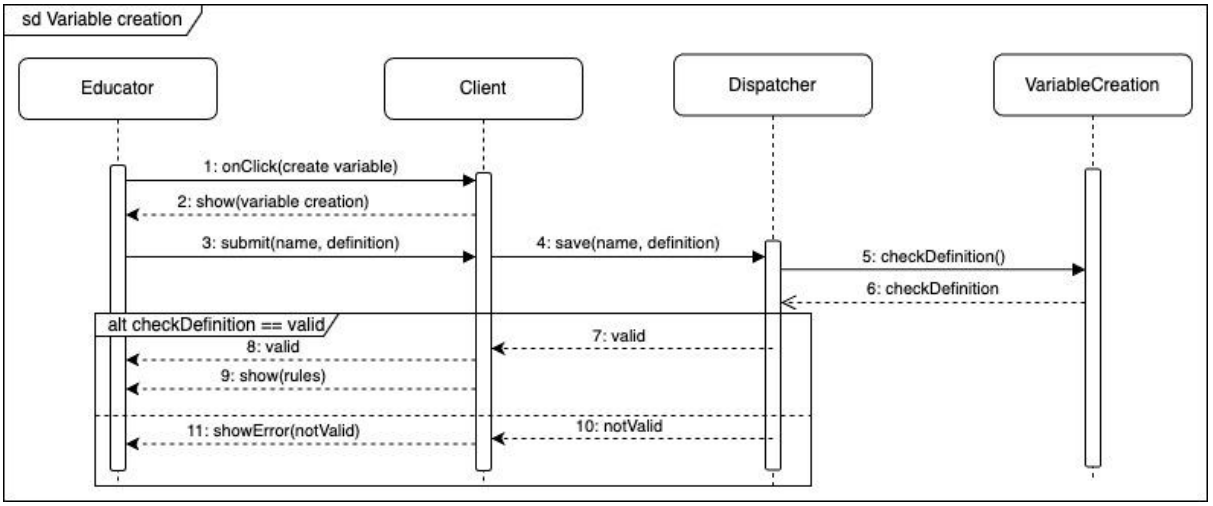

In the process of creating badges, the educator initiates the action by clicking on "Create Badge" in the client interface. The dispatcher then forwards this request to the Badge Creation Manager. The Badge Creation Manager interacts with the database (DBMS) to retrieve badge creation information, like the already existing rules, which is then displayed to the educator through the client. The educator proceeds to submit badge details, including rules, through the client. This information is saved by the client and dispatched to the Tournament Manager through the dispatcher. The Tournament Manager processes the request and responds with a success acknowledgment. Finally, the client displays the updated list of badges to the educator.

[UC11] - Educator creates rules




In the process of creating a rule on the CKB platform, an Educator initiates the action by clicking "create rule." This request is sent to the client. The client, in turn, communicates with the system's dispatcher, forwarding the request to the RulesCreation component. The dispatcher routes the request appropriately. Upon receiving the instruction, the RulesCreation component interacts with the database management system (DBMS) by sending an "onClick(create rule)" event. The DBMS responds by showing the rule creation interface, relaying the information back to RulesCreation.
This interaction continues as the RulesCreation component communicates the updated interface back through the dispatcher to the client. The client, after receiving the rule creation display, relays it to the Educator.
Next, the Educator submits the rule details, initiating an "onClick(submit)" event. The client processes this submission by sending a "save" event to the dispatcher. The dispatcher, in turn, forwards the save event to the RulesCreation component and subsequently to the DBMS.
Upon successfully saving the rule information, the DBMS notifies RulesCreation, which then communicates the success back to the client. The client responds to this success by sending a "show(rules)" event to the dispatcher, eventually reaching the Educator.

[UC12] - Educator creates variables

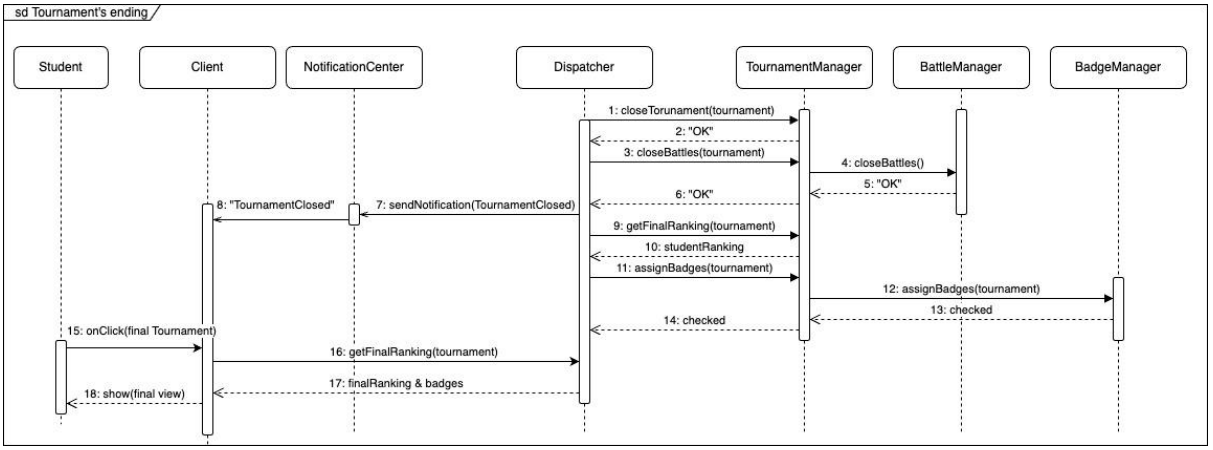



In the process of an educator creating variables, the educator initiates the action by sending an onClick(create variable) request to the client. The client responds by showing the variable creation interface to the educator. The educator then submits the variable details, including its name and definition, to the client. The client, in turn, sends a save(name, definition) event to the dispatcher.

The dispatcher, upon receiving the request, forwards it to the VariableCreation component by sending a checkDefinition() message. VariableCreation then evaluates the definition. If the definition is valid, the dispatcher informs the client by sending a valid response. The client, in this case, proceeds to show the updated list of rules to the educator.

However, if the definition is not valid, the dispatcher notifies the client with a notValid response. The client, in turn, sends an error display message (showError(notValid)) to the educator, indicating that there is an issue with the submitted variable definition.

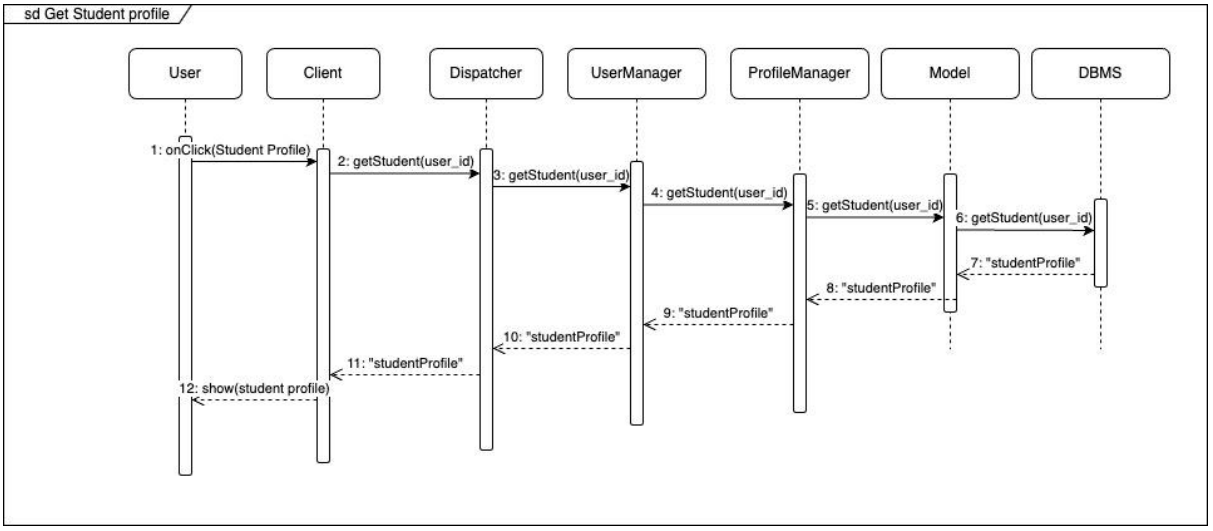
[UC13] - Tournament ends




This sequence of events encapsulates the conclusion of a tournament within the CodeKataBattle system. In the culmination of a tournament, the system orchestrates a series of coordinated actions. The Dispatcher initiates the process by sending a close Tournament(tournament) request to the TournamentManager. Upon successful closure, the Dispatcher proceeds to close the associated battles by sending a close Battles(tournament) request to the TournamentManager. The TournamentManager, in turn, communicates with the BattleManager, issuing a closeBattles() command to conclude the battles.

Simultaneously, the Dispatcher informs the NotificationCenter to send a TournamentClosed notification to the clients. The clients receive and acknowledge this notification. Following these steps, the Dispatcher seeks to gather the final ranking by issuing a getFinalRanking(tournament) request to the TournamentManager. Meanwhile, the Tournament Manager handles badge assignments by sending an assignBadges(tournament) request to the BadgeCreationManager, ensuring that all badges are duly checked and allocated.The clients, eager to view the conclusive results, interact with the system by sending an onClick(final tournament) event to the client. In response, the client issues a getFinalRanking(tournament) request to the Dispatcher, which promptly responds with the final ranking and badge information.

