Design Document

Politecnico di Milano
Merlino Lorenzo & Iodice Andrea

December 18, 2023

Contents

1 Introduction

1.1 Purpose

The System to Be will allow users to take part in coding battles. Each coding battle will be part of a tournament and consists of a coding project to be completed. Educators will create tournaments and create battles by supplying the project structure. Students will be able to create teams and the teams will be able to submit a solution to the coding battle. Each solution will be scored using different criterias, such as the amount of test cases passed, amount of time required to submit the solution and quality of the code provided. Some criteria will be automatically asserted and the remaining ones will be asserted by the creator of the battle.

1.2 Scope

CodeKataBattle (CKB) is a new platform that helps students improve their software development skills by training with peers on code kata.

Educators use the platform to challenge students by creating code kata battles in which teams of students can compete against each other, thus proving and improving their skills.

Educators create tournaments to which students can enroll. After enrolling in a tournament students can take part in code kata battle by joining or creating a team.

A code kata battle is a programming exercise. The exercise includes a brief textual description and a software project with build automation scripts that contains a set of test cases that the program must pass.

Teams participating in a battle are expected to follow a test-first approach and develop a solution that passes the required tests. Groups deliver their solution to the platform before the deadline. At the end of the battle, the platform and the educators assign scores to groups, in order to create a competition rank.

A Educator cannot be a student and vice versa.

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Coding Battle**: programming exercise proposed by an Educator in the context of a tournament.
- **Tournament**: a series of coding battles, in which Students can take part as teams. It's created and managed by an Educator.
- **Team**: group of students who enroll in a tournament. Students in the same team work together and the solution they submit will be taken into consideration for each member of the team.
- **Rank**: position of a Student in a tournament's leaderboard, which is calculated on each Student's score.
- **Score**: points gained by a Student by submitting a solution to a Coding Battle.
- **Open battle**: a battle that accepts new submissions.
- **Closed battle**: a battle that doesn't accept new submissions but some still need to be evaluated.
- **Completed battle**: a battle that doesn't accept new submissions and where all the submissions have been evaluated.
- **Open tournament**: A tournament where new battles can be created.

- **Closed tournament**: A tournament where new battles cannot be created but some are still not completed.
- **Completed tournament**: A tournament where new battles cannot be created and all battle have been completed
- **Team leader**: Student who created the team.
- **Automated grading**: A series of scores that is automatically assigned to a solution, it is composed of:
  - Number of test passed
  - Time passed after the battle publication
  - Static analysis result
- **User**: either a Student or an Educator
- **Educator**: user of the system, who manages tournaments and battles.
- **Student**: user of the system, who participates in tournaments and battles.

1.3.2 Acronyms

- **RASD**: Requirements Analysis and Specification Document
- **DD**: Design Document

1.3.3 Abbreviations

- [**Rn**]: the n-th requirement
- [**UCn**]: the n-th use case

1.4 Revision History

- Version 1.0

1.5 Reference Documents

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

1.6 Document Structure

Mainly the current document is divided in 7 chapters, which are:

1. Introduction: The first chapter includes the introduction in which is explained the purpose of the document, then, a brief recall of the concepts introduced in the RASD is given. Finally, important information for the reader is given, i.e. definitions, acronyms, synonyms and the set of documents referenced.

2. Architectural Design: it includes a detailed description of the architecture of the system, including the high level view of the elements, the software components of CKB, a description through runtime diagrams of various functionalities of the system and, finally, an in-depth explanation of the architectural pattern used.

3. User Interface Design: are provided the mockups of the application user interfaces, with the links between them to help in understanding the flow between them.

4. Requirements Traceability: it describes the connections between the requirements defined in the RASD and the components described in the first chapter. This is used as proof that the design decisions have been taken with respect to the requirements, and therefore that the designed system can fulfill the goals.

5. Implementation,Integration and Test Plan: it describes the process of implementation, integration and testing to which developers have to stick in order to produce the correct system in a correct way.

