Design Document

CodeKataBattles

Karl Monrad Kieler {karlmonrad.kieler@mail.polimi.it}
Aske Schytt Meineche {askeschytt.meineche@mail.polimi.it}
Leonie Dragun {leonie.dragun@mail.polimi.it}

07 Januar 2024

Contents

1 Introduction

1.1 Purpose

The purpose of this document is to assist developers in the construction of software that complies with the requirements established in the corresponding RASD document. This is done by describing the components the system will consist of, as well as their interactions during critical processes. The document will also present a variety of perspectives on the chosen architecture, both with regards to physical infrastructure, logical separation and application event flow. The components described in this document will be presented in conjunction with the requirements presented in the RASD, in order to display why each component is present to management. Lastly, the document will present the plans for implementing, integrating and testing the systems components. All section contain reflections on how these architecture choices affect the systems functionality with regards to scalability, availability etc.

1.2 Scope

The scope of this project is described below, as it is in the corresponding RASD. In recent years, online availability of scalable educational offers have increased within languages (DuoLingo) and math and science (Brilliant) . The aim of this project is to build a platform that supports educators around the world in hosting small to large scale coding challenges, honing the skills of inquisitive students.

CodeKataBattles presents an environment where students form teams to engage in code kata battles, challenging them to develop solutions that meet specific coding requirements and pass predefined tests. The platform's intuitive user interface enables students to participate in these battles, receive immediate feedback, and learn from both successes and mistakes.

CKB supports coding challenges that promote hand-on learning, as well as collaboration through team development and knowledge sharing. Users are also able to track their progress in both tournaments and battles, being awarded with badges when achieving predefined goals.

CKB provides two primary interfaces: a student interface for participating in battles, reviewing codes, collaborating, and tracking progress, and an educator interface for setting up battles, monitoring student progress, and accessing analytics to improve educational content.

Following the World and Machine paradigm by M. Jackson and P. Zave, we identify the Machine as the CKB system to be developed and the educational environment as the World. This distinction allows categorization of phenomena into those within the World (educational needs, team interactions), those controlled by the Machine (coding challenges, feedback mechanisms), and shared phenomena (student engagement, learning outcomes).

1.3 Definitions, Acronyms

1.3.1 Definitions

1. **Educator**: A type of user that is unable to participate in battles, but can create battles and tournaments.
2. **Student**: A type of user that can participate in battles and subscribe to tournaments
3. **GitHub**: One of the most widely used version control platforms for code.
4. **UI**: User Interface

1.3.2 Acronyms

1. **Educator**: A type of user that is unable to participate in battles, but can create battles and tournaments.
2. **Student**: A type of user that can participate in battles and subscribe to tournaments

3. **GitHub**: One of the most widely used version control platforms for code.

1.4 Revision History

1. Version 1.0 (5th January 2024)

1.5 Reference Documents

This Document is strictly based on

1. Specification of DD project of the Software Engineering II course, held by professor Matteo Rossi, Elisabetta Di Nitto and Matteo Camilli at the Politecnio di Milano, A.Y 2023/2024

2. Slides of Software Engineering 2 course on WeBeep

1.6 Document Structure

The document is divided into three overall parts:

1. **Architectural Design** which is concerned with various architectural choices and their effects on the system requirements and goals. We examine the high level architecture division of the "business logic" of the system, as well as the architecture of the flow of functionality within the system and the physical infrastructure partitions. We also explain the chosen components, and the interactions occurring between them during relevant processes.

2. **Requirements Traceability** aims to describe how each service and component actively maps to the Requirements stated in the RASD document.

3. Lastly, **Implementation, Integration and Testing** outlines the intended strategy for development of the software.

2 Architectural Design

2.1 Overview

The CodeKataBattles system is architecturally designed as a web-based application with a client-server architecture. The primary components include:

- **Client interface**: A web-based user interface accessed by both students and educators. It provides the functionalities for participating in code katas (battles), joining tournaments, and viewing performance metrics (i.e. battle or tournament rankings).
- **Server Backend**: The backend serves as the brain of the system used to handle user requests, process and store data, and communicate with external services, such as GitHub.
- **GitHub Integration**: The system leverages GitHub as a version control and repository hosting service. This integration is crucial for managing code submissions, automating code testing/evaluation, and ensuring version control.

The system shall be built as a distributed system with a three-tier architecture. It includes the topmost layer; the presentation tier (client tier), presenting information and user interface elements to users, the application tier (logic tier), containing the business logic that enables the systems core functionalities by communicating with the database and other external services to retrieve or update data, and the bottom layer; the data tier, dealing with the data storage, retrieval, and management (see fig. 1).

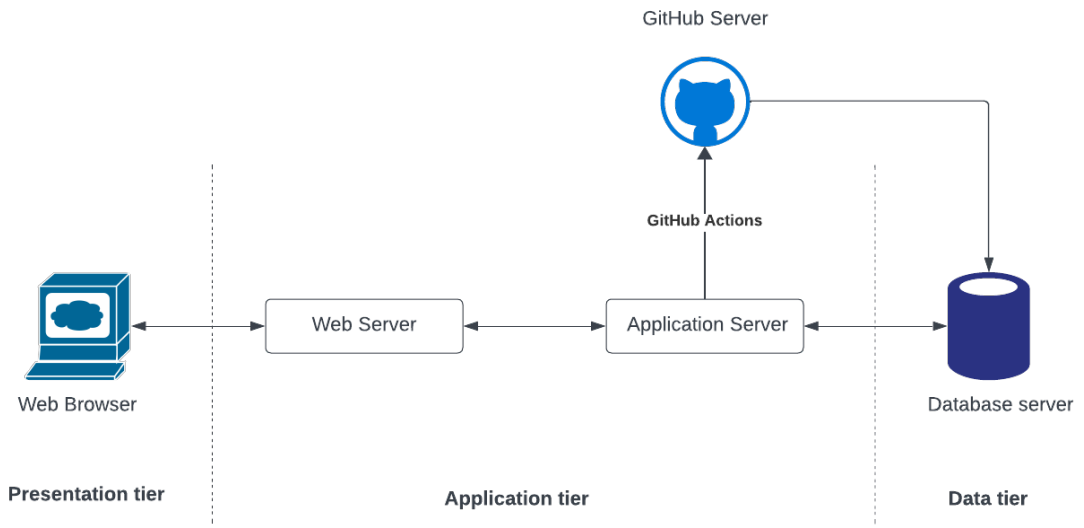

Figure 1: High level system architecture

The external service integration most crucial to the CodeKataBattle's main functionality is the integration of GitHub. This is enabled through GitHub Actions which allows for the setting up of workflows that automate tasks based on events in specific GitHub repositories. This aspect falls under the system's Application Tier, which consists of the Webserver, the Application Server, and the communication with the Github Server.

Regarding the internal set-up of the application tier, an event-driven architecture has been chosen, as this is greatly supported by GitHub Actions. The deployment of GitHub actions removes the necessity to build or integrate event brokers, publishers or subscribers, as this is handled by the action yaml-file (see fig. 2).

Every single Battle will simply consist of a standardised GitHub repository with standardised actions built in. The actions ensure that, upon a push to a forked repository, certain specified

test files are run and the results are subsequently written to the database, if the push occurs within the battle time limit.

The effective scaling of the testing modules in case of increased amount of submissions to a battle immediately before the deadline, has been identified as a potential bottleneck of the system. However, as GitHub Actions spins up an independent container for each event (e.g. a push) being handled, it harvests the strengths of a microservice architecture and averts this specific bottleneck.

The workload is split between three physical infrastructures; The submissions and testing occurs on GitHub's infrastructure, The application is hosted on the application and web server, and the database containing the results are secluded on the database server (see fig. 3). The Application Server "writes" to the GitHub server only to trigger the creation of a new battle repository upon educator input. In response GitHub actions provides the battle repository URL, which it writes to the database, from where the Application Server reads it. Thus the Application Server "writes" to the GitHub server and the Github server "writes" to the database (see fig. 1).

This physical dichotomy means that most of the workload is going to be placed on the container-based infrastructure of GitHub Actions, decreasing the requirements for the remaining infrastructure in order to remain scalable. The Web-server simply needs to support users reviewing their results and rankings of their battles, while all the scoring is handled in another system. While scoring of submissions are performed in relative parallel, potential bottlenecks can occur if many submissions are made at the same time, creating a long write-queue to the database, potentially increasing the time from a submission is made until the result is visible on the web app. Another potential bottleneck is simply that of many sign-ins and requests for battle results, for which we currently don't supply any mitigation strategy, as it is unclear how badly it will scale with user count.