[UC14] - Get students profile




In the process of retrieving a student's profile, the user initiates the action by clicking on "student profile." This request is then conveyed through the client, which interacts with the dispatcher to route the request appropriately. The usermanager takes charge of managing user-related functionalities, guiding the request to the profile manager. Within the profile manager, the model interacts with the DBMS to retrieve the student profile. The outcome of this operation, the student profile, is then sent back through the same components in reverse order, reaching the user as the final response.

## 2.5 Component interfaces

This section lists all the methods that each component interface provides to the other components.

NotificationI

| Name | Parameters | Returns | Notes |
|---|---|---|---|
| sendNotification() | notification: Notification | Boolean | Sends a notification. Returns true if the notification is successfully sent, false otherwise |

DispatcherI

| Name | Parameters | Return | Notes |
|---|---|---|---|

| | | | |
|---|---|---|---|
| routeRequest() | httpRequest: HttpRequest | HttpResponse | Routes an incoming HTTP request to the appropriate component based on the request type and content |
| handleException() | exceptionData: ExceptionData | HttpResponse | Handles exceptions that may occur during request processing and generates an appropriate HTTP response |

CEB

| Name | Parameters | Return | Notes |
|---|---|---|---|
| runTest() | repo: Repo | testResults: List<Test> | Runs the tests of a team's repo |

EmailProviderI

| Name | Parameters | Return | Notes |
|---|---|---|---|
| sendVerificationEmail() | email: String verificationCode: String | Boolean | Sends a verification email to the provided email address with the given verification code. Returns true if the email is sent successfully, false otherwise |
| sendGeneralEmail() | email: String subject: String body: String | Boolean | Sends a general email with the specified subject and body to the provided email address. Returns true if the email is sent successfully, false otherwise |

LoginManagerI

| Name | Parameters | Return | Notes |
|---|---|---|---|
| logout() | None | None | Logs out the currently logged-in user |
| checkCredentials() | username: String password: String | Boolean | Checks credentials during the login process |
| isAuthenticated() | user: UUID | Boolean | Checks if the user is |

| | | | |
|---|---|---|---|
| | | | authenticated |

SignUpManagerI

| Name | Parameters | Return | Notes |
|---|---|---|---|
| sendCredentials() | playerinfo:PlayerInfo | Boolean | Manages the signup process for new users, returns `true` if successful, `false` otherwise |
| getEmail() | void | string | Getter method for retrieving the user's email |
| setEmail() | email: string | void | Setter method for updating the user's email |
| getPassword() | void | String | Retrieves the password |
| setPassword() | password: String | void | Sets the password |
| createVerificationEmailCode() | code: string | Boolean | Creates a verification code, returns `true` if successful, `false` otherwise |
| checkDataValidity() | user: User | Boolean | Checks the validity of the provided user data. Returns true if the data is valid, false otherwise |
| processVerificationCode() | verificationCode: String | Boolean | Processes the verification code received from the user. Returns true if the verification is successful, false otherwise |
| createUserAccount() | user: User | Boolean | Creates a new user account based on the provided user data. Returns true if the account creation is successful, false otherwise |

ProfileManagerI

| Name | Parameters | Return | Notes |
|---|---|---|---|
| viewStudentProfile() | studentID: UUID | StudentProfile | Retrieves and displays a student's profile |
| getStudentProfile() | studentID: UUID | StudentProfile | Retrieves the profile information of a specific student |
| getBadges() | studentID: UUID | List<Badge> | Retrieves the badges earned by a specific student |
| addBadge() | studentID: UUID<br>badge: Badge | Boolean | Adds a new badge to the profile of a specific student. Returns true if the addition is successful, false otherwise |
| removeBadge() | studentID: UUID<br>badge: Badge | Boolean | Removes a badge from the profile of a specific student. Returns true if the removal is successful, false otherwise |
| getTournamentHistory() | studentID: UUID | List<Tournament> | Retrieves the tournament history of a specific student |
| addTournamentHistory() | studentID: UUID<br>tournament: UUID | Boolean | Adds a new entry to the tournament history of a specific student. Returns true if the addition is successful, false otherwise |
| getStudentID() | void | UUID | Retrieves the student ID of the currently managed profile |

TournamentManagerI

| Name | Parameters | Return | Notes |
|---|---|---|---|
| getTournaments() | void | List<Tournament> | Return the list of visible tournament |
| updateStudentRanking() | teamMembers: List<Student> | Ranking: List<Student> | Return the ranking of students in a tournament |
| saveBadge() | tournament: UUID<br>name: String<br>rules: List<Rule> | Boolean | Saves a badge in a tournament |

| | | | |
|---|---|---|---|
| closeTournament() | tournament: UUID | Boolean | Closes a tournament |
| closeBattles() | tournament: UUID<br>battles: List<Battle> | Boolean | Closes one or more battles inside a tournament |
| getFinalRanking() | tournament: UUID | studentRanking: List<Student> | Returns the final ranking of students in a tournament |
| assignBadges() | tournament: UUID | void | Checks if there are badges to assign to students |

TournamentCreationManagerI

| Name | Parameters | Return | Notes |
|---|---|---|---|
| createTournament() | info: TournamentData | Boolean | Initiates the creation of a new tournament. Returns true if the creation is successful, false otherwise |
| updateTournament() | tournamentID: UUID<br>updatedData: TournamentData | Boolean | Updates the information of an existing tournament. Returns true if successful, false otherwise |
| setTournamentBadges() | tournamentID: UUID<br>badges: List<Badge> | Boolean | Sets the badges for a specific tournament. Returns true if successful, false otherwise |
| getTournamentBadges() | tournamentID: UUID | badges: List<Badge> | Gets the badges for a specific tournament |
| setTournamentStatus() | tournamentID: UUID<br>newStatus: String | Boolean | Updates the status of a tournament (e.g. "Active"). Returns true if successful, false otherwise |
| deleteTournament() | tournamentID: UUID | Boolean | Deletes a specific tournament. Returns true if successful, false otherwise |

TournamentSubscriptionManager

| Name | Parameters | Return | Notes |
|---|---|---|---|
| submitTournament() | studentID: UUID TournamentId:uuid | bool | Insert a student inside a tournament. Returns true if successful, false otherwise |

TournamentScoreManagerI

| Name | Parameters | Return | Notes |
|---|---|---|---|
| getTeamScores() | teamID: UUID | int | Retrieves the score of a specific team |
| getAllTeamScores() | tournamentID: UUID | List<TeamScore> | Retrieves scores of all teams in a given tournament |
| updateTeamScore() | teamID: UUID newScore: int | Boolean | Updates the score of a specific team. Returns true if successful, false otherwise |
| getStudentRank() | studentID: UUID | int | Retrieves the rank of a specific student in a tournament |
| getTournamentRank() | tournamentID: UUID | List<Student> | Retrieves the ranking of the tournament |
| getStudentTournamentScore() | studentID: UUID tournament: UUID | int | Retrieves the total score of a student in a tournament |
| getStudentTotalScore() | studentID: UUID | int | Retrieves the total score of a student across all tournaments |

TeamManagerI

| Name | Parameters | Return | Notes |
|---|---|---|---|
| generateScore() | teamID: UUID | teamScore: int | Calculates the score of a team |
| updateRanking() | battleID: UUID | ranking: List<Team> | Calculates the ranking of the teams in a battle |

BadgeManagerI

| Name | Parameters | Returns | Notes |
|---|---|---|---|
| getBadgeInfo() | badge: Badge | BadgeInfo | Retrieves information about a specific badge |
| getAllBadges() | tournament: UUID | List<Badge> | Retrieves a list of all available badges |
| assignBadgeToStudent() | student: UUID, badge: Badge | Boolean | Assigns a specific badge to a student. Returns true if successful, false otherwise |
| getBadgeCriteria() | badge: Badge | String | Retrieves the criteria data associated with a specific badge |
| getBadgeRules() | badge: Badge | List<Rule> | Retrieves the rules associated with a specific badge |
| setBadgeRules() | badge: Badge rule: List<Rule> | Boolean | Sets rules for a specific badge. |
| createBadge() | title: String rules: List<Rule> | Boolean | Creates a new badge based on the provided badge data. Returns true if successful, false otherwise |

RulesCreationI

| | | | |
|---|---|---|---|
| createRule() | name: String variable: List<Variable> | Boolean | Creates a new rule based on the variables available. Returns true if successful, false otherwise |

VariableCreationI

| | | | |
|---|---|---|---|
| createVariable() | name: String | Boolean | Creates a new variable |

BattleManagerI

| Name | Parameters | Returns | Notes |
|---|---|---|---|
| getBattles() | tournamentId: uuid | List<Battle> | Return the list of visible battles within a tournament |

| | | | |
|---|---|---|---|
| battleScore() | team: Team | score: int | Returns the score of a team in a battle |
| ranking() | team: Team | ranking: int | Returns the ranking of a team in a battle |
| updateScore() | team: Team | score: int | Updates the score of a team in a battle |
| updateBattleRanking() | team: Team | ranking: int | Updates the ranking of a team in a battle |