6. Effort spent

7. References: it contains the references to any documents and to the Software used in this document.

2 Architectural design

2.1 Overview

This section describes the overview of the system from a design standpoint, defining its main components and their relationships

2.1.1 High level view

The CodeKata system is a distributed system with a four-tier architecture. The four tiers are:

- Presentation: displays the user interface and manages the user interactions.
- Web Server: manages the information sent by the presentation layer, is in charge of load balancing and caching.
- Application: handles the system logic managing authentication and requests from the presentation layer. It filters access to the data layer
- Data: maintains all persistent data of the system and the shared data between different instances of the application layer

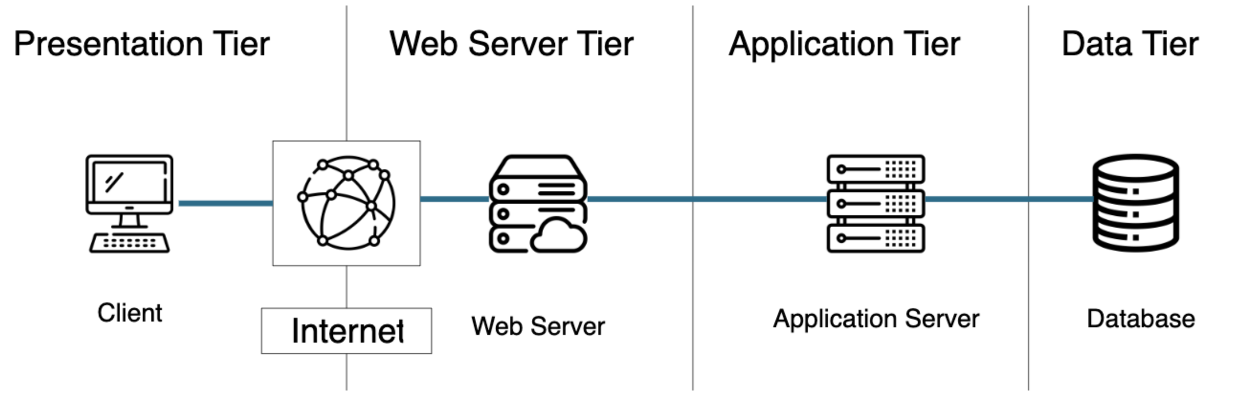

Figure 1: High level view

The communication between client and server through http requests and responses. The system can be accessed via browser by both Students and Educators.

The presentation tier communicates with the web server layer which is in charge of distributing the request between the multiple application layer instances.

The application tier elaborates the requests and access the data from the data layer. The application tier interfaces with the GitHub api on the user behalf.

User session is identified a by JWT, Json Web Token. it contains the user internal ID, the user Github username and the user Github access token

