POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

CodeKataBattle - ITD

Software Engineering 2 Project
MSc Computer Science and Engineering

Authors: **Michele Bersani, Paolo Chiappini, Andrea Fraschini**
Reference Professor: **Prof. Matteo G. Rossi**

Student IDs: **10707192, 10743051, 10725610**
Academic Year: **2023-24**
Revision: **v1.0 04/02/24**
GitHub repository: **https://github.com/paolo-chiappini/BersaniChiappiniFraschini**

Contents

1. Introduction

1.1. Purpose

This document serves the primary purpose of providing a comprehensive overview of the implementation choices and testing processes employed during the development of the prototype for the CodeKataBattle platform.

1.2. Scope

The scope of this document is to offer detailed insights into crucial aspects of the software development lifecycle, focusing on the following key areas:

- **Implemented Functionalities**: this section details the functionalities that have been successfully implemented within the CodeKataBattle platform. It outlines the rationale behind the inclusion of specific features and provides clarity on any decisions regarding the exclusion of certain functionalities.
- **Adopted Development Frameworks:** the focus of this section is placed on the chosen development frameworks, including programming languages and middleware. The document discusses the advantages and disadvantages associated with these choices.
- **Source Code Structure:** this section details the structure and organization of the source code. This section offers insights into the architecture and interplay of different components within the codebase.
- **Testing Procedures:** this section covers the testing procedures executed during the development phase.
- **Installation Instructions:** this section provides comprehensive instructions for installing and running the CodeKataBattle prototype. This section includes details on prerequisites, dependencies, and any essential information crucial for a successful installation.

2. Implemented requirements

In this section are presented the implemented requirements as well as the motivations behind their implementation. We will refer to the requirements presented in the RASD using the same notation: **[Rx.y]**.

2.1 Functional requirements

[R1]: The system allows the user to sign up.

This is one of the fundamental requirements for the application as it allows identification of the users and their role by creating an account on the platform.

[R1.1]: The system does not allow registration if the username is already present in the database.

[R1.2]: The system does not allow registration if the email is already present in the database.

Both requirements [R1.1] and [R1.2] are needed to guarantee the *unique* identification of users inside the platform.

[R2]: The system allows the registered user to log in.

[R2.1]: The system does not allow access if at least one of the credentials (username/email or password) is wrong.

Both of these requirements are the dual of requirements [R.1.x] as they allow the user to access the account they have created. Of course, if the user tries to access any account for which they do not know the credentials, the login fails.

[R3] The system allows the student to subscribe to tournaments.

This requirement relates to one of the main features of the platform: allowing students to participate in tournaments. The subscription to tournaments also allows the platform to

keep track of the users therein when sending notifications for updates and when performing consistency checks on incoming requests.

[R3.1]: The system does not allow non-registered users to subscribe to a tournament.

The implementation of this requirement is related to the more general problem of identifying users. Non-registered users are denied access to all features of the platform beside the one presented by requirements [R1.x] and [R2.x].

[R3.2]: The system should not allow a student to subscribe to a tournament they are already subscribed to.

Not performing this check would lead to the potential insertion of the same subdocument inside the database.

[R3.3]: The system should not allow a student to subscribe to a tournament after the subscription deadline has expired.

This requirement, like many others, has been implemented due to its simplicity.

[R4]: The system allows the student to enroll in a battle.

This requirement relates to one of the main features of the platform: allowing students to participate in battles. Enrollment in a battle is fundamental for keeping track of scores and updates from students.

[R4.1]: The system should not allow a student to enroll in the same battle by joining different groups.

Before creating the group to register for the battle, the platform checks whether the student is already an effective member in another group of the same battle

---

[R4.2]: The system allows students to leave the battle before it begins.

This requirement has not been implemented as it is classifiable as a "nice-to-have" feature, and was not required by the project assignment.

[R4.3]: The system should not allow groups that don't meet battle group size requirements to participate (in battles).

This is implemented as it is easy to verify whether the number of students in a group is between the minimum and maximum requirements for the battle.

[R4.4]: The system should not allow groups to enroll in battles outside the enrollment deadlines.

Same as [R3.3].

[R5]: The system allows the student to invite another student to a battle, creating a group.

Together with [R4], this requirement relates to one of the main features of the platform.

[R5.1]: The system should not allow a student to invite another student to a battle if they are already in a group for that battle.

This requirement has not been implemented to allow testing with a few students in the database.

[R6]: The system allows the educator to create a tournament.

This requirement relates to one of the main features of the platform and allows Educator "type" users to create the tournaments that students participate in as described by requirements [R3.x].