NotificationCenterI

| Name | Parameters | Returns | Notes |
|---|---|---|---|
| notifyTournamentStart() | tournamentID: UUID | void | Notifies the NotificationCenter about the start of a tournament |
| notifyTournamentEnd() | tournamentID: UUID | void | Notifies the NotificationCenter about the end of a tournament |
| notifyBattleStart() | battleID: UUID | void | Notifies the NotificationCenter about the start of a battle |
| notifyBattleEnd() | battleID: UUID | void | Notifies the NotificationCenter about the end of a battle |
| notifyBadgeAssignment() | studentID: UUID<br>badge: Badge | void | Notifies the NotificationCenter about the assignment of a badge to a student |

BattleSubscriptionManagerI

| Name | Parameters | Returns | Notes |
|---|---|---|---|
| AddPlayer() | battleID: UUID, studentID: UUID | Boolean | Subscribes a student to a specific battle. Returns true if successful, false otherwise |
| getBattleSubscribers() | battleID: UUID | List<Student> | Retrieves a list of students subscribed to a specific battle |

| | | | |
|---|---|---|---|
| addStudentToTeam() | team: TeamId<br>studentID: UUID<br>BattleId: UUID | Boolean | Subscribes a student to a specific team in a battle |

BattleScoreManagerI

| Name | Parameters | Returns | Notes |
|---|---|---|---|
| submitTeamScore() | teamID: UUID<br>score: int | void | Submits the score for a specific team in the battle |
| getTeamScore() | teamID: UUID | score: int | Retrieves the score of a specific team in the battle |
| updateBattleRanking() | battleID: UUID | Boolean | Updates the ranking of teams in the battle based on their scores |
| getTeamRanking() | teamID: UUID | rank: int | Retrieves the ranking of a specific team in the battle |
| getAllTeamRankings() | battleID: UUID | List<Team> | Retrieves rankings of all teams in the battle |
| updateScore() | team: Team | score: int | Updates the score of a team in a battle |

BattleCreationManagerI

| Name | Parameters | Returns | Notes |
|---|---|---|---|
| createBattle() | codeKata: CodeKata<br>minTeamSize: int<br>maxTeamSize: int<br>registrationDeadline: Date<br>submissionDeadline: Date | Boolean | Initiates the creation of a new battle. Returns true if the creation is successful, false otherwise |
| setTest() | battleID: UUID<br>test: Test | Boolean | Adds a new test for a battle |
| setScoreFunction() | battleID: UUID<br>scoreFuntions: List<ScoreFunction> | Boolean | Sets the scoring function in order to assign automated evaluations |

GitHubActionsI

| Name | Parameters | Returns | Notes |
|---|---|---|---|
| branchPushed() | teamID: UUID | void | Triggers method for specific team pushes |
| getRepo() | teamID: UUID | repo: Repo | Returns the GitHub repository code of a team |

ModelI

| | | | |
|---|---|---|---|
| createTournament() | tournament: Tournament | Boolean | Initiates the creation of a new tournament. Returns true if the creation is successful, false otherwise |
| createBattle() | battle: Battle tournament: Tournament | Boolean | Initiates the creation of a new battle inside a tournament. Returns true if the creation is successful, false otherwise |
| enrollStudentInTournament() | student: Student tournament: Tournament | Boolean | Enrolls a student in a specific tournament. Returns true if successful, false otherwise |
| enrollStudentInBattle() | student: Student battle: Battle | Boolean | Enrolls a student in a specific battle. Returns true if successful, false otherwise |
| updateProfile() | student: Student | Boolean | Updates the profile of a student |
| createRule() | name: String variable: Variable | Boolean | Initiates the creation of a new Rule. Returns true if the creation is successful, false otherwise |
| createVariable() | name: String | Boolean | Initiates the creation of a new Variable. Returns true if the creation is successful, false otherwise |

LiveDBManagerI

| | | | |
|---|---|---|---|
| getTournament() | void | tournament: Tournament | Retrieves a tournament |

| | | | |
|---|---|---|---|
| storeTournament() | tournament: Tournament | void | Stores a tournament |
| getBattles() | void | battle: Battle | Retrieves a battle |
| storeBattle() | battle: Battle | void | Stores a battle |
| getBadges() | void | badge: Battle | Retrieves a badge |
| storeBadges() | badge: Battle | void | Stores a badge |
| getProfile() | void | profile: Student | Retrieves a profile |
| storeProfile() | profile: Student | void | Stores a profile |
| getRule() | void | rule: Rule | Retrieves a rule |
| storeRule() | rule: Rule | void | Stores a rule |
| getVariable() | void | variable: Variable | Retrieves a variable |
| storeVariable() | variable: Variable | void | Stores a variable |

ArchiveDBManagerI

| | | | |
|---|---|---|---|
| archiveTournament() | tournament: Tournament | void | Archives a tournament |
| archiveBattle() | battle: Battle | void | Archives a battle |

## 2.6 Selected architectural Styles and patterns

### 2.6.1 3-tier Architecture

In embracing a decision of selecting a 3-tier architecture, the CKB platform delineates its structure into the Client Tier, Server Tier, and Data Server, aligning with industry best practices for enhanced scalability, maintainability, and systematic separation of functional elements.

1. Client Tier: This outward-facing stratum encapsulates the user interface and client-side logic. Distinct from the backend operations, this lightweight tier ensures fluid communication with the Server Tier, prioritizing user interaction and responsiveness.
2. Server Tier: As the intermediary between user interactions and data management, the Server Tier is compartmentalized into the Web Server, Application Server, and Test Server. The Web Server manages HTTP requests, facilitating standardized communication with the Application Server through an API. The Application Server is positioned at the core and it is a RESTful application that executes business logic, processes user commands, and interacts directly with the Data Server. It incorporates the GitHub WebHooks interface for real-time synchronization, detecting new commits and calculating scores. The Test Server is dedicated to code compilation, execution, and evaluation, the Test Server ensures compliance with predefined criteria. The CodeEval Bridge (CEB) facilitates a custom interface for secure data exchange between the Test Server and Application Server.
3. Data Server: The Data Server encompasses the Main Database and Archive Database, serving distinct roles. The Main Database stores current and active data, handling real-time queries and transactions to maintain data integrity. The Archive Database manages historical data, optimizing storage and retrieval of older records through triggers.

This chosen 3-tier architecture underscores modularity, empowering each layer to operate autonomously and contributing to a robust framework for system growth.

### 2.6.2 Model View Controller pattern

For implementing the CKB platform was chosen to follow the MVC pattern:

1. The Model component within the MVC architecture of the CKB platform encapsulates the business logic, data management, and application state. It is responsible for:
    - Managing tournament-related data and functionalities
    - Managing battle-specific data and functionalities
    - Governing user-related operations
    - Incorporating logic for the badges
2. The View component represents the user interface and presentation layer of the CKB platform. It is responsible for:
    - Rendering the interface for Educators, displaying tournament and battle creation forms, as well as the overall platform dashboard
    - Displaying the interface for Students, providing access to ongoing tournaments, battles, and personal profiles
    - Managing the display of user profiles, showcasing individual achievements, tournament participation, and badges
3. The Controller serves as an intermediary between the Model and the View, handling user inputs, updating the Model, and refreshing the View accordingly. It is responsible for:
    - Managing user inputs related to tournaments
    - Handling user interactions with battles, including the creation of new battles, team formation, and submission processing.
    - Managing user-related and badge-related actions

The MVC pattern facilitates efficient development and updates by isolating components responsible for specific functionalities.

### 2.6.3 Facade pattern

The Dispatcher acts as the facade in the CKB platform, presenting a simplified interface for handling incoming requests. It serves as a gateway, directing requests to the appropriate components within the system. The Dispatcher encapsulates the intricacies of routing and dispatching, shielding clients from the complexities of the internal subsystems.

### 2.6.4 Event Based pattern

The CKB platform adopts the Event-Based Pattern to streamline communication and coordination among its diverse components. This pattern, commonly observed in modern development practices such as continuous integration/deployment (e.g., GitHub Actions), offers advantages like easy addition/deletion of components. However, it comes with potential scalability problems (bottlenecks) and the caveat that the ordering of events is not guaranteed.

This is how the Event-Based Pattern is manifested in key components:

- The Educator Client initiates events by interacting with the front-end, triggering HTTP calls to the Web Server.
- The Dispatcher plays a crucial role in managing incoming requests, translating them into events, and directing them to the appropriate components like UserManager, TournamentManager, BattleManager, etc.
- TournamentCreationManager, TournamentScoreManager, and BadgeManager operate based on events, responding to actions initiated by educators or changes in the system, such as the conclusion of a tournament or score updates.
- BattleCreationManager and BattleScoreManager rely on events to handle the initiation of battles and manage the scoring process. Educators trigger events, initiating the creation of new battles, while scores are updated based on events from GitHubWebHooks.
- The AutoTestManager is event-triggered by GitHubWebHooks, responding to new code submissions by students. This ensures that code evaluation is automatically initiated upon code submission events.
- GitHub WebHooks serve as external triggers, notifying the CKB platform of specific actions on student repositories, like new code submissions or changes. These events initiate subsequent processes within the system.
- Components interact with the Model, triggering events for data exchange. The Model ensures data integrity and conformity before passing it to the DBMS.
- Internal and external notifications are treated as events. The NotificationCenter acts as a hub, receiving notifications from various components and directing them to the intended recipients.