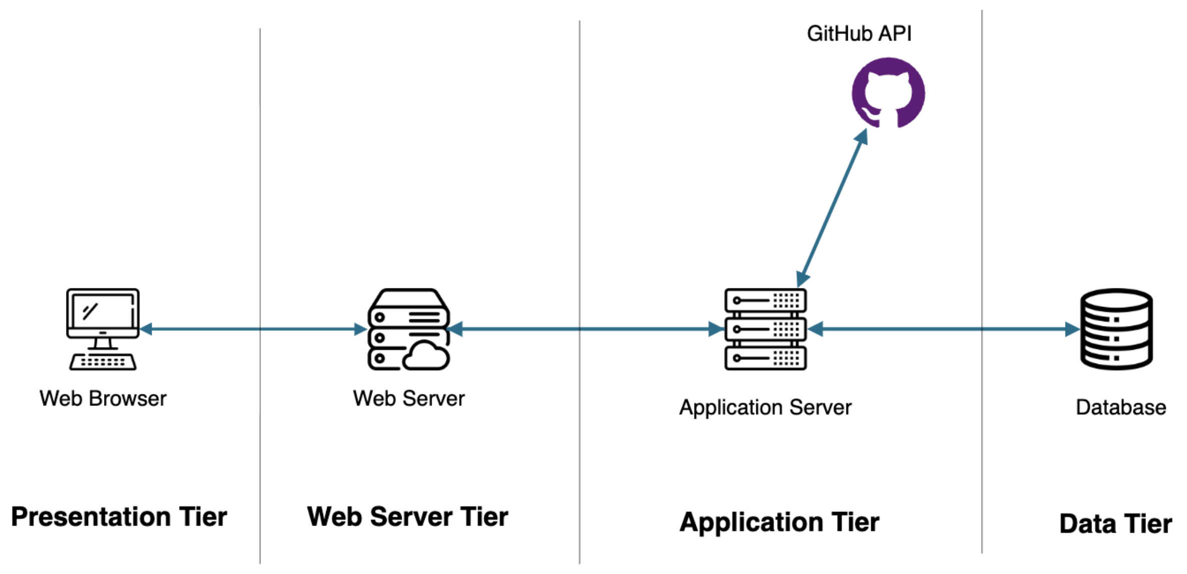

Figure 2: High level view

2.1.2 Distributed view

The system has multiple instances of the application tier to meet availability and performance requirements.

2.2 Component view

The system is divided into different components, the following diagram describes which components are in the system and how they are connected between each other.

External components that are not part of the system are colored white. Presentation components are colored purple. Web server components are colored green. Application components are colored yellow. Data components are colored blue.

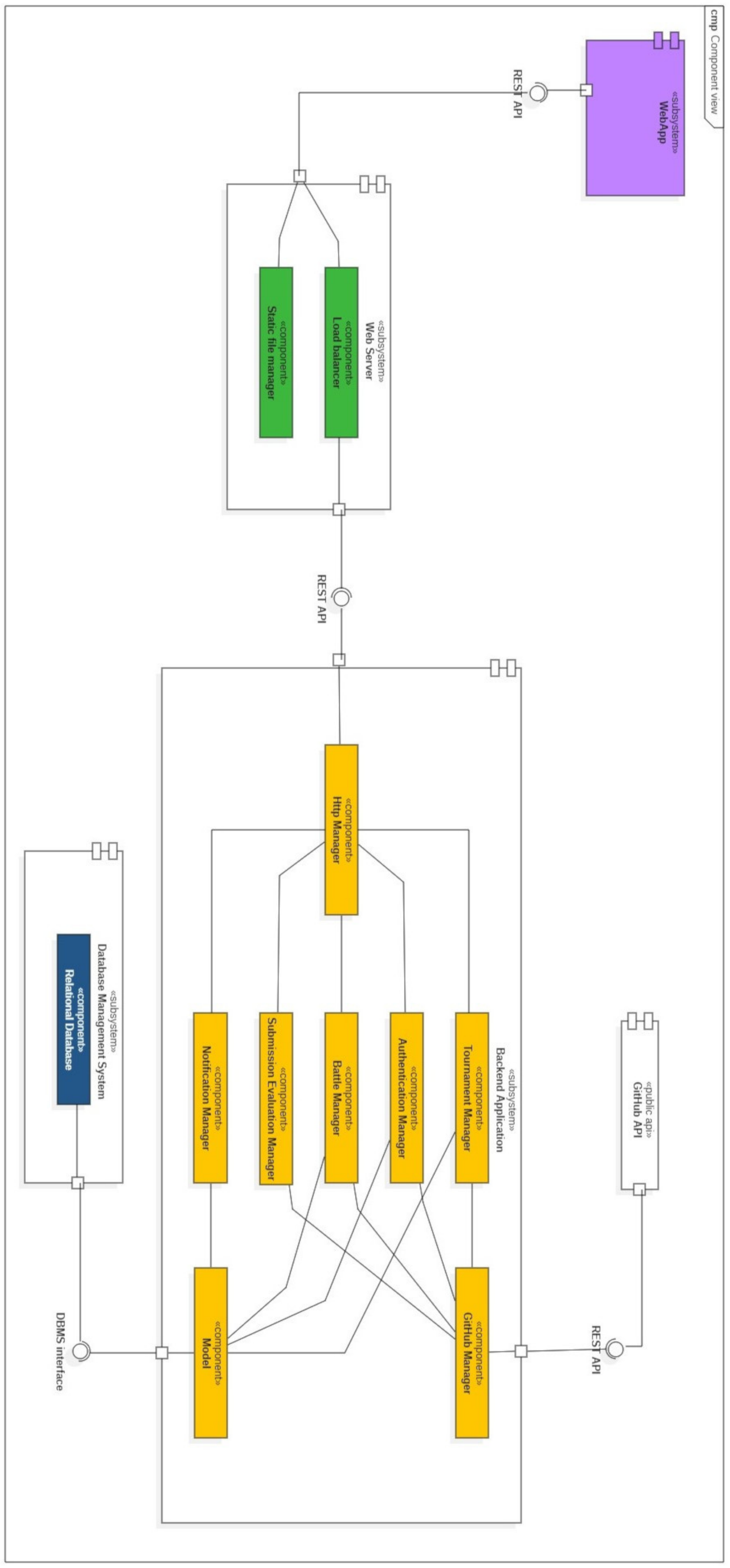
cmp Component view
«subsystem» WebApp
REST API
«subsystem» Web Server
«component» Load balancer
«component» Static file manager
REST API
«subsystem» Backend Application
«component» Http Manager
«component» Tournament Manager
«component» Authentication Manager
«component» Battle Manager
«component» Submission Evaluation Manager
«component» Notification Manager
«component» GitHub Manager
«component» Model
«public api» GitHub API
REST API
«subsystem» Database Management System
«component» Relational Database
DBMS interface