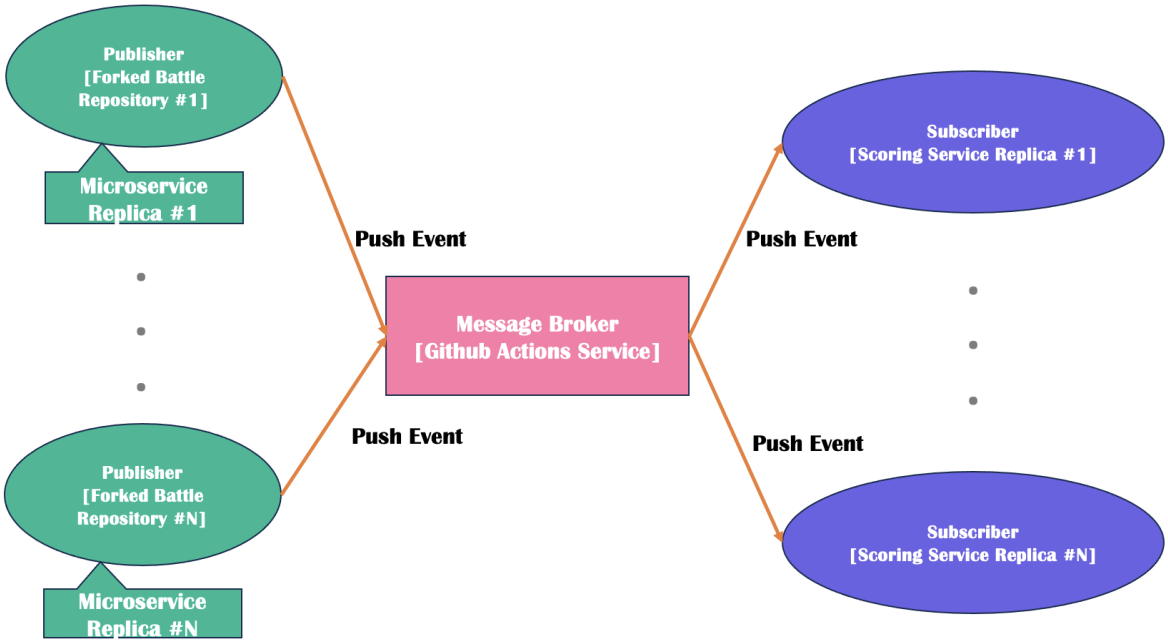

Figure 2: Event-driven architecture (Sub/Pub)

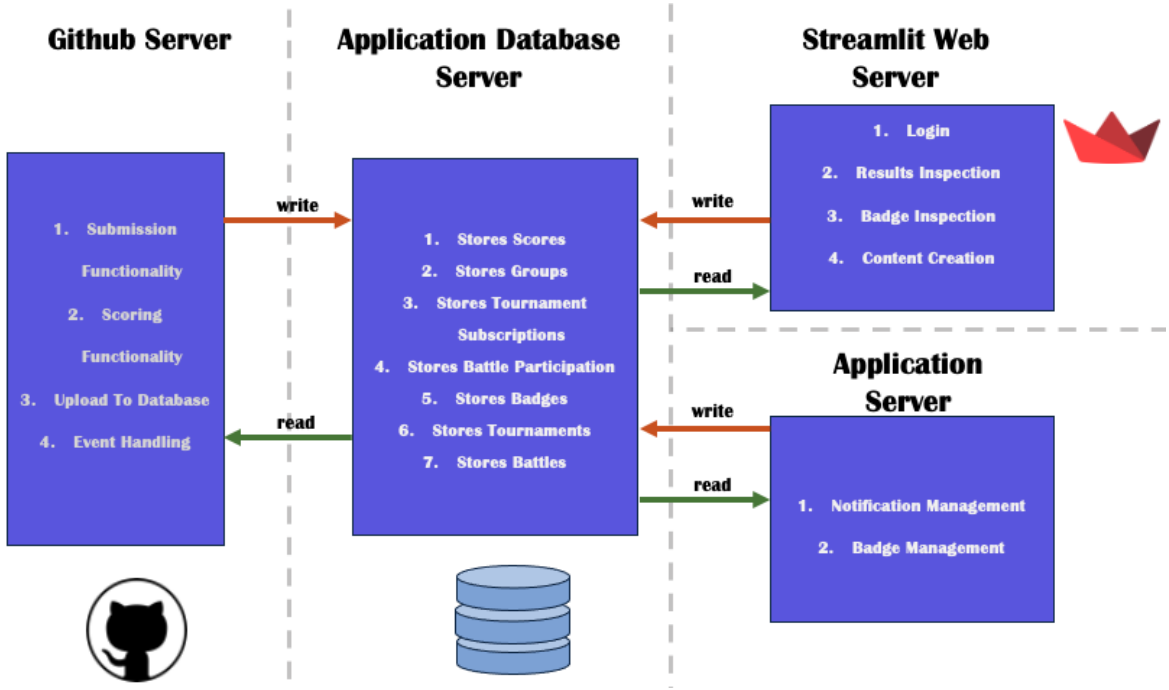

Figure 3: Physical Infrastructure Partitions

2.2 Component view

The components are organized into the following modules:

User Interface Components:
Responsible for rendering the web pages, handling user interactions, and making requests to the back-end. Individual components include:

- **Front-end Service**
  Represents the front-end of the application, handling user interactions, displaying information, and communicating with back-end services.
- **User Profile Service**
  Manages user profiles, including the display of badges, tournament ranks, and overall performance visualisation
- **Educator Tools Service**
  Provides tools and interfaces specifically for educators to create and manage tournaments, battles, and perform manual evaluations.
- **Authentication Service**
  Responsible for user authentication and authorization.
- **Github Management Service**
  Manages the creation and updates of relevant Github Repositories, as well as general automated communication with the Github API.

Back-end Components:
Responsible for processing user requests, performing application-specific functionalities and enabling the interactions between the presentation and data tier.

- **Scoring Service**
  Reads code submission test results from the database (written their by GitHub Actions) to calculate the teams battle score based on aspects such as number of tests passed, timeliness of submission and code quality levels.

- **Badge Management Service**
  Deals with the creation, assignment rules, and management of tournament badges. Assigns badges depending on the educator's defined rules at the end of each tournament.

- **Notification Service**
  Handles the notification system for informing users about new tournaments, battle updates, final battle results, and tournament badge achievements.

- **Data Persistence Service (DBMS)**
  Manages the storage and retrieval of data related to tournaments, battles, user profiles, scores, and badges.

GitHub Integration Components:

- **GitHub Actions**
  Used to set-up the event-driven automated testing of code submissions to forked battle repositories, upon each push. It stores test results to the system's database, where they are retrieved by the Scoring Service. GitHub Actions creates battle repositories, ultimately triggered by the Front-end Service.

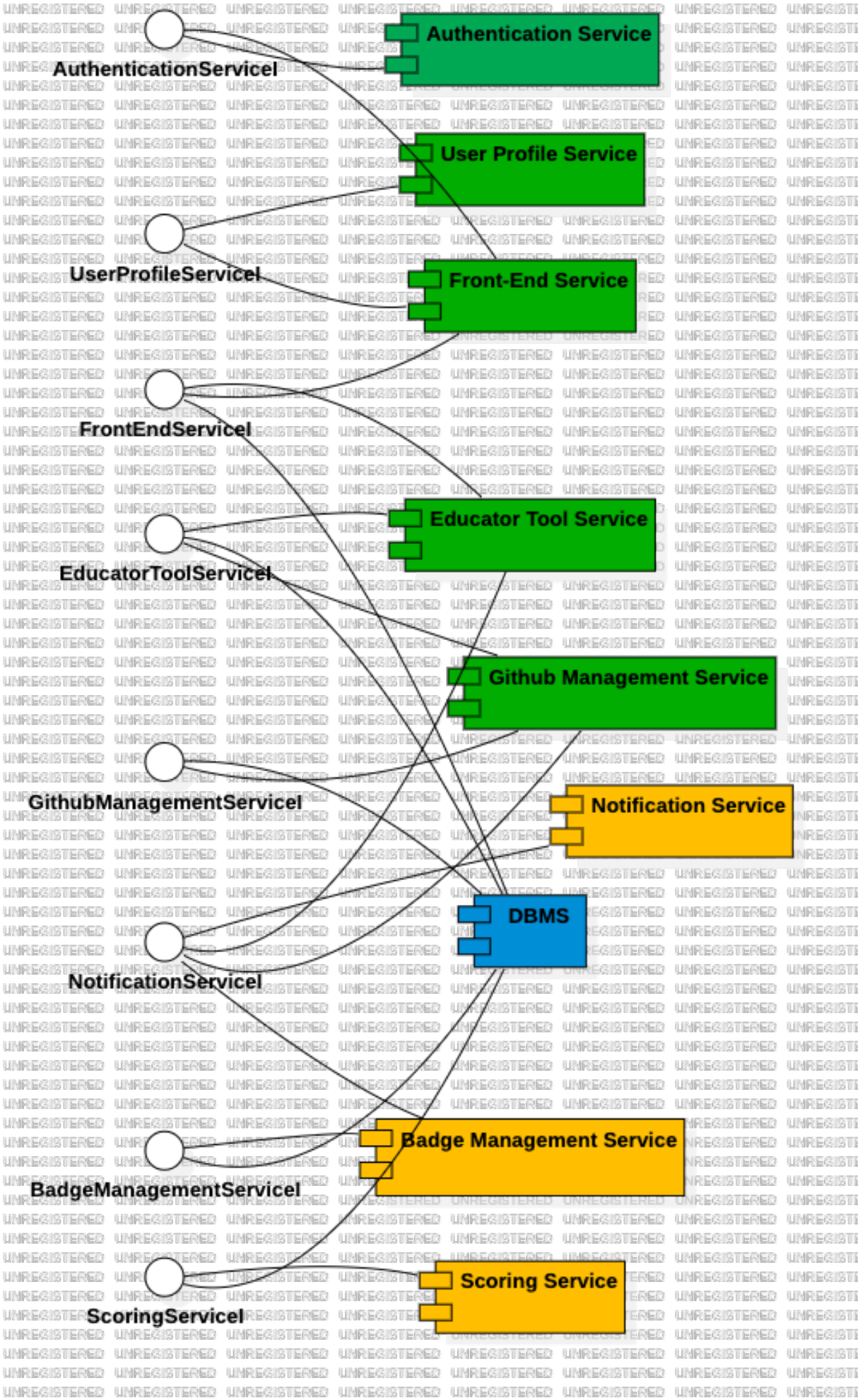

Figure 4: Component Diagram

2.3 Deployment view

The system will be deployed using a cloud-based infrastructure, ensuring scalability and availability. The deployment consists of:

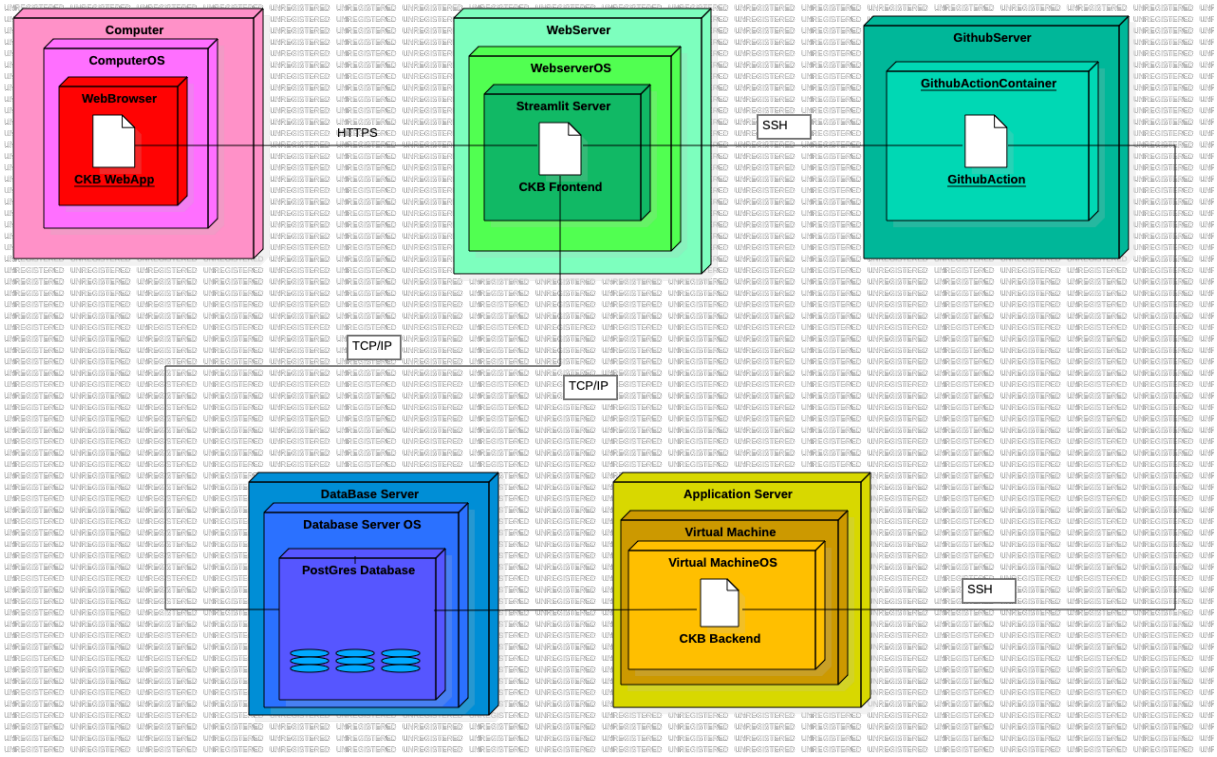

Figure 5: Deployment Diagram

2.3.1 Web Server

The Webserver will host the actual front end web application, making it available to users. Certain components, related to the frontend will also be hosted there, such as the GithubManagementService, which is responsible for communicating with the Github API, relevant during user profile management and battle creation. However, all services hosted on this machine are services, where high traffic is not expected. It communicates with the client through an https connection.

2.3.2 Application Server

The application server hosts the services that routinely monitor the activity in the system, occasionally triggering database inserts, such as notifications and badge achievements.

2.3.3 Database Server

The Database server stores all information, not explicitly stored in Githubs system, such as battle participation, tournament subscriptions, notifications, scores, badges, user profiles etc. All communication with the database is done through TCP/IP as this is the standard protocol.

2.3.4 Github Server

While Github is not hosted on machines we have direct control over, it can perform some of the heavier processing within the system. As each Github Action activation triggers a specified container to be spun up, the processing specified in our system is performed on this infrastructure. Since GitHub has a large, load balanced infrastructure, we the load balancing of parts

of our system is performed by them. The most prevalent service being run on these machines is the Scoring service, which is run whenever a participating group pushes a new submission to a forked battle repository. All communication with the Github Server service is performed through SSH as this is new standard for communication with the API.

2.4 Runtime views

2.4.1 Login Sequence

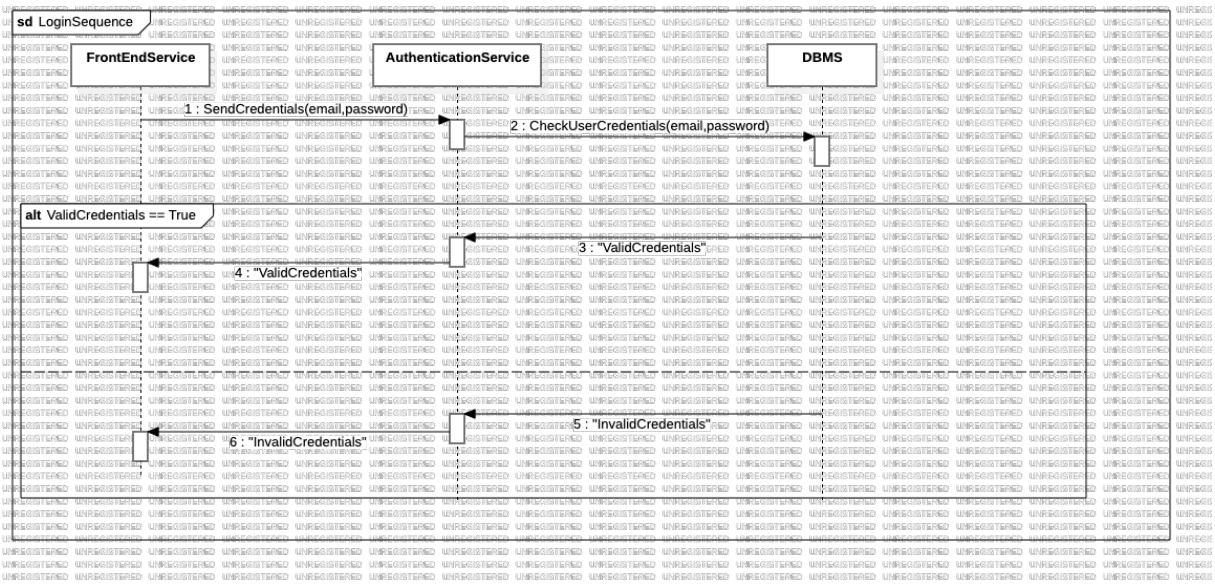

Figure 6: Login Sequence

The login-sequence in our case is quite simple, as the user simply inputs their credentials on the login page of the front-end. The credentials are then forwarded to the AuthenticationService which checks the validity of the credentials against the stored credentials in the DBMS. This functionality is the same, whether done as a Student User or an Educator User.