- The StudentClient triggers events by facilitating user actions, leading to interactions with components like TournamentManager, BattleManager, and GitHubActions.
- GitHubActions operates based on events defined in .yml files. These events automatically execute tasks after each repository push, ensuring code readiness for evaluation on the TestServer.

The Event-Based Pattern allows for easy integration and removal of components, fostering a modular and adaptable system. However potential scalability problems may arise, particularly if there's a high volume of events leading to bottlenecks. The ordering of events can be an issue in scenarios where strict sequence execution is crucial, but they are rare.

### 2.6.5 Role-Based Access Control pattern

The CKB platform employs the Role-Based Access Control (RBAC) pattern to regulate and manage access to its various components and interfaces based on the roles of users, primarily distinguishing between Educators and Students. The RBAC pattern ensures that each user, whether an Educator or a Student, interacts with the system through interfaces and components tailored to their specific roles, maintaining a secure and role-appropriate user experience.

## 2.7 Other design decisions

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

### 2.7.1 Availability

To ensure high availability and distribute incoming traffic efficiently, a load balancing mechanism is implemented across multiple instances of the Web Server and Application Server. This enhances system reliability by preventing a single point of failure and optimizing resource utilization. Employing database replication and backup strategies to ensure data availability. Redundant copies of the live database are maintained to handle potential failures, minimizing downtime and safeguarding against data loss. It is also expected to implement database replication and backup strategies. Redundant copies of the live database are maintained to handle potential

failures, minimizing downtime and safeguarding against data loss. In the end are implemented fault-tolerant mechanisms in event handling components. For critical components like GitHubWebHooks and NotificationCenter, redundant configurations and failover strategies are in place to handle system interruptions.

### 2.7.2 Data Storage

All real-time and actively used data, including active tournaments, battles, badges, rules, and variables, is stored in the Main Database. This ensures quick and efficient access for ongoing system operations. The Archive Database is optimized for historical data storage and retrieval, contributing to efficient system performance. A policy and triggers are used to move older data, such as tournaments and battles older than three months, from the Main Database to the Archive Database. To maintain data consistency are implemented transactional integrity operations, this includes utilizing transaction management mechanisms to ensure that database operations, especially moves to the archive, are executed reliably without compromising data integrity.

### 2.7.3 Design Choice of Badges

The inclusion of gamification badges in the CodeKataBattle (CKB) platform serves as a motivating and rewarding mechanism for individual students, recognizing their achievements and contributions. The design choice of badges is characterized by flexibility, allowing educators to define new badges and rules using a JSON-like language. This design decision aims to provide educators with a straightforward and expressive way to customize the badge system according to the unique requirements of each tournament.

The badge creation process involves educators specifying badges and their associated rules using a JSON-like format. This format is both human-readable and machine-parseable, ensuring ease of use and efficient processing by the CKB platform. Each badge is defined by a title and a set of rules:

- Title: Describes the name or label of the badge (e.g., "Tournament Participant," "Top Committer").
- Rules: A collection of conditions that must be met for a student to be eligible for the badge.

Rules are conditions that evaluate student actions or achievements. They are written in a structured format within the JSON-like language. Each rule consists of a name and a definition.

- Name: Describes the purpose or nature of the rule (e.g., "Participate in a Battle," "Be the Top Committer").
- Definition: Represents the logical condition that must be satisfied for the rule to be considered true.

Inside the field “Definition” of a rule, the variables that educators can use are formatted like Strings (see Figure 2.7.3), CKB platform will parse the “def” field of rules. These variables can be created by educators, through sql queries (see Figure 3.3), to capture essential metrics related to student participation and performance.
Figure 2.7.3 includes "tot_attended_battles," "tot_invite_sent," "tot_commits_student," and "max_tot_commits."

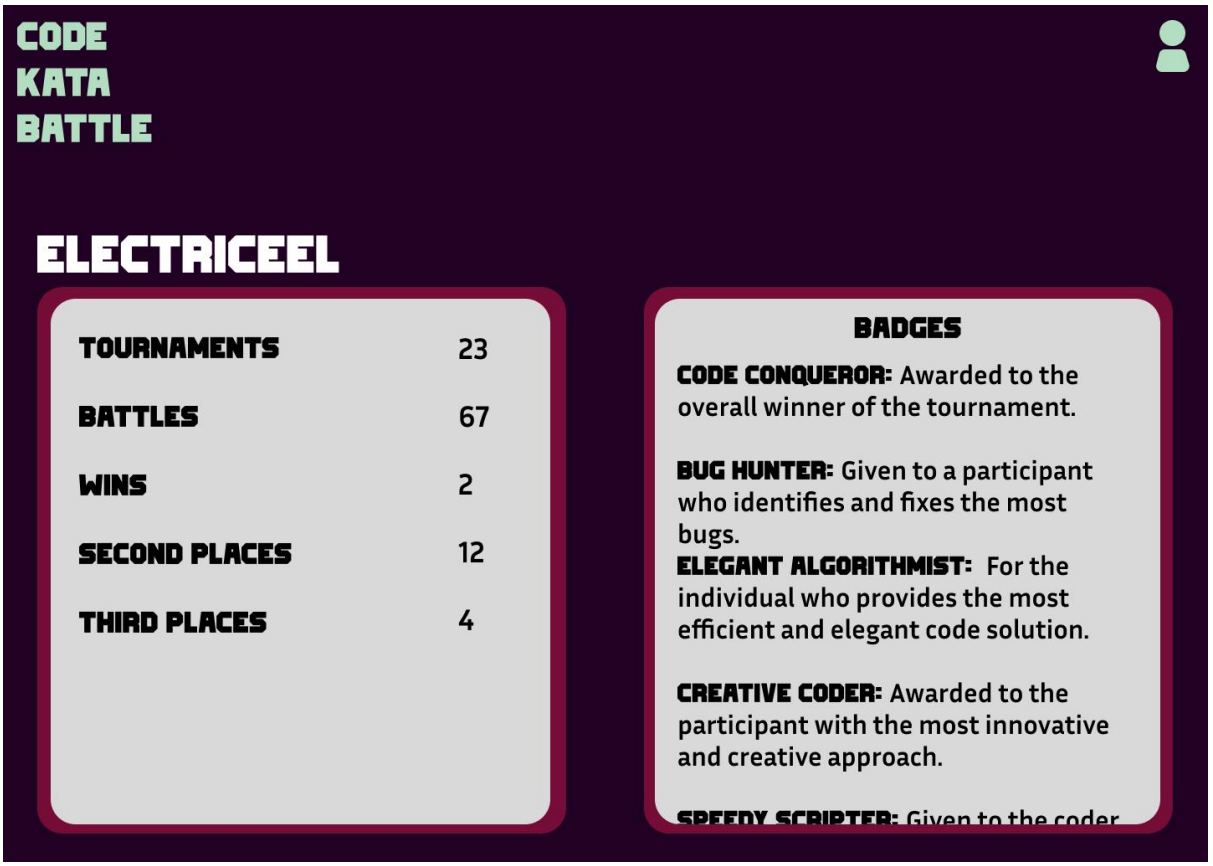
Educators have the flexibility to create new badges and define rules and variables tailored to the specific goals and characteristics of each tournament.