[R6.1]: Every tournament should have one and only one owner which is the educator who has created it.

The implementation of this requirement is partially implicit in the tournament creation process: tournaments have unique names and at the moment of their creation the owner is the only educator managing the tournament (invited educators do not count).

[R6.2]: Every educator who owns a tournament should have permission to manage that tournament.

As mentioned above, the owner of a tournament is also regarded as a manager for that tournament from the moment of its creation.

[R6.3]: The creation of a tournament must notify all users on the platform.

This requirement has been implemented due to its simplicity.

[R7]: The system allows the educator to create badges within a tournament.

The gamification aspects have not been implemented as they were not of interest to the scope of the prototype. This includes requirements [R7.1], [R7.2], [R7.3] and [R7.4].

[R8]: The system allows the educator to create a battle within a tournament.

This requirement relates to one of the main features of the platform similar to [R6].

[R8.1]: The system allows the educator to upload the code kata.

This requirement, again, relates to one of the main features of the platform on the educators' side. It has been implemented to allow educators to provide the project at the core of a battle (code kata) and students to interact with the GitHub integration.

[R8.2]: The system allows the educator to set the minimum and maximum number of students per group.

[R8.3]: The system allows the educator to set a registration deadline.

[R8.4]: The system allows the educator to set a final submission deadline.

[R8.5]: The system allows the educator to set additional configurations for scoring.

Like [R8.1], these requirements have been implemented to allow educators to configure all parameters of a battle and make use of features such as the automated assessment provided by the platform.

[R9]: The system allows the educator to grant permissions to another educator.

[R9.1]: The system allows the educator to grant permissions to another educator to create battles.

[R9.3]: The system allows the educator to grant permissions to another educator to give manual evaluations.

All the above requirements have been implemented as a counterpart to the invitation feature already described for students in requirements [R5].

[R9.2]: The system allows the educator to grant permissions to another educator to create badges.

As mentioned for [R7], the prototype does not include the implementation of gamification features, however, it would follow the same reasoning for the other requirements [R9.x].

[R10]: The system allows the educator to make a manual assessment of the solution provided by the groups if specified in the scoring configurations.

This requirement has been implemented to showcase the possibility of giving manual assessments at the end of a battle before the consolidation process. This feature also

relates to the more general problem of grading students' solutions which is one of the main features of the platform.

[R11]: The system must create the GitHub repository containing the code kata for the battle.

[R11.1]: The system must send the GitHub link to all students who are members of subscribed teams.

[R11.2]: The system must make the GitHub repository link visible on the battle page.

[R11.3]: The system must provide an authorization token for the group to make API calls.

This feature, along with all its sub-requirements, has been implemented to showcase the integration of the platform with the GitHub API and the automation provided by the platform.

[R12]: The system must run tests and give an evaluation of the provided solutions.

This requirement also feeds into the desire to showcase the automation features of the platform, although to a limited extent (only for Java projects up to Java 17).

[R12.1]: The system must receive messages, through the CKB platform's API, with each new commit made by the students in the main branch.

This particular requirement is necessary to ensure that the platform provides an endpoint that is accessible from GitHub.

[R12.2]: When tests have been conducted, the system should show the outcomes of tests to users.

[R12.3]: The system must analyze the timeliness of the student's solution, measured in terms of time passed between the registration deadline and the last commit.

[R12.4]: The system must analyze the code by calling ECAs.

[R12.5]: The system must calculate and update the battle score of the team.

All of the above are related to the requirement of automatically grading student's solutions. For the purpose of the prototype, this is achieved, partially, through the use of "Mock classes" that simulate the presence of actual ECA Runners in the case of [R12.4], providing only an example of actual implementation. This last choice was made due to the limited time for development that would have had to include extensive research into the intricacies of language-specific tools and libraries.

[R12.6]: The system should not consider solutions outside the submission deadlines.

This requirement has been implemented as it entails a simple check on the date of the submission of the solution.

[R13]: The system allows the user to see the current rank evolving during the battle.

[R14]: The system must update the personal tournament score of each student, that is the sum of all battle scores received in that tournament, at the end of each battle.

These features have been implemented as a part of the overall scoring system.

[R15]: The system allows the educator who created the tournament to close it.

This requirement has been implemented as it entails a simple check on the (unique) username of the user on the client side.

[R15.1]: The system must notify all the subscribed students of the closing tournament.

[R15.2]: The system must notify all the subscribed students when the final battle rank becomes available.

Both requirements have been implemented as they refer to the same event: the closing of the tournament.

[R15.3]: The system must notify the students in the tournament who have completed the process to obtain a badge of their achievement.