2.4.2 Create Battle

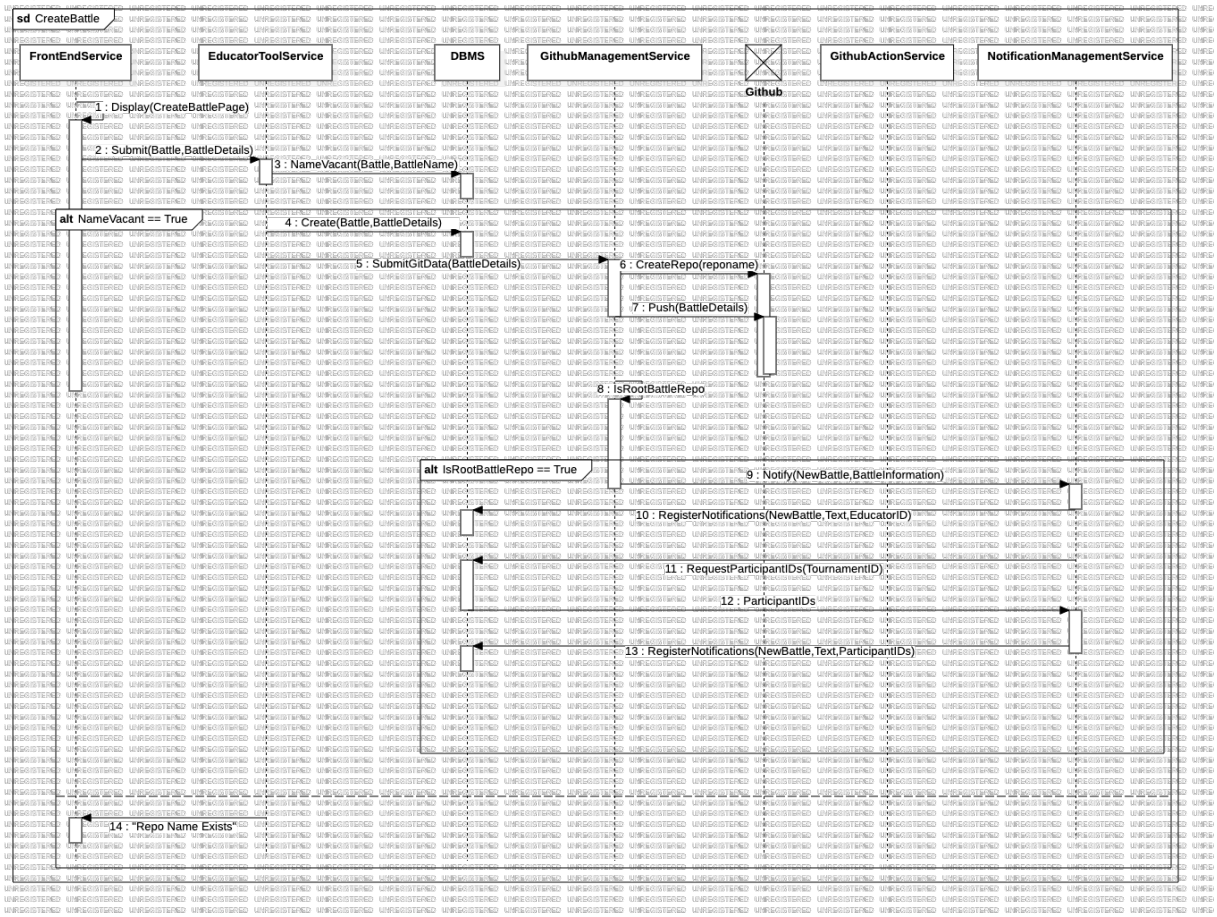

Figure 7: Create Battle

When creating a battle in our chosen architecture, we can leverage the functionality and infrastructure of Github and Github Actions. When an Educator wishes to create a Battle, they do so through the front-end service, by simply filling in the battle details, such a name, sign-up deadline, submission deadline and description. Additionally, the Educator submits a series of test files that can be used to evaluate the submissions. When the Educator finalizes the battle creation, the front-end service first verifies that no other battle shares this name, as the name will be used in the creation of the battle's Github repository, which must be unique. If the name is unique, the Educator Tool Services can generate a repository name for the battle and officially create the Battle in the Database Management System. However, this only officially creates the Battle *internally*, so a Github repository is then created from a Battle template-repository complete with Github Action YAML-files, relevant directories and so on through the Github Management Service. Subsequently, the relevant files and information from the Educator is pushed to the copied template. A check is performed to verify that this is, in fact, the root Battle repository, and not a forked submission repository. When this is verified, an action can activate the notification procedure using the Notification Management Service. The Notification Management Service first requests all userIDs of Students subscribed to the Tournament, to which this Battle belongs. Then Notifications of a new Battle is sent to all these users' email addresses.

2.4.3 Create Tournament

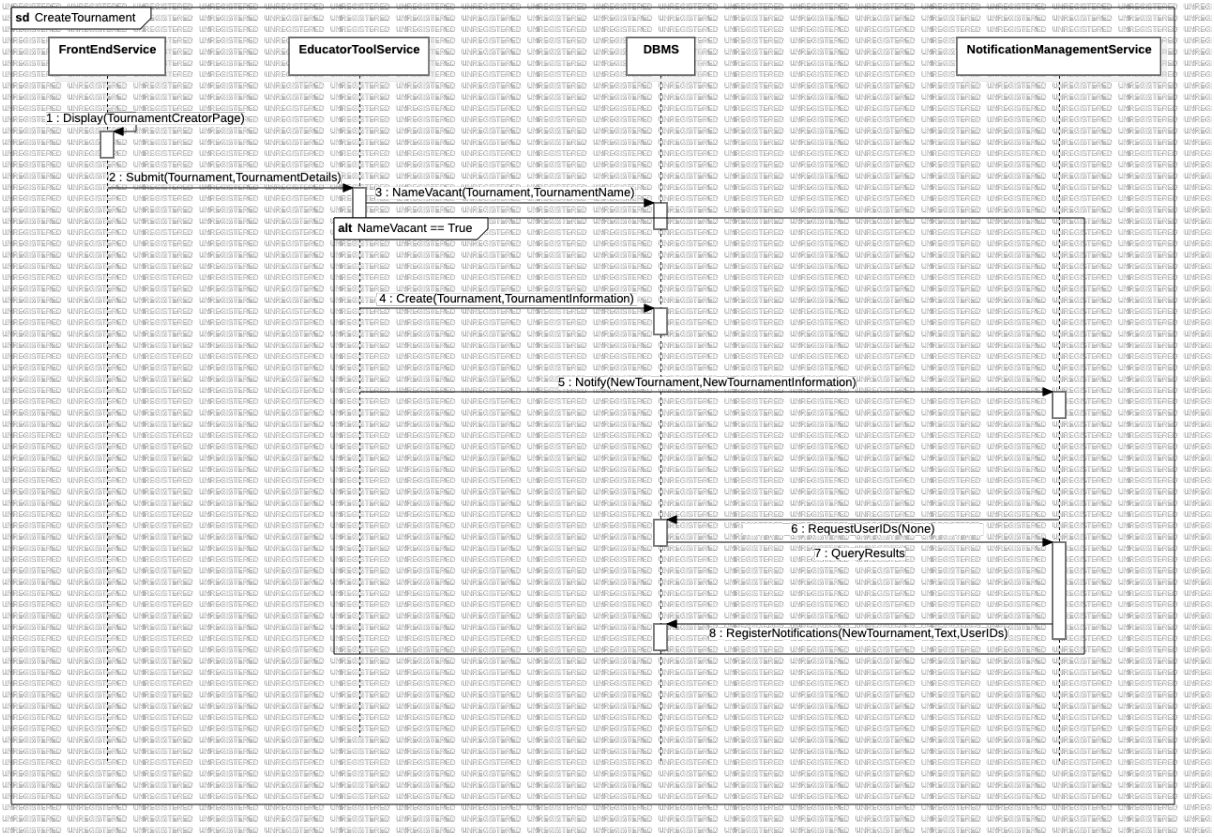

Figure 8: Create Tournament

Creating a Tournament is much simpler than creating a Battle, as no repositories are involved yet. When an Educator creates a Tournament by filling out the CreateTournament-form through the Front-end service, the Tournament details are forwarded to the Educator Tool Service. This, in turn, checks whether the a Tournament of this name already exists. If not, the Educator Tool Service can officially create the Tournament in the DBMS. When the Front-end service refreshes, it will pick up the new Tournament from the DBMS. Subsequently, a trigger with the relevant Tournament information is forwarded to the NotificationManagementService, which will notify all users on the platform that a new Tournament is available.

2.4.4 Create Badge

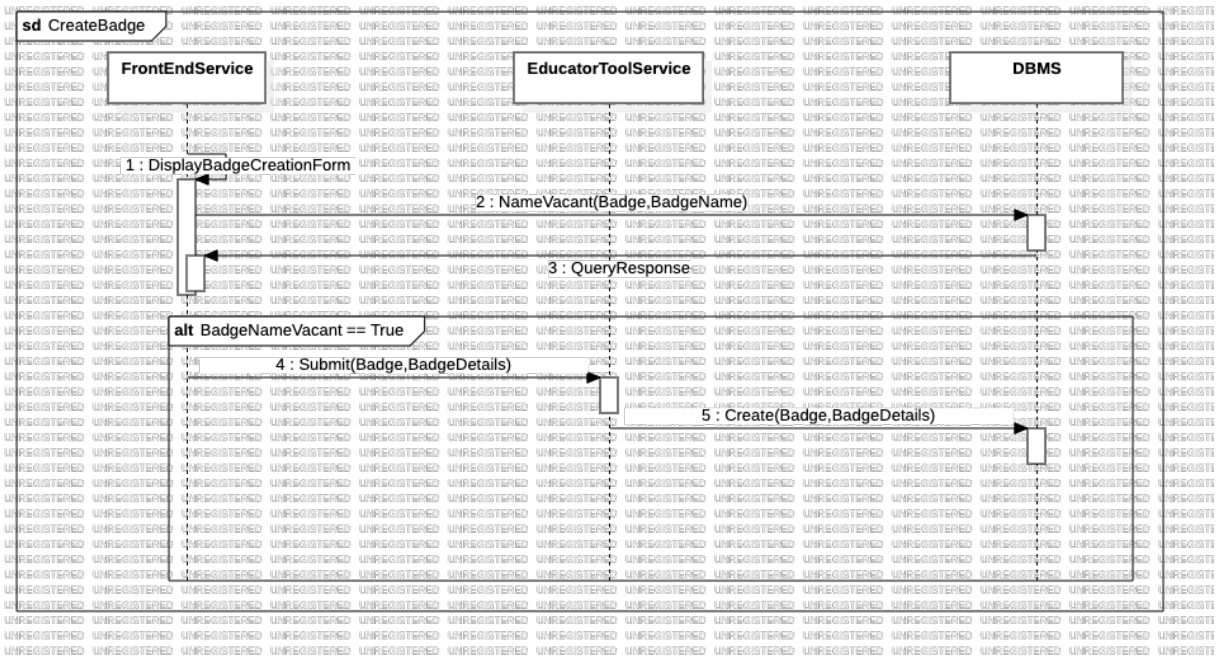

Figure 9: Create Badge

When creating a Badge, the Educator uses the BadgeCreationForm from the FrontEndService. This form consists of a crude graphical user interface, that translates well into SQL queries. First, the Front-end Service ensures, that the badge name is unique to this tournament. If this is the case, the SQL-adjacent logic is forwarded to the Educator Tool service, and stored in the DBMS.