*(Figure 2.7.3) Example of defining badges when creating a tournament*

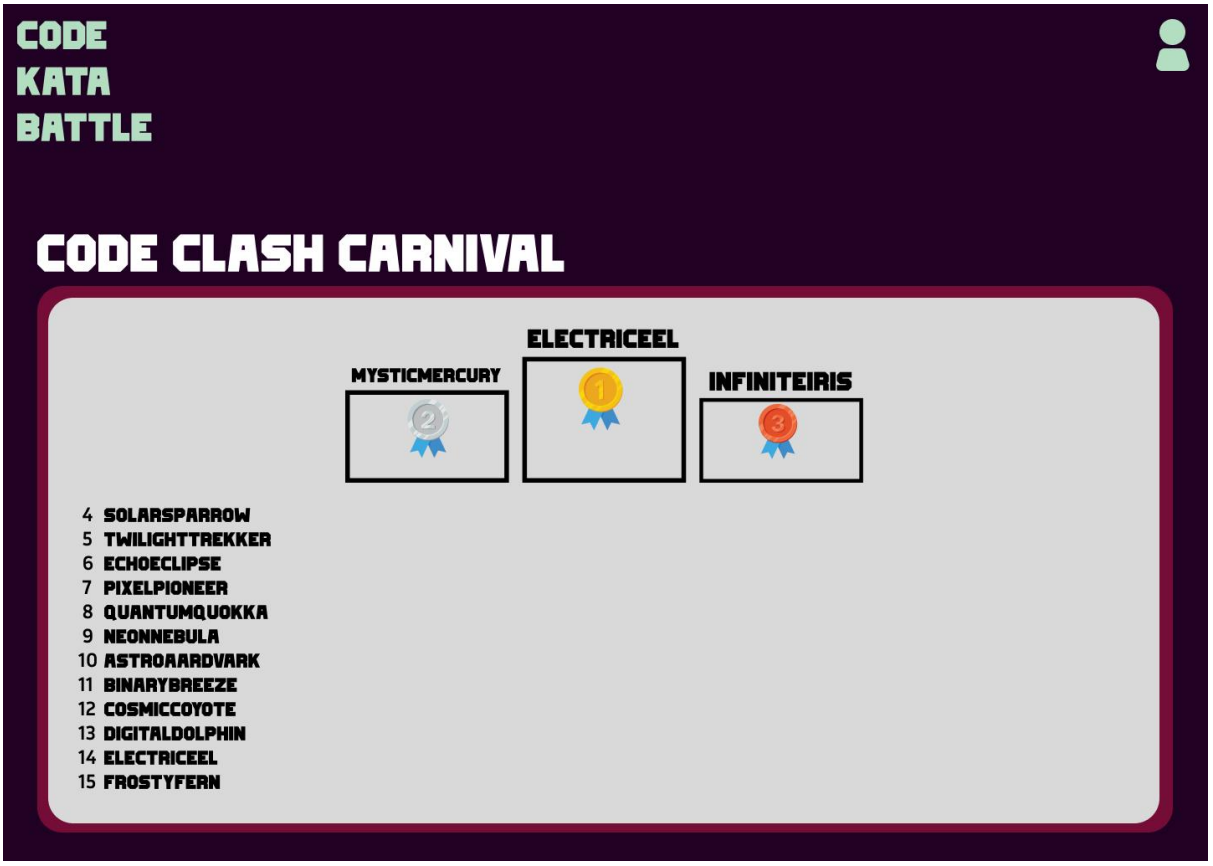
# 3. User Interface Design

In the RASD document, we provided detailed depictions of the primary user interfaces, which serve as the foundation for user interaction within the system. We encourage readers to refer to the RASD to familiarize themselves with these main interfaces. In addition to the primary interfaces, we have conceptualized three supplementary interfaces that address more specific or secondary functionalities within the platform.

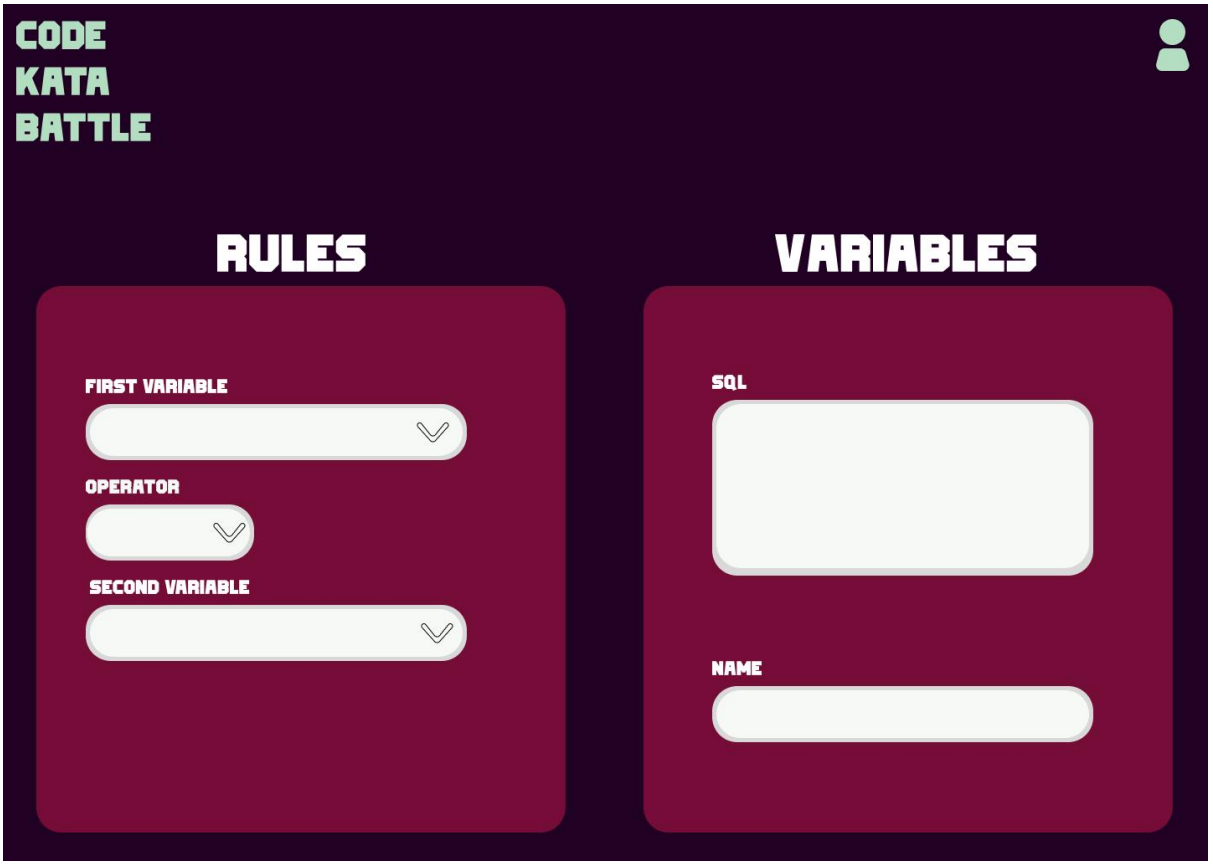
It is important to note that these user interface designs serve as guiding blueprints for the development team. They outline the essential features and layout that the developers are expected to implement. However, these interfaces are not static; they are subject to evolution and refinement. As the system's requirements evolve or as new needs emerge, the interfaces may be adapted accordingly. Similarly, input from UI/UX designers may lead to changes in the design to enhance user experience and interface efficiency.




*(Figure 3.1) User profile interface*




*(Figure 3.2) Tournament result interface (ranking of students)*




*(Figure 3.3) Rules and Variable creation interface*

# 4. Requirement traceability

In this section of the document it is explained, for each requirement defined in the RASD, what design elements are involved for its fulfillment.

| | |
|---|---|
| Requirements: | [R3] The platform allows already registered students to login<br>[R4] The platform allows already registered educator to login |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- LoginManager<br>- Model<br>• DataServer |

| | |
|---|---|
| Requirements: | [R1] The platform allows students to sign up<br>[R2] The platform allows educators to sign up<br>[R48] The platform allows the user to choose during registration whether to be a Student or an Educator |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- SignupManager<br>- Model<br>• DataServer<br>• eMail Provider |

| | |
|---|---|
| Requirements: | [R5] The platform allows educators to create a tournament<br>[R6] The platform allows educators to set a tournament topic<br>[R7] The platform allows educators to set a tournament deadline |

| | |
|---|---|
| | [R8] The platform allows educators to invite a collaborator to a tournament |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- TournamentManager<br>-TournamentCreationManager<br>- Model<br>• DataServer<br>• NotificationManager |

| | |
|---|---|
| Requirements: | [R9] The platform allows educators to create a new battle<br>[R10] The platform allows educators to upload a CK for a battle<br>[R11] The platform allows educators to set minimum number of students per team<br>[R12] The platform allows educators to set maximum number of students per team<br>[R13] The platform allows educators to set the language about a tournament<br>[R14] The platform allows educators to set the details about a battle<br>[R15] The platform allows educators to set registration deadline to a battle<br>[R16] The platform allows educator to set a final submission deadline to a battle<br>[R17] The platform allows educator to set additional configuration for scoring |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- BattleCreationManager<br>- Model<br>• DataServer<br>• NotificationManager |

| | |
|---|---|
| Requirements: | [R18] The platform allows the student to enroll to a tournament |

| | |
|---|---|
| Components: | ● WebServer:<br>- Dispatcher<br>● ApplicationServer:<br><br>-TournamentSubsciptionManager<br>- Model<br>● DataServer<br>● eMail Provider |

| | |
|---|---|
| Requirements: | [R19] The platform allows student to invite others students to a their team for a battle<br>[R20] The platform allows student to accept a team invitation for a battle respecting the constraints about the size of the team<br>[R21] The platform allows student to enroll to a battle |
| Components: | ● WebServer:<br>- Dispatcher<br>● ApplicationServer:<br>- TeamManager<br>- Model<br>● DataServer<br>● eMail Provider |

| | |
|---|---|
| Requirements: | [R22] The platform creates a github repository containing the CK when registration deadline expires<br>[R23] The platform send the repo's link to all student who are members of a subscribed team when registration deadline expires<br>[R24] The platform must receive information through API calls from github action |

| | |
|---|---|
| | [R25] The platform is triggered when a push is committed before deadline<br>[R26] When a push is committed the platform must pull the latest sources |
| Components: | • ApplicationServer:<br>- BattleManager<br>- Model<br>• DataServer<br>• GitHub:<br>- Repo<br>- GitHubWebHooks |

| | |
|---|---|
| Requirements: | [R27] When the platform pulls latest sources from the repo of a team, must analyze them and run the tests on the corresponding executable<br>[R28] When the tests of a team's code are executed, the platform must calculate and update the battle score of the team<br>[R29] The final score of a team must be a natural number between 0 and 100<br>[R30] The platform must evaluate the score in a fully automated way<br>[R32] The platform automatically updates the battle scores of a time as soon as new push actions on github are performed |
| Components: | • ApplicationServer:<br>- BattleScoreManager<br>- Model<br>- AutoTestManager<br>• DataServer<br>• GitHub:<br>- Repo<br>- GitHubWebHooks |

| | |
|---|---|
| Requirements: | [R31] The platform allows educators to manually adjust the score of the battle<br>[R33] Both student and educator involved in a battle must see the current rank evolving during the battle |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- BattleScoreManager<br>- Model<br>• DataServer |