Figure 3: Component view

2.2.1 The Relational Database

contains all the relevant data to the application and it resolves the queries received from the model component with the requested data.

2.2.2 The Model

is used to store the data that are part of the persistent state of the system (which in our case is stored on a database) when this is loaded into the application. Moreover, it provides all the methods that are necessary to access those data and it also implements the logic to manipulate them. In other words: whenever some data contained in the database is needed by any component of the system, for any kind of computation, they delegate the task of retrieving such information to the model component. It not only retrieves them but it also stores them in ad-hoc data structures so that they are ready to be used for the required computations. The model component is the only one interacting with the Relational Database.

2.2.3 The GitHub Manager

manages all interactions between components and the GitHub api.

2.2.4 The Notification Manager

registers the notification's creation and manages its internal status until deletion. It interfaces with the model to record new notifications, retrieve existing ones and delete old ones

2.2.5 The Authentication Manager

handles the user login and interfaces with the GitHub Manager to verify the user access token. It interfaces with the model to record the access token or retrieve user info.

2.2.6 The HTTP Manager

handles http requests and manages the http sessions.

2.2.7 The Battle Manager

handles battles' creation and updates their status when needed. When creating a battle it verifies that the battle configuration is valid, handles teams creations, teams joins and team invites. It interfaces with the Notification manager to notify invites to join a team, when a Student joins a team and when a battle is created or closed. Interfaces with the database to record battles and teams data.

2.2.8 The Tournament Manager

handles the creation of tournaments, the additions of allowed educators into the tournaments, the enrollment of students into tournaments and the status of the tournaments. It interfaces with the database to record and retrieve tournament data.

2.2.9 The Submission Evaluation Manager

handles the automatic grading of new submissions of open battles. Handles manual grading of closed battles. Interfaces with the database to record the team gradings.

2.2.10 The Web-App

is the web interface used by users to access the system and interact with it. It sends requests to the Web Server via HTTP and shows the user the responses from the system.

2.2.11 The Web Server

manages incoming requests from the Web-App component and distributes them between the application's instances. Handles the distributions of static files to the Web-App.