2.4.5 Receive Badge

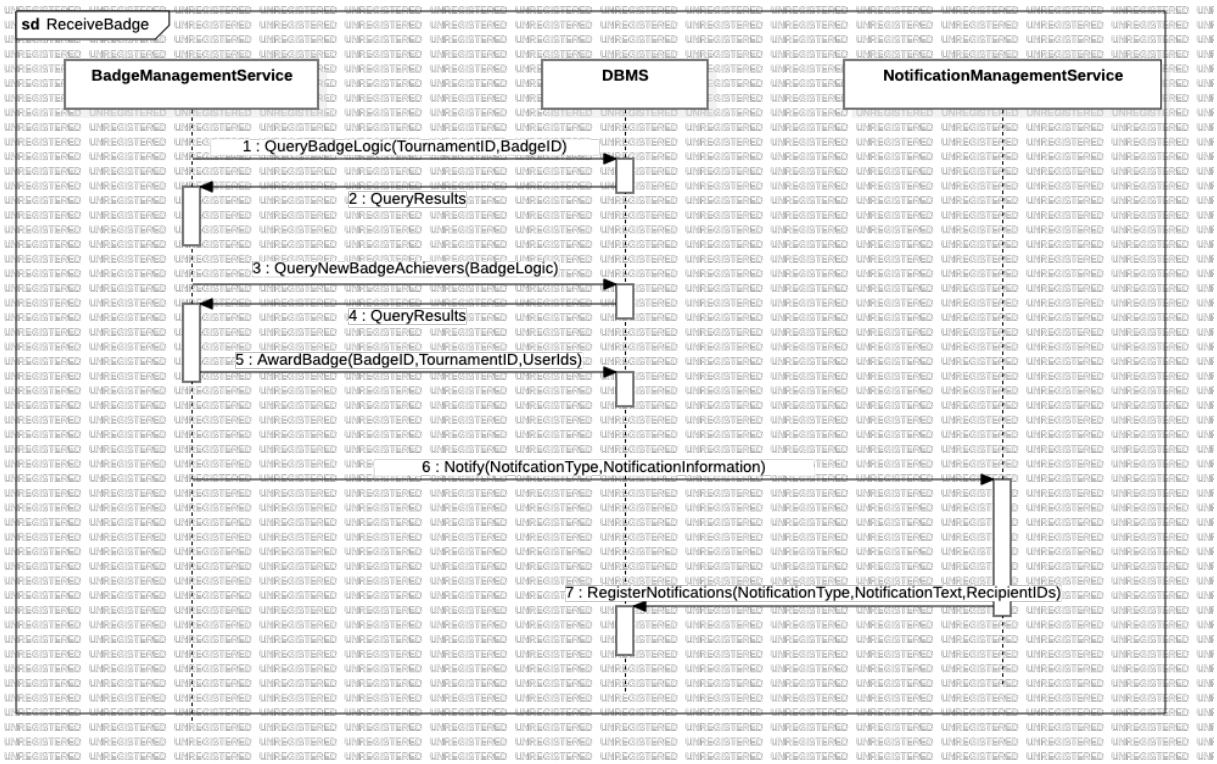

Figure 10: Receive Badge

There can potentially be quite a lot of variety in the events that can trigger a Badge. Therefore, the Badge management service simply performs a query each day, using the SQL-adjacent logic of the Badges, to retrieve all Students for all Tournaments that correspond to the queries, but have not been assigned the Badge. This solution, however, might scale poorly, as Tournaments may be active for years.

2.4.6 Submit Solution

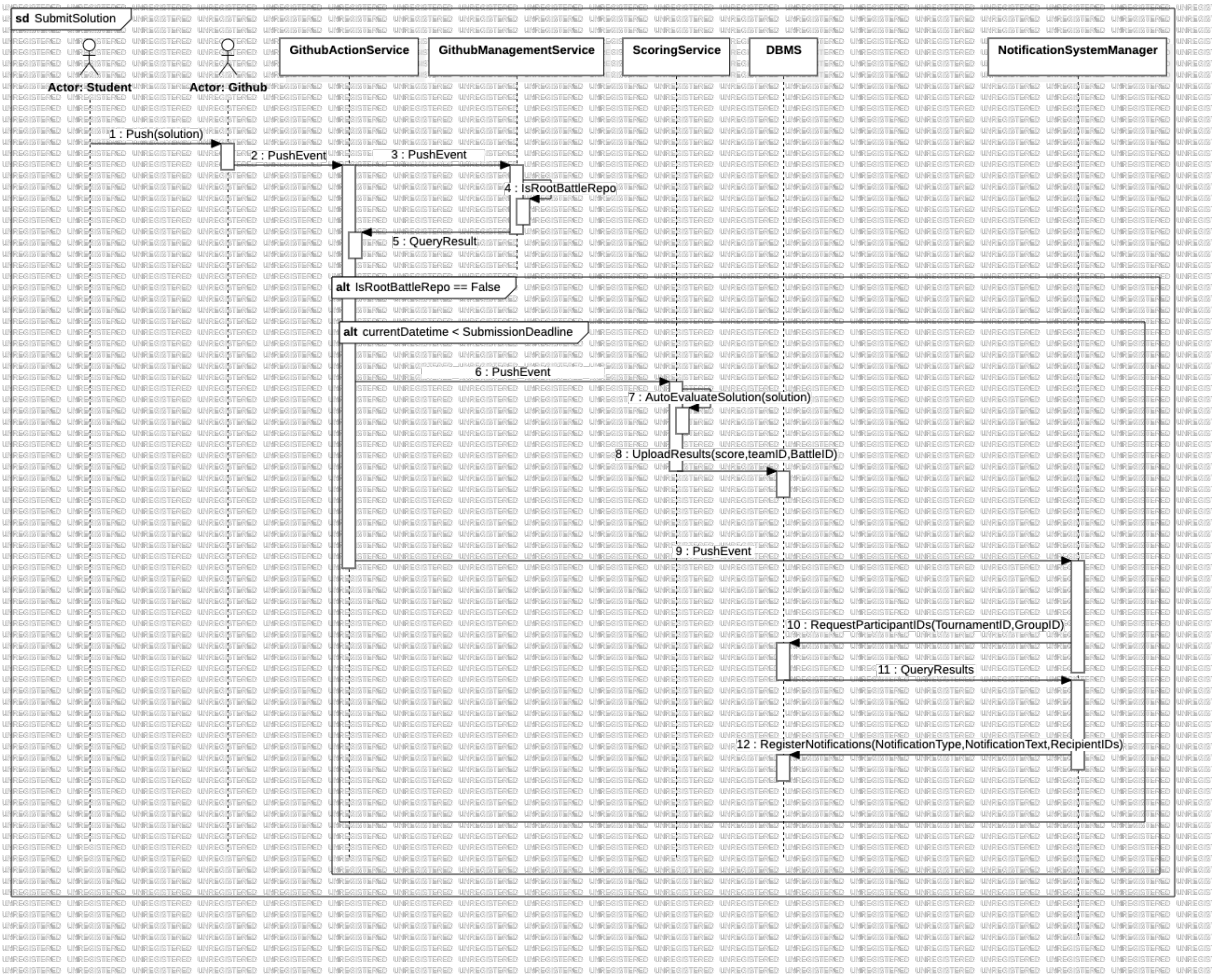

Figure 11: Submit Solution

When making a Submission, we get to leverage the choice of architecture once more. When a Student performs a push to their forked Battle repository, before the deadline, it triggers the Github Action Service. The Github Action Service will follow the instructions in the Github Actions YAML-file and activate the Scoring service, automatically evaluating the Submission. The evaluation is then written to the DBMS. Subsequently, the YAML-file dictates that a Notification is sent to the users of the responsible group, by gathering the relevant userIDs from the DBMS.