| | |
|---|---|
| Requirements: | [R34] Once the consolidation stage finishes all students participating in the battle must be notified when the final rank becomes available. |
| Components: | • ApplicationServer:<br>- BattleScoreManager<br>- Model<br>• DataServer<br>• NotificationManager |

| | |
|---|---|
| Requirements: | [R35] The platform, for each tournament, updates a rank that measure how a student's performance compares to other students in the context of that tournament |
| Components: | • ApplicationServer:<br>- TournamentScoreManager<br>- Model<br>• DataServer |

| | |
|---|---|
| Requirements: | [R36] The student ranking in a tournament is available for all students and educators subscribed to the platform |
| Components: | ● WebServer:<br>- Dispatcher<br>● ApplicationServer:<br>- TournamentScoreManager<br>- Model<br>● DataServer |

| | |
|---|---|
| Requirements: | [R37] The list of ongoing tournaments is available for all students and educators subscribed to the platform |
| Components: | ● WebServer:<br>- Dispatcher<br>● ApplicationServer:<br>- TournamentManager<br>- Model<br>● DataServer |

| | |
|---|---|
| Requirements: | [R38] When an educator closes a tournament, as soon as the final tournament rank becomes available the platform notify all students involved in the tournament |
| Components: | ● WebServer:<br>- Dispatcher<br>● ApplicationServer:<br>- TournamentManager<br>- Model<br>● DataServer<br>● NotificationManager |

| | |
|---|---|
| Requirements: | [R39] An educator can define badges when they create a tournament<br>[R40] Each badge has a title and one or more rules that must be fulfilled to achieve the badge |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- BadgeCreation<br>- Model<br>• DataServer |

| | |
|---|---|
| Requirements: | [R41] At the end of a tournament the badge rules must be checked |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- TournamentManager<br>- Model<br>- BadgeCreation<br>• DataServer |

| | |
|---|---|
| Requirements: | [R42] An educator can define new rules for a badges |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- RulesCreation<br>- Model<br>• DataServer |

| | |
|---|---|
| Requirements: | [R43] An educator can define new variables for a badges |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- VariableCreation<br>- Model<br>• DataServer |

| | |
|---|---|
| Requirements: | [R44] Badges can be visualized by all users |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>-TournamentManager<br>- Model<br>• DataServer |

| | |
|---|---|
| Requirements: | [R45] Both students and educators can see collected badges when they visualize the profile of a student.<br>[R46] An user can view the profile of any Student |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- ProfileManager<br>- Model<br>• DataServer |

| | |
|---|---|
| Requirements: | [R48] The platform allows the user to choose during registration whether to be |

| | |
|---|---|
| | a Student or an Educator<br>[R49] The platform supports file upload functionality.<br>[R52] The platform supports different programming languages for coding exercises. |
| Components: | • ApplicationServer:<br>- BattleManager<br>- BattleCreationManager<br>- Model<br>• WebServer:<br>- Dispatcher |

| | |
|---|---|
| Requirements: | [R50] Educators only see battles they manage. |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- TournamentManager<br>- Model<br>• DataServer |

| | |
|---|---|
| Requirements: | [R51] Educators have visibility of the list of teams enrolled. |
| Components: | • WebServer:<br>- Dispatcher<br>• ApplicationServer:<br>- BattleManager<br>- Model<br>• DataServer |

# 5. Implementation, Integration and Test Plan

## 5.1 Overview

In this section, we delve into the practical aspects of bringing the CodeKataBattle (CKB) platform to life through implementation, integration, and a comprehensive testing strategy. Our approach to implementation and

testing is geared towards meticulous bug detection, ensuring the robustness and reliability of the system.

## 5.2 Implementation plan

The implementation of the CKB system will be orchestrated through a hybrid approach, combining elements of both the bottom-up and thread strategies. This strategy aims to leverage the strengths of each approach, ensuring a balance between stakeholder visibility, incremental integration, and bug tracking. The thread strategy involves the identification of system features and the corresponding components (referred to as sub-components) responsible for delivering these features. Each feature relies on the collaboration of multiple sub-components, necessitating a defined order for their implementation. This thread approach facilitates the creation of intermediate deliverables, providing stakeholders with tangible results for evaluation and validation. It aligns with the iterative nature of software development, allowing continuous feedback and refinement.
Complementing the thread strategy, the implementation plan embraces a bottom-up approach to promote incremental integration. This method supports the step-by-step assembly of sub-systems resulting from the integration of individual components. The bottom-up strategy is instrumental in bug tracking, allowing for the systematic testing of intermediate results as additional modules are integrated. This incremental integration fosters a more robust and stable system by identifying and addressing issues early in the development process.

### 5.2.1 Features identification

The features to implement in the system are extracted directly from the system requirements. In the following there is a brief recap of the features of the system.

**[F1] Sign Up and Login**
These functions are available to all users. Users can register as either students or educators, providing an email (used as the username) and a password. After registration, a verification email is sent, and upon confirmation, users can access the system.

**[F2] Manage Tournaments and Battles (Educators)**

Educators can create tournaments and battles, view their dashboard displaying all created tournaments and battles, and see the list of participants enrolled in a battle or tournament. Educators can create battle settings, upload CK, and set other configurations.

**[F3] Create Teams for Battles (Students)**
After registration, students can search for tournaments and engage in a battle individually or by forming a team. Students can send invitations to others to join their team.

**[F4] GitHub Integration and Automated Testing**
Integration with GitHub, triggering actions on code push, and automated testing of submitted code.

**[F5] Availability of Rankings**
Ranking of student performance in a tournament and team performance in a battle, visibility of ongoing tournaments, and notifications.

**[F6] Badge System**
Definition, creation, and visualization of badges based on rules set by educators.

**[F7] User Management**
Viewing profiles of students, choosing user type during registration, file upload functionality, and educator-specific visibility features.

**[F8] Notifications**
All possible notifications that can be sent automatically within the CKB platform.

## 5.3 Component Integration and Testing

In the following section, we specify for each stage, what components are implemented in the stage, and how the components are integrated and tested (there is an integration diagram for each stage).

In the first step Figure 5.1 the Model is implemented and unit tested using a driver for the components that are still under implementation. These components are responsible for all the interactions with the database and are needed by the majority of all the other components.

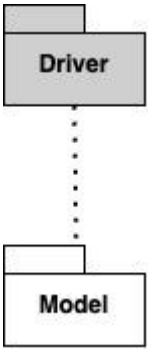



Figure 5.1 - Model developing

**[F1] Sign Up and Login**

Then we give priority to the development of authentication mechanisms because some components need features developed in this stage. The EmailProvider component is not tested because it is an external component and so it is considered reliable.

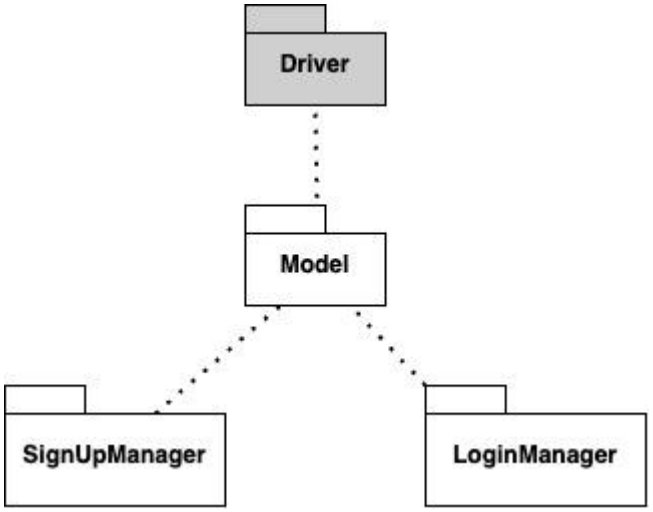



Figure 5.2 - Sign in and Sign up developing

**[F2] Manage Tournaments and Battles (Educators)**

Now we are ready to implement the testing part related to the management of the tournaments and battles. This is one of the main features of our system so it needs to be well tested.

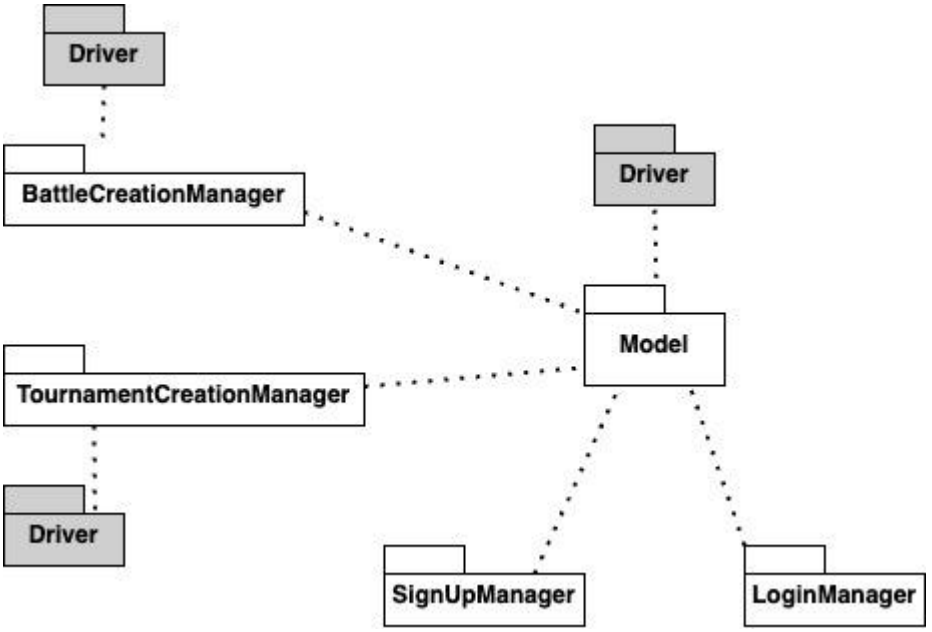



Figure 5.3 - Tournament and battles management

**[F3] Create Teams for Battles (Students)**

The next part to be tested concerns students who can participate in battles alone or by forming teams.