As mentioned for [R7], the prototype does not include the implementation of gamification features.

[R15.4]: The system should not allow educators who aren't hosts of a tournament to close it.

This requirement has been implemented for the same reasons stated for [R15].

[R15.5]: Once a tournament has been closed, it cannot be reopened.

This requirement also has been implemented as it entails a simple client-side check.

[R16]: The system should assign a badge to one or more students at the end of the tournament if the students have fulfilled the badge's requirements to achieve it.

The gamification aspects have not been implemented as they were not of interest to the scope of the prototype. This includes requirements [R16.1] and [R16.2].

[R17]: The system allows the user to view another student's badge

The gamification aspects have not been implemented as they were not of interest to the scope of the prototype.

[R18]: The system should automatically close battles after their submission deadline is reached.

This requirement has been implemented as it is necessary to trigger the automatic evaluation of students' solutions.

[R18.1]: Once the submission deadline for a battle has been reached, the battle cannot be reopened.

This requirement has been implemented as it is implicit in how battles are closed: the closing of a battle is bound to an event that happens once and is in the control of the system, not of the user.

[R19]: The system allows users to search by battle and by tournament.

[R19.1]: The system allows users to search for specific battles.

[R19.2]: The system allows users to search for specific tournaments.

These requirements have been implemented as Spring provides ready-made interfaces that allow it.

[R20]: The system allows invited users to either accept or reject.

[R20.1]: The system allows educators who have been invited as tournament managers to accept the invite.

[R20.2]: The system allows educators who have been invited as tournament managers to reject the invite.

[R20.3]: The system allows students who have been invited in groups for a battle to accept the invite.

[R20.4]: The system allows students who have been invited in groups for a battle to reject the invite.

[R20.5]: The system automatically rejects all pending invitations for a battle if the student joins a group.

These requirements have been implemented as the prototype allows to showcase the formation of groups of students and the collaboration between managers in a tournament.

[R21]: If configured, when the battle ends the system must notify the educators of the tournament that a manual evaluation is required to consolidate scores.

This requirement has been implemented due to its simplicity.

2.2 User interface requirements

The frontend for the prototype only implements a small subset of the requirements stated in the RASD. The main implemented requirements in the available version of the frontend are:

- Labeled inputs for forms;
- Concealment of sensitive inputs;
- Visual language accompanied by a textual description (although not fully);
- Responsive design (to a limited extent).

The decision to implement the above (and exclude the others) stems from prioritizing the backend features over the frontend in the development process..

3. Adopted frameworks

3.1. Frontend framework

For the development of the front-end of CKB, the chosen framework is Vue. The reasons behind this choice are multiple:

- **Easy to use**: Vue provides an easy-to-use framework requiring only basic knowledge of JavaScript, CSS, and HTML. The documentation for Vue is also well-crafted further contributing to providing a low entry barrier to start using the framework.
- **Components structure**: Vue, like many other frameworks, allows to decompose views into basic components that can be reused. As presented in the Design Document (and in the attached mockup), the GUI was designed to feature as many reusable components as possible to expedite its development. This modular approach not only expedites the overall development process but also allows for the independent development of various elements within the main views.
- **Two-way data binding**: Vue allows binding data to views to create reactive and dynamic views.

In conjunction with Vue, we have integrated Vue-router, VueX, and Typescript into our development stack. Vue-router enhances our ability to map views to specific routes and dynamically render them client-side, eliminating the need to request pages from the server.
VueX allows the implementation of client-side storage similar to cookies. This plugin was used for storing the JWT token and user data, but also to exchange data between components like in the case of invites.
Although Typescript is not mandatory, we've chosen to incorporate it for a more structured development environment and enhanced type safety. This decision enables us to clearly define the expected data types from the server, facilitating static checks. With Typescript, we can identify errors and bugs without running the application, contributing to a more robust and error-resistant codebase at the cost of losing part of the flexibility that an untyped language like JavaScript offers and the need to ensure consistency between client and server objects.

3.2. Backend framework

For the development of the back-end, we've chosen Spring for Java as our framework. The choice was made out of the many features that Spring has to offer, namely:

- **Dependency Injection:** Spring's core strength lies in its support for dependency injection through inversion of control. This architectural pattern enhances modularity and flexibility by allowing components to be loosely coupled, facilitating easier maintenance, testing, and scalability of the application. The creation and injection of object instances is automatically handled by Spring allowing us to focus on the business logic and adopt more of an aspect-oriented programming style.
- **Mocking and testing:** Spring provides comprehensive support for unit testing and mocking. The framework simplifies the creation of mock objects using a library called *Mockito* and allows us to create in-memory temporary databases for integration testing.
- **Database management and data mapping:** Spring offers powerful tools for database management and object-relational mapping (ORM). One of the main reasons why we chose Spring is how easy and simple working with databases is.
- **Event handling and task scheduling:** Spring facilitates event handling and task scheduling, allowing the execution of tasks at scheduled times. This capability is essential for building the tournament and battle engine of CKB.
- **Email:** Spring also provides a very easy-to-use email service to send notifications to users.