2.3 Deployment view

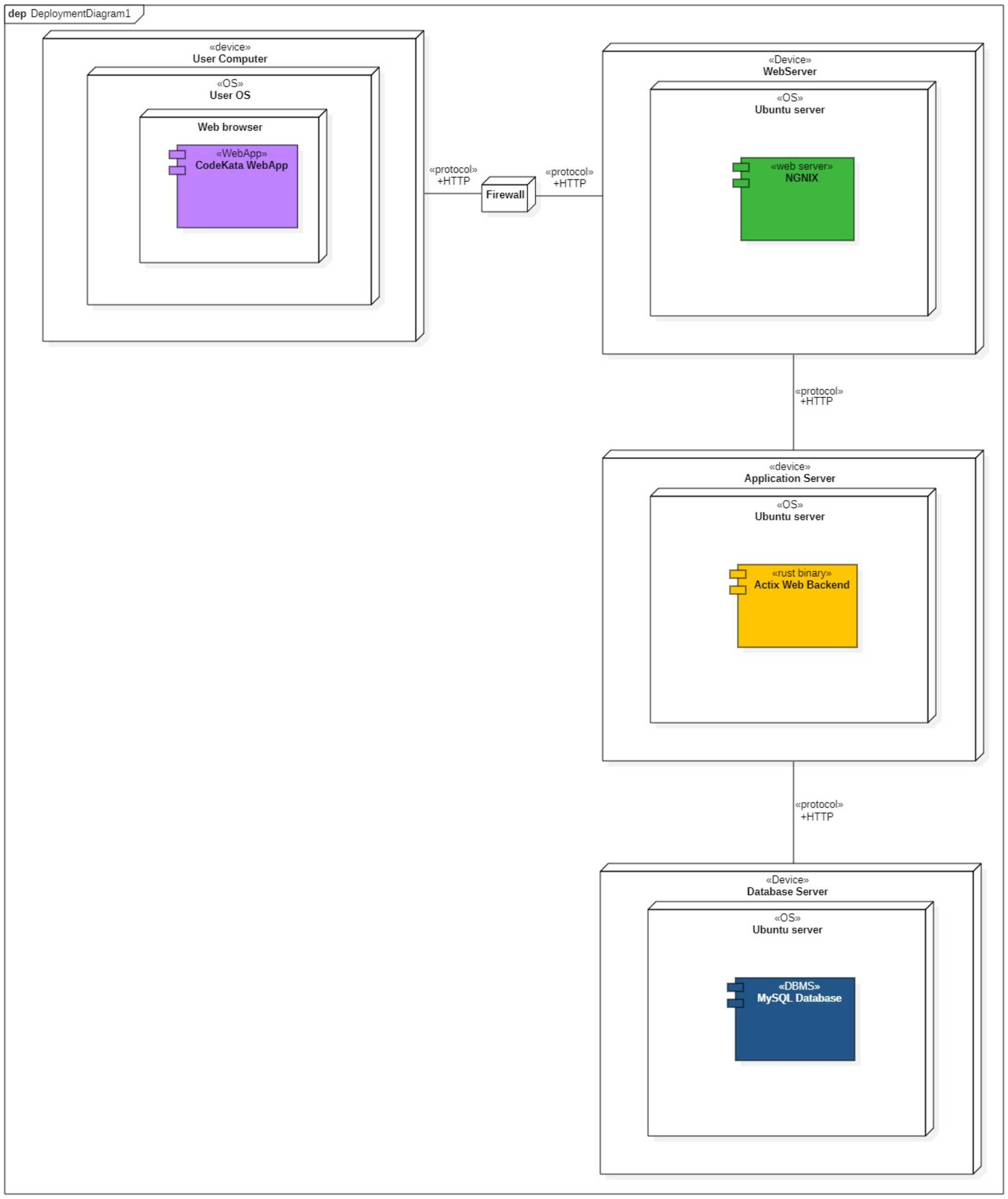

Figure 4: Deployment view

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

- **Firewall** is a device that monitors the packets incoming to the system, if a packet is potentially dangerous it is not forwarder. It is placed before the Web Server and allows incoming packers only if directed to the web server with the correct port and the correct protocol, in this way the only packets that enter the network are considered safe.