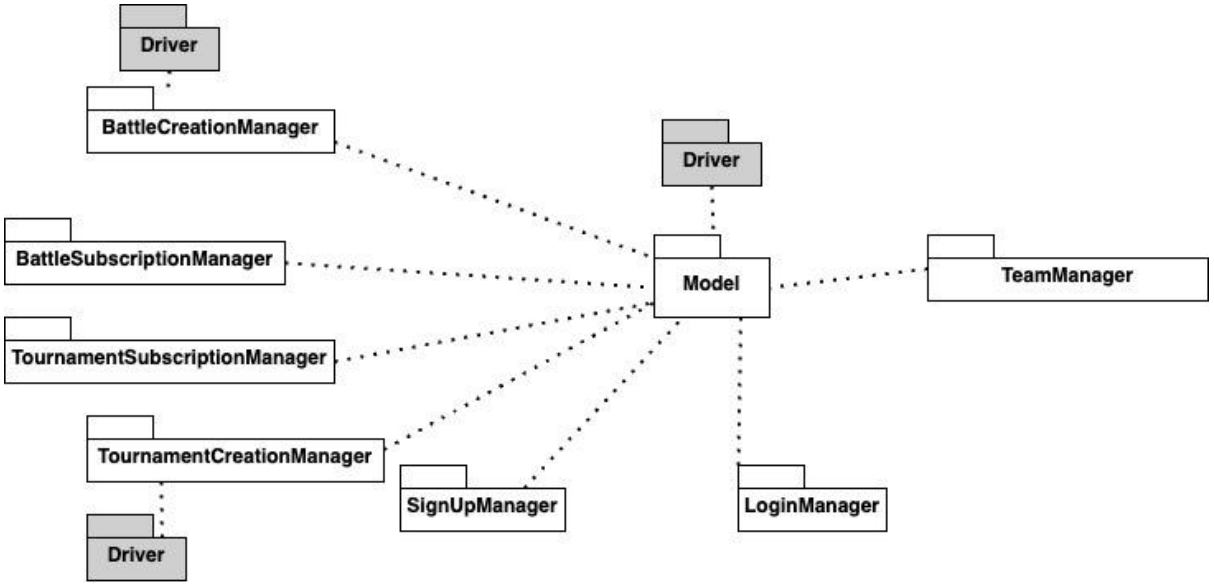



Figure 5.4 - Team creation for battles

### [F4] GitHub Integration and Automated Testing

One of the most critical parts to test is the integration with GitHub, triggering actions on code push, and automated testing of submitted code.

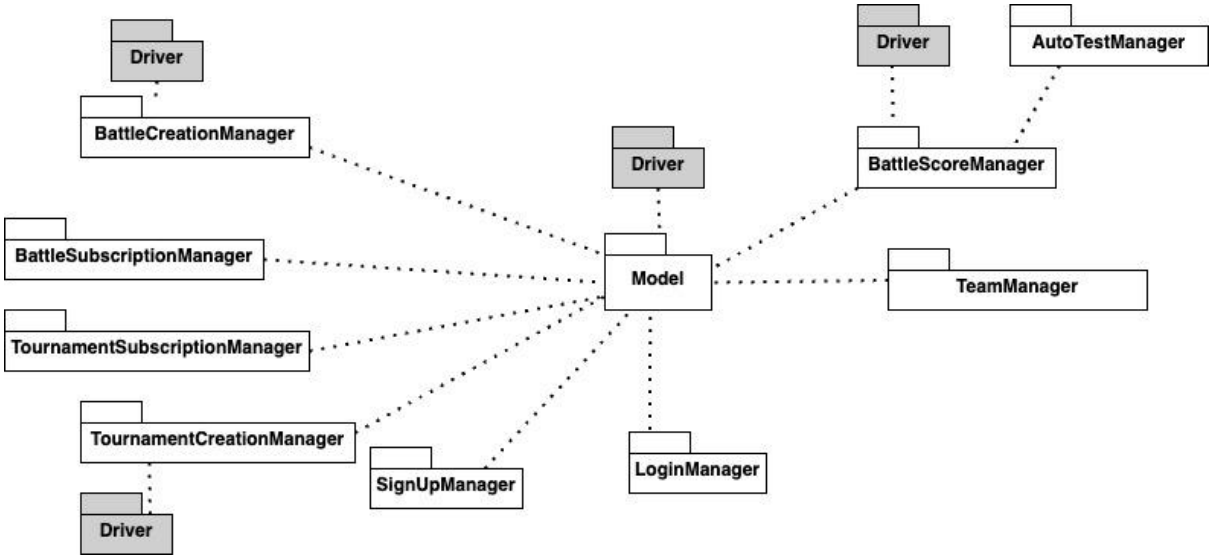



Figure 5.5 - GitHub Integration and Automated Testing Developing

### [F5] Availability of Rankings

The next step to test are the following rankings:

- Student performances in a tournament
- Teams in a battle

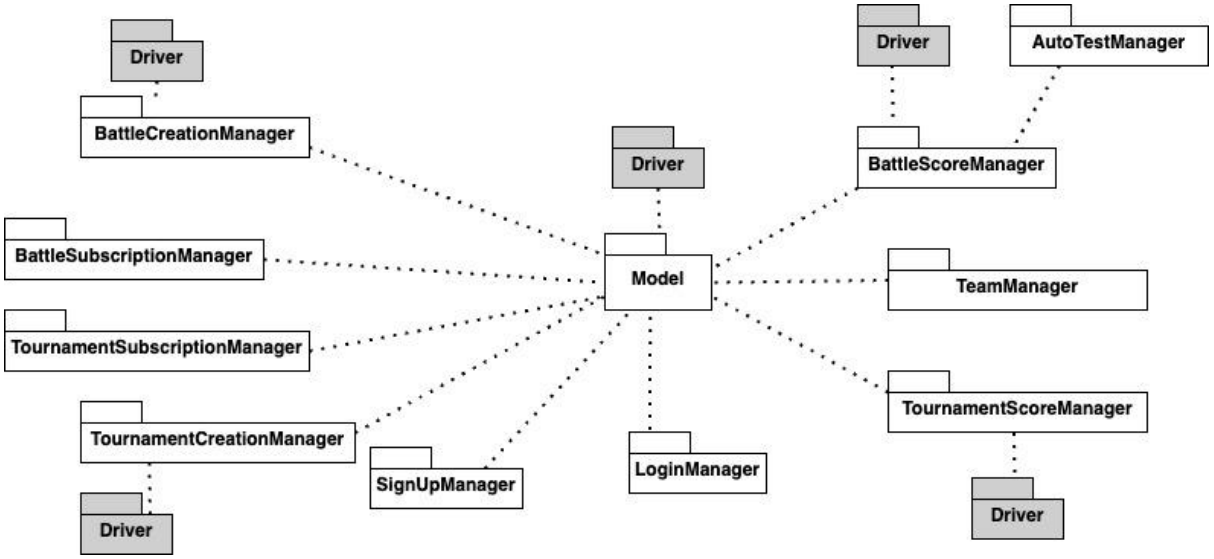



Figure 5.6 - Rankings Developing

### [F6] Badge System

One important functionality is the definition, creation, and visualization of badges based on rules created by educators.

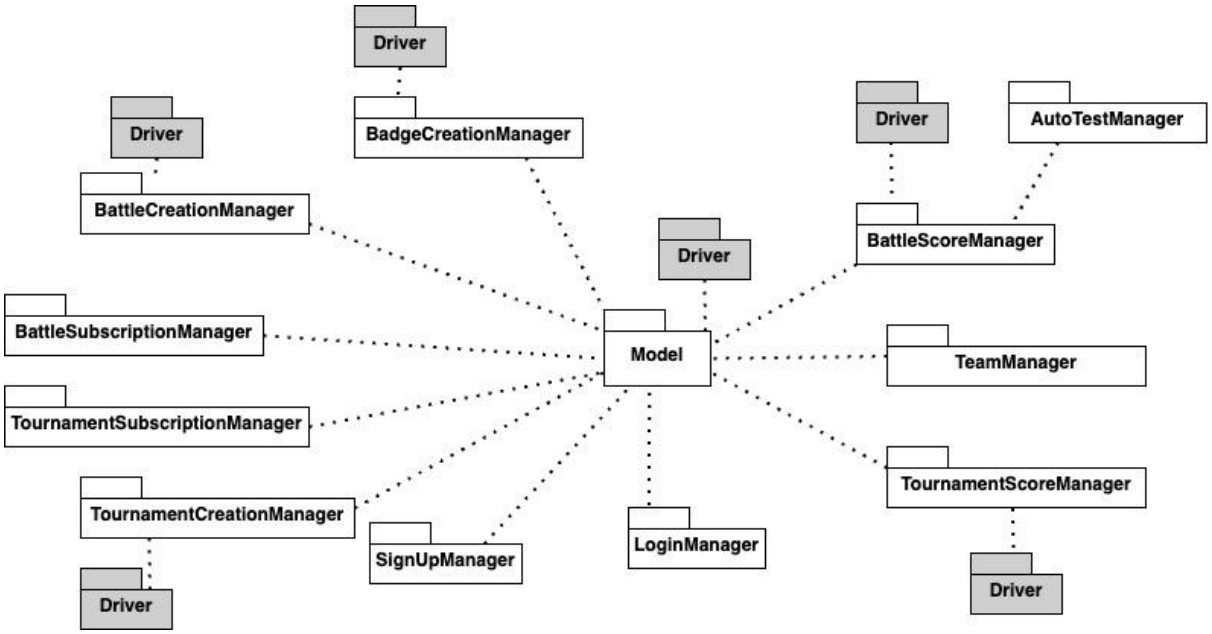



Figure 5.7 - Badge system developing

### [F7] User Management

Now user-related functionalities can be tested: viewing profiles of students, file upload functionality, and educator-specific visibility features.