2.4.7 Join Battle

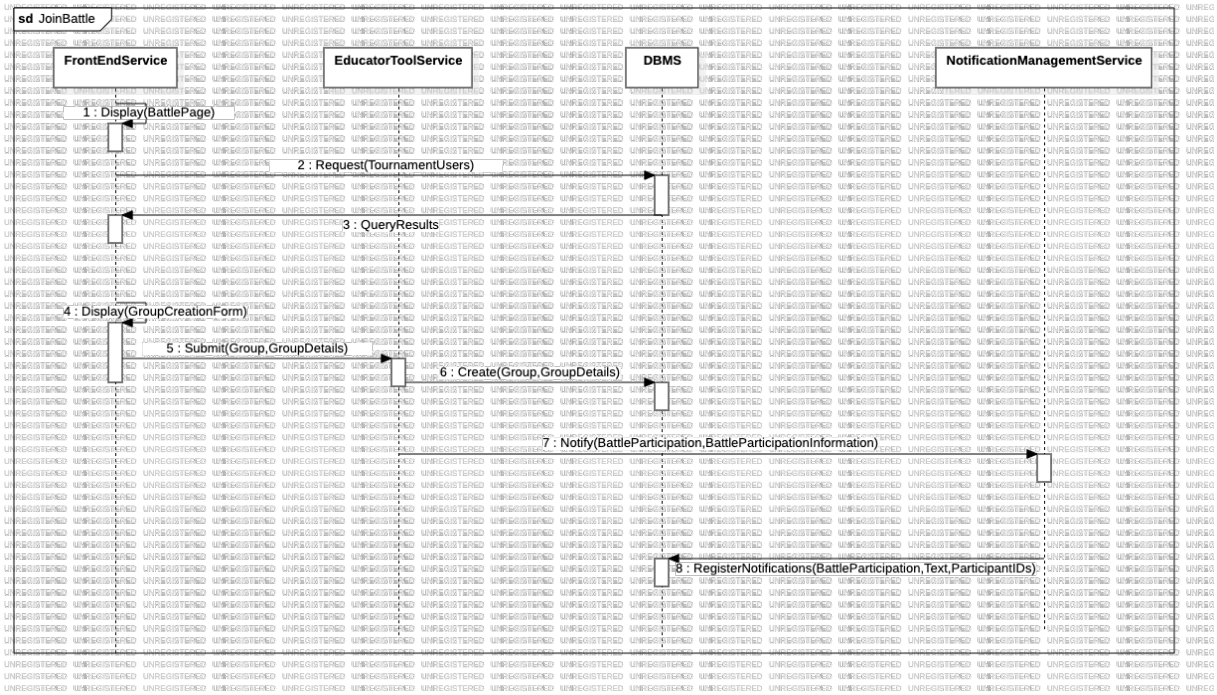

Figure 12: Join Battle

When joining a Battle from the Front End Service, a Student will be prompted with a drop down of the Students subscribed to the current Tournament without a group all retrieved from the DBMS. They can then select their preferred team mates, as long as they don't exceed the Educator determined threshold for group size. The group is then written to the DBMS, through the Educator Tool Service, which manages all user-facing *writes* to the DBMS, thus finalizing the group creation. The Notification Management Service is then triggered and notifies the users of their successful group formation, and Battle sign-up.

2.4.8 Manual Scoring

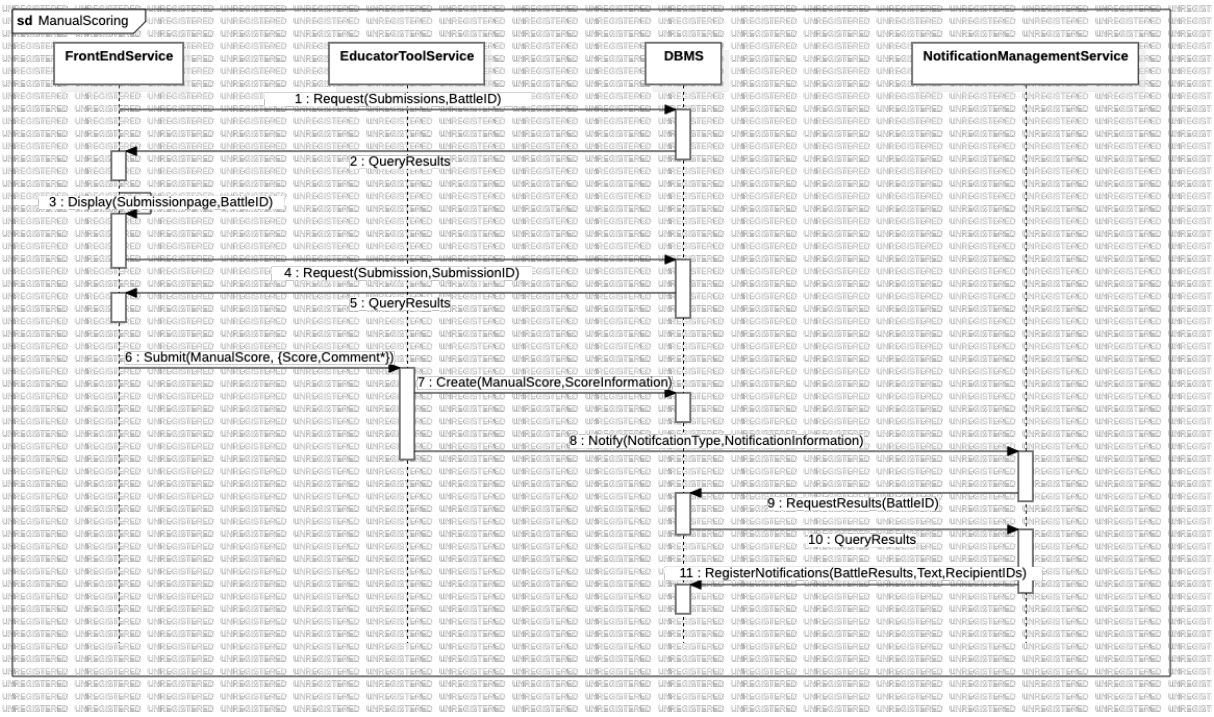

Figure 13: Manual Scoring

When a Battle has ended the Educator is able to assign a manual score, as well as a manual feedback to the Students. This is done throug the Front-end Service, which displays the Submissions from each Group which has received the maximum automated score. The Educator can then assign a score on the same range as the automated score, as well as a comment on the overall quality of the submission. This information is then written to the DBMS, and then notified to the responsible group through the Notification Management Service. The new scores will be available through the Front-end Service upon refresh.

2.4.9 Subscribe to Tournament

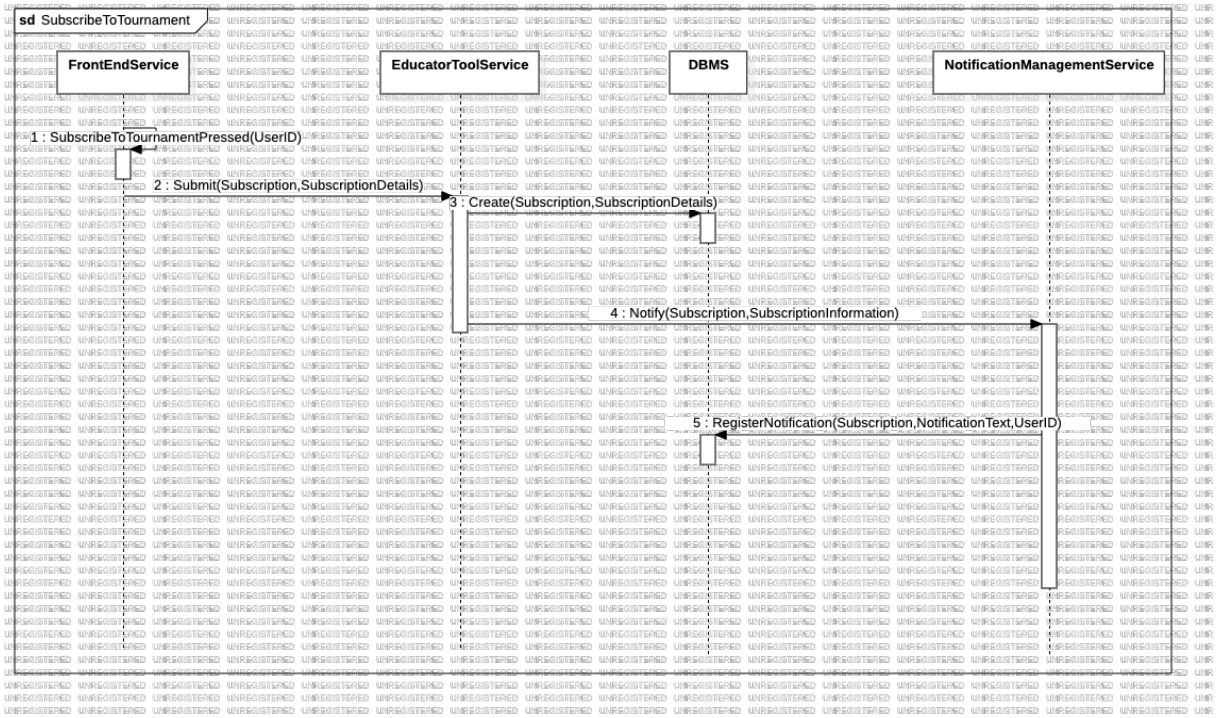

Figure 14: Subscribe To Tournament

When a Student subscribes to a Tournament through the Front-end Service, their UserID is simply added to the DBMS, allowing the system to notify the Student of future Battles, as well as making them eligible to participate in these Battles. Immediately, the Front-end Service forwards the relevant information to the Notification Management Service, which in turn notifies the user of their successful subscription.