- **Computer** is a regular computer used by the user. No special applications are required to be installed on it, except for a web browser. This will be necessary to surf the internet and reach the CKB web page. An HTTPS connection with the system is established by passing through a firewall and after the forwarding, operated by the designated load balancer, to an available application server.
- **Web server** receives all the requests sent from the web app. It then forwards the received request to the different instances of the application allowing for the incoming load to be balanced on the different instances . It also distributes the web app static files and handles the caching of requests when needed.
- **Application server** receives all the requests sent from the web server and elaborates them. It sends requests to the Database server to read and write data when needed. It communicates directly with the GitHub api.
- **Database server** host the relational database that contains all the system data it is accessed by the application server.

2.4 Runtime views

2.5 Component interfaces

2.5.1 Web Server

selectTournament(tournamentID)
Rest API call used to get the tournament view
Param: tournamentID the ID of the tournament
Returns: all data needed to build the tournament view if successful

selectBattle(battleID)
Rest API call used to get the battle
Param: battleID the ID of the battle
Returns: all data needed to build the battle view if successful

createTeam(teamInfo)
Rest API call used create a new team for a battle
Param: teamInfo the info about the new team
Returns: 200 if successful

getInvites(battleID)
Rest API call used to get the invites for the specified battle
Param: battleID the ID of the battle
Returns: list of invites and their id if successful

acceptInvite(inviteID)
Rest API call used to accept an invite to a team
Param: inviteID the id of the invite
Returns: 200 if successful

getAvailableStudents(battleID)
Rest API call used to get the available students for the specified battle
Param: battleID the ID of the battle
Returns: list of students usernames and their id if successful

inviteStudent(newInviteInfo)

Rest API call used to invite a new student
Param: newInviteInfo info regarding the student being invited and the battle
Returns: 200 if successful

2.5.2 Http Manager

selectTournament(tournamentID)
Rest API call used to get the tournament view
Param: tournamentID the ID of the tournament
Returns: all data needed to build the tournament view if successful

selectBattle(battleID)
Rest API call used to get the battle
Param: battleID the ID of the battle
Returns: all data needed to build the battle view if successful

createTeam(teamInfo)
Rest API call used create a new team for a battle
Param: teamInfo the info about the new team
Returns: 200 if successful

getInvites(battleID)
Rest API call used to get the invites for the specified battle
Param: battleID the ID of the battle
Returns: list of invites and their id if successful

acceptInvite(inviteID)
Rest API call used to accept an invite to a team
Param: inviteID the id of the invite
Returns: 200 if successful

getAvailableStudents(battleID)
Rest API call used to get the available students for the specified battle
Param: battleID the ID of the battle
Returns: list of students usernames and their id if successful

inviteStudent(newInviteInfo)
Rest API call used to invite a new student
Param: newInviteInfo info regarding the student being invited and the battle
Returns: 200 if successful

2.6 Selected architectural Styles and patterns

2.7 Other design decisions

2.7.1 Availability

Load balancing and replication are concepts that have been introduced in the system to increase the availability and reliability of the system. In this way the system would be as fault-tolerant as possible regarding data management and service availability.

2.7.2 User Session

The user session is being tracked between http calls thanks to an encrypted JWT in the request cookies. This JWT contains:

- UserID internal id
- Github API user access token
- Github username

the encryption of the JTW is a critical aspect of the system security.

2.7.3 Notifications

New notifications are loaded when the user loads the home page.

2.7.4 Data Storage

A single SQL database is present in the design. This could introduce availability and performance issues, to compensate the database should be deployed on a separated device allowing it to have the maximum ammound of recources. On the other hand having a single SQL database allows the system data storage to preserve ACID proprierties, which would be impossible with a distributed database. Periodicals backup of the database are needed to increase the system reliability.

2.7.5 Rust backend

The rust backend allows the system to handle high ammounts of requests consuming as little resources as possibles. this allows the system to be very efficent as the majority of the requests are mainly interactions with the database and have little data transformation overhead.

3 User Interface Design

4 Requirement traceability

5 Implementation, Integration and Test Plan

6 Effort Spent

Merlino Lorenzo	2h	10h	-h	-h	-h
Iodice Andrea	-h	-h	-h	-h	-h

7 References

- lol