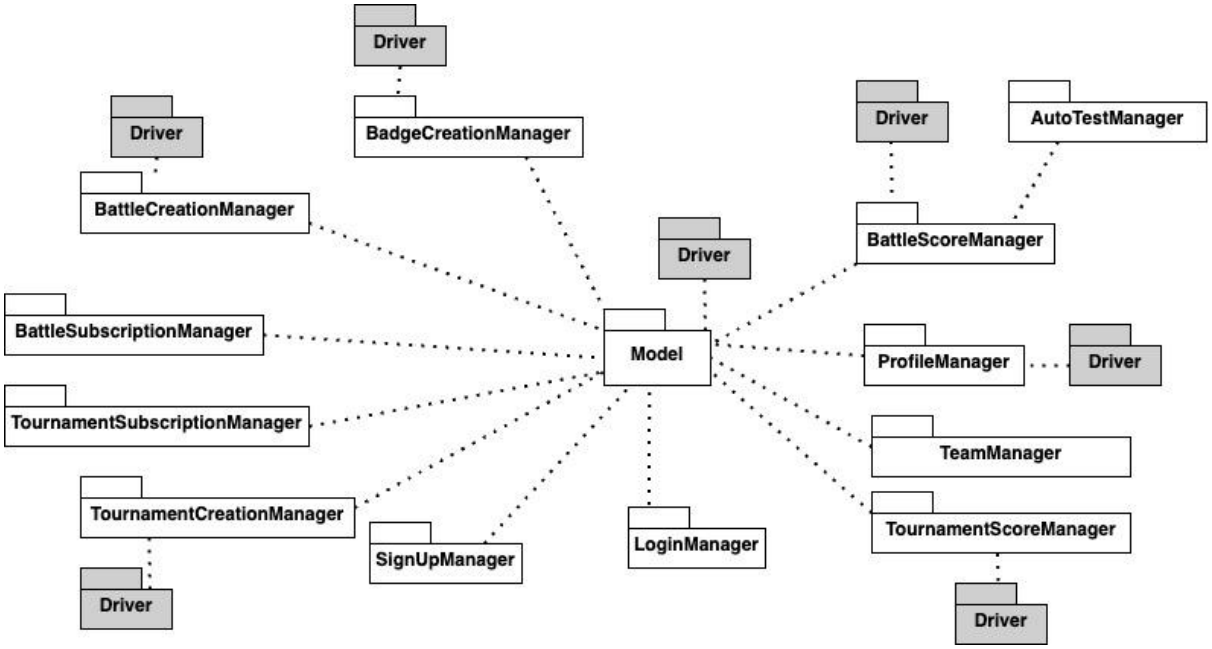



Figure 5.8 - User management developing

### [F8] Notifications

After testing all the functionalities of the platform, the last important thing to test remains the notification mechanism with all possible notifications that can be sent automatically.

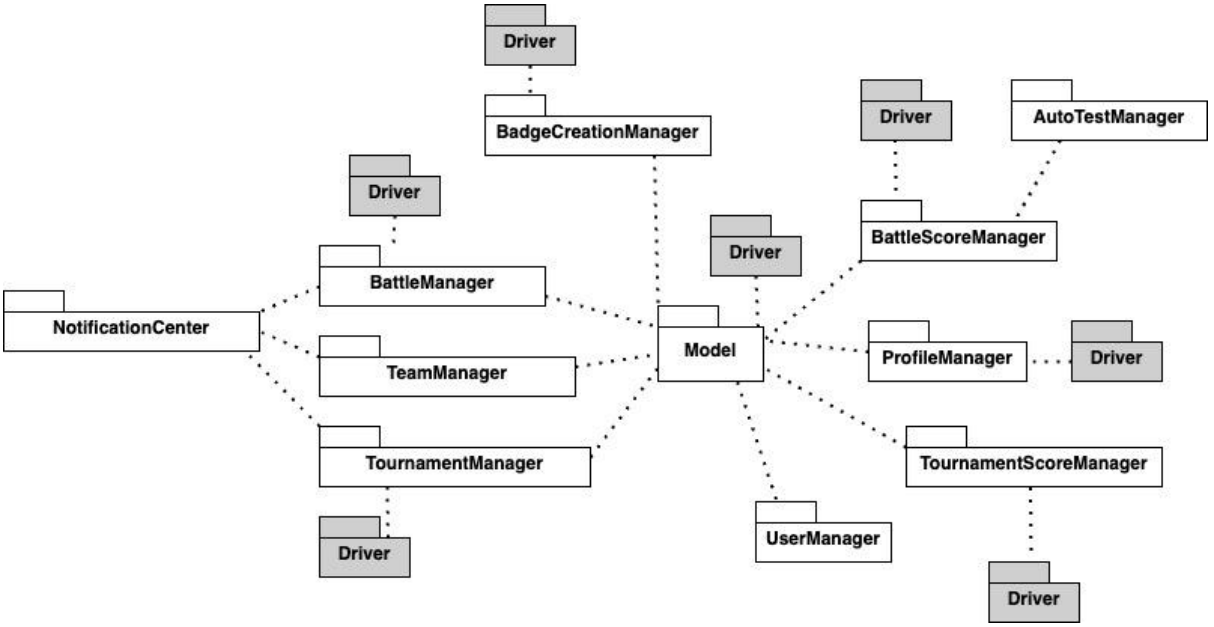



Figure 5.9 - Notification functionalities developing

## 5.4 System testing

The CKB platform undergoes comprehensive system testing to ensure its reliability, performance, and usability. The testing activities are conducted at various levels of granularity, progressing from individual components to the integrated system. During the development phase, each component or module undergoes individual testing to verify its standalone functionality. In cases where components are dependent on others, Driver and Stub components are designed and implemented to simulate the behavior of surrounding modules, including both expected and unexpected behaviors. Integration testing involves combining components and testing their interactions. The CKB platform employs a combination of bottom-up and thread strategies for integration testing. This ensures that components are integrated incrementally, allowing for early bug detection and effective tracking. Once individual components and their integrations are validated, the entire system undergoes comprehensive system testing. This phase is crucial for verifying that all features have been correctly implemented and align with the functional and nonfunctional requirements defined in the RASD.

- Functional Testing: the objective is to verify that the system satisfies all requirements specified in the RASD. Tests are conducted with an expected workload and predefined performance targets.
- Performance testing: to identify bottlenecks, inefficient algorithms, and other performance-related issues, the metrics are response time, resource utilization, and throughput.
- Usability testing: to evaluate how well users can utilize the system. This testing ensures the system is user-friendly and tasks can be performed efficiently.
- Load testing: useful to expose bugs like memory leaks and identify upper limits of components. The approach is to gradually increase the load until reaching the system's operational threshold.
- Stress Testing: the objective is to ensure the system recovers gracefully after failure

## 5.5 Additional specifications on testing

New functionalities are rigorously tested whenever they are added to the system. This ensures the early detection and rectification of bugs, maintaining the overall stability of the platform. User and stakeholder feedback are integral to the development process. Regular interactions with users and stakeholders occur at key development milestones. Alpha versions of the CKB system are shared with users to gather feedback on functionality, identify any malfunctions, and assess overall satisfaction. Users have the opportunity to evaluate the system and provide insights during its developmental stages. This proactive approach facilitates agile adjustments and ensures the final product aligns seamlessly with the expectations of both educators and students using the platform.

# 6. Effort

In the following table is reported the effort of each component of the group for each chapter.

Aleksandro Aliaj

| chapter | hour spent |
|---|---|

| | |
|---|---|
| 1 | 1 |
| 2 | 40 |
| 3 | 1 |
| 4 | 1 |
| 5 | 15 |

Federico Albertini

| chapter | hour spent |
|---|---|
| 1 | 2 |
| 2 | 50 |
| 3 | 2 |
| 4 | 4 |
| 5 | 5 |

Leonardo Betti

| | |
|---|---|
| 1 | 1 |
| 2 | 40 |
| 3 | 1 |
| 4 | 10 |
| 5 | 3 |

# 7. References

- Diagrams made with: draw.io ( https://app.diagrams.net/ ), dbDiagram.io ( https://dbdiagram.io/ )
- Mockups made with: figma