2.4.10 Announce Battle Results

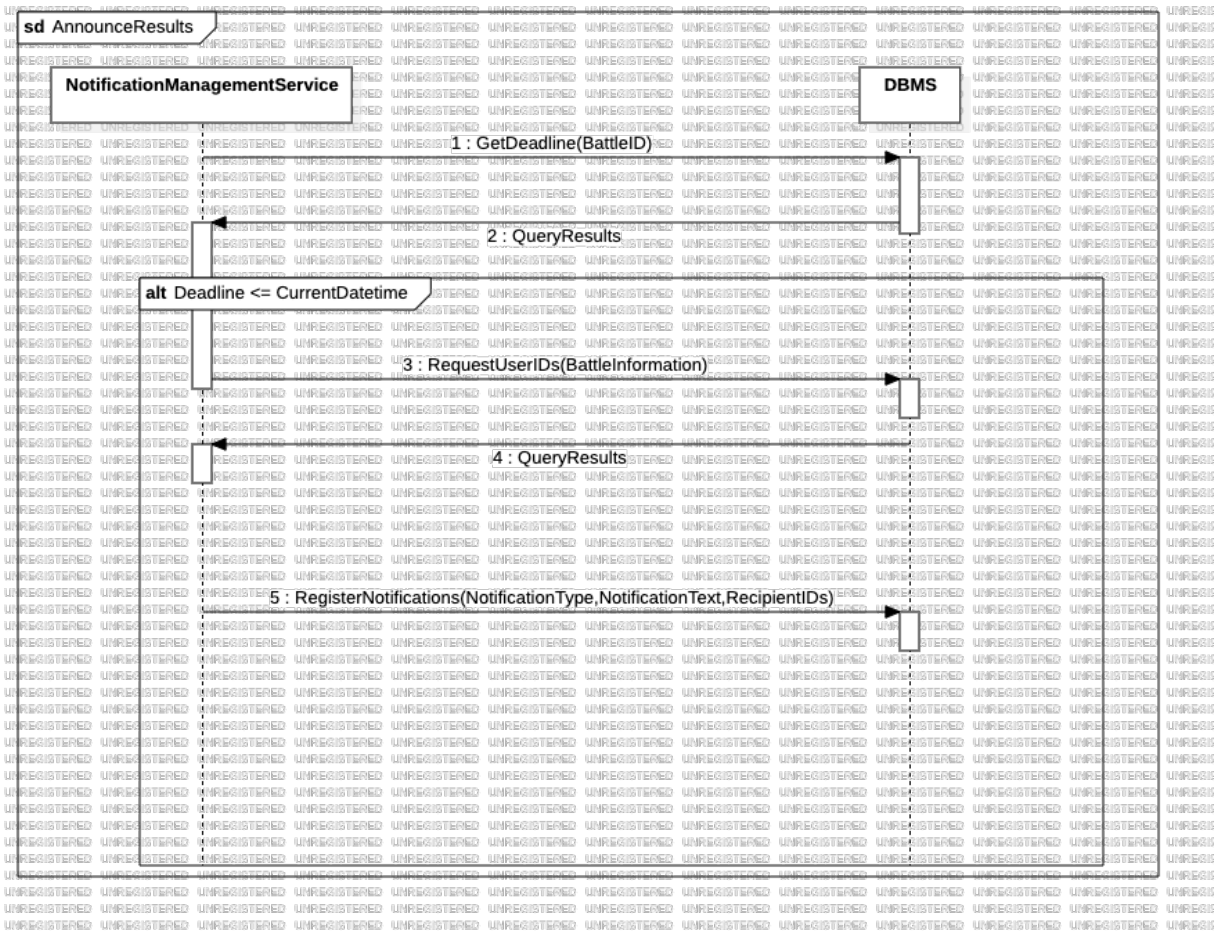

Figure 15: Announce Battle Results

Every day, the Notification Management Service, checks which Battles have ended since yesterday, by checking the DBMS. All participants of the Battle is then notified of their final ranking, which can also be found on the Battle page of the Front-end Service.

2.5 Component interfaces

The following section aims at providing a complete list of all the methods each component interface provides to the other components. (note: actual method names and parameters will be different based on the implementation details, we can change them later - consider these placeholders)

User Interface Components

- **FrontndServiceI**
  - Display(PageName,PageInformation)
  - SendCredentials(email,password)
  - Submit(Objecttype,Objectdetails)
  - NameVacant(ObjectType,ObjectInformation)
  - Request(data)
- **AuthenticationServiceI**

  - checkUserCredentials(userName, userPassword)

- **GithubManagementServiceI**
  - updateParticipantList(userID, teamID)
  - IsRootBattleRepo()
  - CreateRepo(RepoName)
  - Push(PushDetails)
  - Notify(NotifcationType,NotificationInformation)

- **UserProfileServiceI**
  - displayUserProfile(userID)
  - updateUserProfile(userID, newProfileData)

- **EducatorToolsServiceI**
  - Create(Objecttype,Objectdetails)
  - NameVacant(ObjectType,ObjectInformation)
  - Notify(NotifcationType,NotificationInformation)
  - SubmitGitData(BattleDetails)

Back-end modules

- **ScoringServiceI**
  - AutoEvaluateSolution(solution)
  - UploadResults(score,teamID,BattleID)

- **BadgeManagementServiceI**
  - AwardBadge(TournamentID,BadgeID,UserIDs)
  - QueryBadgeLogic(BadgeID)
  - QueryBadgeAchievers(BadgeLogic)
  - Notify(NotifcationType,NotificationInformation)

- **NotificationServiceI**
  - RegisterNotification(NotificationType,NotificationText,RecipientIDs)
  - SendNotifications(NotificationType,NotificationText,RecipientIDs)
  - RequestUserIDs(ObjectInformation)
  - RequestDeadline(BattleID)
  - RequestResults(BattleID)

2.6 Selected architectural styles and patterns

2.6.1 three-tier architecture

The system adopts a three-tier architecture differentiating between:

- **Presentation (User interface tier)**
  - **Responsibility**: The presentation tier is the topmost layer and is responsible for presenting the application's user interface and handling user interaction.

  - **Components**: This tier includes components such as user interfaces, graphical elements, and client-side logic.
  - **Interaction**: It interacts directly with end-users, collecting user input and displaying results.

- **Application logic (Business logic tier/middle tier**)
  - **Responsibility**: The application tier contains the business logic or application logic that processes user requests, performs application-specific functionality, and manages the communication between the presentation and data tiers.
  - **Components**: This tier includes server-side logic, application servers, and by extension the external GitHub Server for processing business rules and workflows.
  - **Interaction**: It communicates with both the presentation tier, the data tier, and the external service of GitHub Actions, orchestrating the flow of data and application functionality.

- **Data management (Database tier)**
  - **Responsibility**: The data tier is responsible for managing and storing data. It stores and retrieves data based on requests from the application tier.
  - **Components**: This tier includes databases, data storage systems, and any components related to data storage and retrieval.
  - **Interaction**: It interacts with the application tier to store and retrieve data as needed.

2.6.2 Event-Driven Architecture

To manage the general flow of the system an event-driven architecture is adopted. In an event-driven architecture three subsystems are considered:

- **Publisher:** The individually forked Battle repositories act as the Publisher of events in this system. This means that the amount of Publishers for each Battle is equal to the number of participating teams. The forked repositories have the ability to trigger one certain event; **push**.
- **Event Broker:** The Event Broker in our system is embodied by the Github Actions Service (GAS). When a push event occurs, GAS forwards the event to the "subscribers".
- **Subscriber:** The Subscriber in our system is the concrete GitHub Action Replica, that runs certain code whenever an event occurs. In our case, the subscriber will check if the submission is made within the time limit, making it eligible for scoring, and if so run the scoring service.

2.7 Other design decisions

- **Web Application Development Platform**: The CodeKataBattles web application will be developed using the Streamlit platform. Streamlit provides a comprehensive set of tools, libraries, and frameworks that streamline web development, ensuring efficiency.
- **Scalability**: The system is designed to scale vertically by upgrading the machine the system runs on. Scaling horizontally would require implementing a load balancer, which is infeasible for this minimum viable product, but the best approach when further scaling the platform.

- **Security**: HTTPS is enforced for secure communication, and user authentication is handled using OAuth through GitHub credentials. We have decided to funnal all write functionality from the UI through a single service in the system (Educator Tool Service), in order to decrease the sources of potential SQL-injections.
- **Caching**: Caching mechanisms are implemented to optimize data retrieval and enhance system performance.

2.7.1 Availability

As described in the RASD document, the requirements of the availability of our system are not tremendous. This is in part due to the relatively low importance of this type of software, as compared to software handling critical infrastructure, sensitive data or life supporting systems. However, due to the chosen design of the system, most of the information related to ongoing battles, such as the Battle-tests, Submissions, Version control and so on, is all stored on Github's infrastructure. This means that, in the case of a breakdown, all Students in on going Battles, can continue their work, relatively unaffected, and simply receive their evaluations when the service returns. All submissions are timestamped in Github's systems, so all submissions made before the deadline will still be regarded, even if the deadline has been passed during an outage.

2.7.2 Data Storage

This architectural design aims to provide a scalable, secure, and maintainable platform for CodeKataBattles, aligning with the specified requirements.

3 User Interface Design

For all visualization of the intended User Interface, we refer to the RASD document.

4 Requirements Traceability

This section establishes the traceability matrix between the functional requirements of CodeKataBattles, identified in the Requirements Analysis and Specification Document (RASD) and the corresponding components, modules, and features in the Design Document (DD). This matrix ensures that each requirement is addressed and implemented in the architectural and design decisions.

Requirements:	[R1] The system shall allow users to log in using their GitHub credentials
Components:	• WebApp • WebServer – Frontend Service – GitHub Management Service • Application Server – Authentication Service

Requirements:	[R3] The Educators shall be able to create tournaments with a specified title. [R4] Educators shall be able to set registration deadlines for tournaments. [R5] Educators shall be able to define tournament badge criteria. [R24] Educators shall be able to define gamification badges with specific rules. [R6] The system shall notify users of new tournaments. [R7] Users shall be able to subscribe to tournaments by a given deadline. [R26] Educators shall be able to close tournaments and trigger final rank notifications. [R27] The system shall enable boolean expressions to define rules of badges and achievements.
Components:	• WebApp • WebServer – Frontend Service – Educator Tools Service • Application Server – Badge Management Service – Notification Service • DBMS

Requirements:	[R8] Educators shall be able to create battles within tournaments. [R9] Educators shall be able to upload technical documents for battles. [R10] Educators shall be able to set the minimum and maximum number of students per group for battle. [R11] Educators shall be able to set a registration deadline for battles. [R12] Educators shall be able to set a final submission deadline for battles. [R13] Educators shall be able to configure scoring methodologies for battles. [R14] Students shall be able to form teams to join battles.
Components:	• WebApp • WebServer – Frontend Service – Educator Tools Service • Application Server – Notification Service • DBMS

Requirements:	[R15] The system shall provide a GitHub repository link for the code kata to registered teams after the registration deadline.

Components:	• WebApp
	• WebServer
	  – Frontend Service
	  – Github Management Service
	• Application Server
	  – Notification Management Service
	• DBMS

Requirements:	[R16] The system shall automatically score submissions based on the results of test cases. [R17] The system shall evaluate the timeliness of submissions. [R18] The system shall assess the quality level of source code through static analysis.
Components:	• Application Server – Scoring Service • DBMS

Requirements:	[R19] Educators shall have the option to manually score submissions.

Components:	• WebApp
	• WebServer
	  – Frontend Service
	  – Educator Tools Service
	• DBMS

Requirements:	[R20] The system shall update battle scores in real-time upon new commits on GitHub.
Components:	• WebApp • WebServer – Frontend Service – Github Management Service • Application Server – Scoring Service – Notification Management Service • DBMS

Requirements:	[R21] The system shall notify participants of final battle ranks after the consolidation phase.
Components:	• WebApp • WebServer – Frontend Service – Educator Tool Service • Application Server – Notification Service • DBMS

Requirements:	[R22] The system shall update personal tournament scores with the sum of battle scores. [R23] The system shall maintain a visible tournament rank for each student.
Components:	• WebApp • WebServer – Frontend Service • DBMS

Requirements:	[R25] The system shall display badges on student profiles. [R28] Students shall be able to visualize their performance. [R29] Students shall be able to visualize their tournament information. [R30] Students shall be able to track their battle involvement. [R31] Students shall be able to see detailed logs of their attempt outcomes
Components:	• WebApp • WebServer – Frontend Service • Application Server – User Profile Service • DBMS

5 Implementation, Integration & Test Plan

This section will specify a plan for implementing the CodeKata Battle system. This includes the specification of system features, the order of their implementation, a strategy to integrate these components into a system, and finally a strategy to test the system as a whole. These tests aim to increase the robustness of the system before, getting into the user's hands.

5.1 Implementation Plan

To ensure implementation is going in the right direction we will be using an iterative development strategy. Here components will be tested as a unit and how they fit into the current iteration of the systems as it is developed. As the CodeKata Battle system consists of few components relative to most systems, we thought a hierarchical approach would be the best fit. Here the components are implemented based on a hierarchical view of the components relations.

Specifically, we have chosen to employ a top-down approach, as it enables us to tackle integrating some of the more complex components like Github actions. Tackling this problem early in implementation is meant to decrease the likelihood of problems later in the integration process. Going top-down also means tackling the issues of user interfaces and high-level functionalities early on in the development process. Enabling early visualization of progress and feedback from stakeholders. A hierarchal approach in general allows for teams to work in parallel, which is key for avoiding potential bottlenecks.

5.1.1 Technology Stack

5.2 Feature Identification

In this section, we have listed all features comprising the CKB system. This is extracted from the system requirements, and the requirements in the RASD.

[F1] User Authentication and Profile Management

- Basic functionalities for account creation and login.

[F2] Battle Creation and Management

- Educators can upload code katas, including descriptions and test cases.
- Setting parameters for challenges: group size limits, deadlines, and scoring configurations.

[F3] Tournament Creation and Management

- Creation and configuration of tournaments by educators.
- Notification system for new tournaments and student subscriptions.

[F4] Student Battle Participation

- Enrollment and team creation for battles within size limits.
- Forking and setting up GitHub repositories for code submissions.

[F5] Real-Time Feedback and Scoring

- Automated evaluation of code based on test cases, timeliness, and code quality.
- Optional additional manual scoring parameter by educators.

[F6] Tournament and Battle Score Aggregation

- Automated calculation of individual battle scores.
- Compilation of tournament scores from individual battle performances.

[F7] Gamification and Badges

- System for educators to create badges.
- Automatic assignment of badges based on rule fulfillment.

[F8] Integration with External Services

- GitHub integration for code repository management and automated testing.

[F9] User Notification System

- Automated notifications for updates, deadlines, and results.

[F10] Profile Visualization of Achievements

- Feature to display collected badges on student profiles.
- Visibility of ongoing tournament details and individual rankings.

5.3 Integration Plan

5.3.1 Component Integration

Here is a prioritized list of orders in which we intend to implement components following the top-down approach stated in the implementation strategy. Unit testing will require the creation of test stubs to accurately simulate components functionality. Unit testing is intended to be done after each component is finished, whereafter an integration test is performed to ensure components interaction perform as expected.

[C1] Front-end Service

- The Front-end Service is the primary user interface, and for this reason, it is important for initial user feedback and specifying the requirements for backend services.

[C2] Data Persistence Service (DBMS)

- As the backbone of the system, storing all persistent data, the database is integrated last to support all other services, once there is a clear understanding of the data requirements.

[C3] GitHub Management Service

- Manages the creation and updates of GitHub repositories, integral to the platform's code submission and review process. Its integration follows the Notification Management Service to ensure that repositories are managed alongside user notifications.

[C4] Authentication Service

- Authentication Service is a core functionality to the very first user interface users interact with, and vital to security and user management. This service must be integrated early to facilitate secure access and testing of subsequent components.

[C5] Scoring Service

- Integral to the competitive aspect of the platform, the Scoring Service is developed after core functionalities to allow for contextualized scoring within tournaments.

[C6] GitHub Actions

- GitHub Actions underpin the automated testing functionality and are critical to operations. They are integrated after the essential user and battle management services are established.

[C7] User Profile Service

- Closely tied to user interaction, this service is integral for personalization and displaying user-specific data, and therefore prioritized early in the process.

[C8] Educator Tools Service

- This service enables educators to create content, a core functionality of the platform. Its integration is prioritized to test the creation and management of challenges.

[C9] Notification Service

- Notifications keep users engaged and informed. This service is prioritized following user authentication to ensure effective communication within the platform.

[C10] Badge Management Service

- As part of the gamification strategy, this service is integrated after the Notification Service to utilize the infrastructure for user communication.

5.4 System Testing

After having integrated all components together, we will have a system that can be tested as a whole. System testing intends to ensure the system as a whole meets all functional and non-functional requirements. To do this the system must undergo several different types of testing phases as laid out in the following section.

Functional Testing: This phase checks if the system complies with all the requirements specified in the Requirement Analysis and Specification Document (RASD). This will be done by trying to execute the use case scenarios and see if the system complies with the intended outcomes.

Performance Testing: The system will be evaluated to identify any potential performance bottlenecks, such as inefficient elements that may impact response times or resource utilization. To check this testing the system under the expected workload will be done.

Usability Testing: To ensure the system is both intuitive and easy-to-use, we'll have a person outside the development team evaluate the system.

Load Testing: The system will be tested under increasing loads to discover bugs that may not surface under normal operation, such as memory leaks or buffer overflows. It will also help determine the system's maximum operating capacity over extended periods.

Stress Testing: This testing ensures the system can handle and recover from adverse conditions. It involves overwhelming the system's resources or depriving it of them to see how it behaves under extreme stress.

6 Efforts Spent

The tables below indicate how much time each participant has spent on each of the sections in the report.

6.1 Karl Kieler

Section	Time Spent (Hours)
Introduction	1
Architectural Design	6
User Interface Design	2
Requirements Tracability	4
Implementation, Integration & Test Plan	20

6.2 Leonie Dragun

Section	Time Spent (Hours)
Introduction	1
Architectural Design	20
User Interface Design	2
Requirements Tracability	12
Implementation, Integration & Test Plan	2

6.3 Aske Schytt Meineche

Section	Time Spent (Hours)
Introduction	2
Architectural Design	24
User Interface Design	2
Requirements Tracability	5
Implementation, Integration & Test Plan	3

7 References

1. Diagrams made with: StarUML and Microsoft PowerPoint
2. Mockups made with Microsoft PowerPoint