These as well as other features (such as the various integrations with IntelliJ) made Spring our framework of choice for building the back-end of Code Kata Battle.

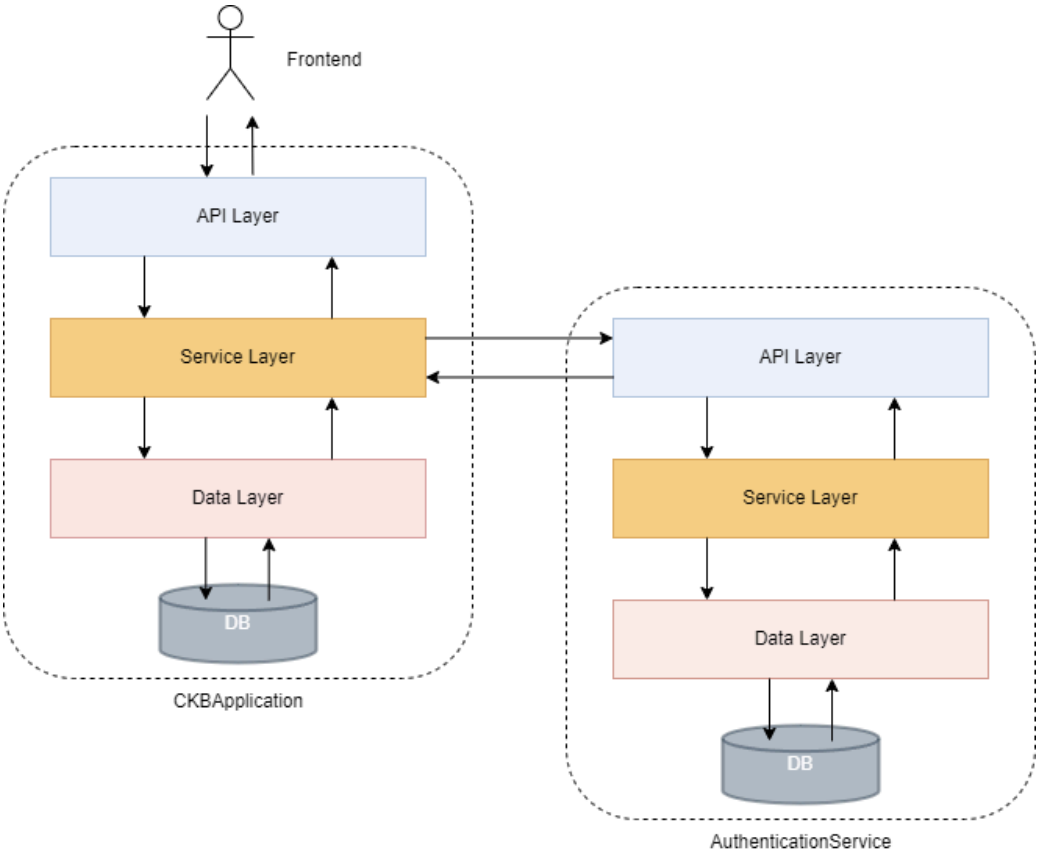
4. Source code structure

4.1. Backend structure

The general structure of the platform was developed on two three-tier structures, as the first structure acts as a server for the entire platform while the second is a microservice that carries out the authentication and registration tasks.

The backend code follows a three-tier structure. The API layer handles the requests and builds and returns the responses using the information returned by the service layer. The service layer contains the business logic of the application. Last but not least, the data layer manages all interactions with the database.

We've decided to organize the source code by entities rather than by tiers. Every source file associated with an entity is kept in the same package as the other files of the same entity. For example: the package *"user"* contains all the files concerning users such as the DTO (Domain Transfer Object) User class, the UserService class, and the UserRepository class.



4.1.1. Database structure

For the development of the platform we used two types of databases:

- **CKBApplicationServer**: we have used the non-relational database MongoDB. The MongoDB technology automatically manages the distribution of data across multiple infrastructures so that the platform can be scalable, thus managing more and more data in a decentralized manner. It also made it possible to facilitate the mapping between the documents stored in the database and the objects used in the server to represent the entities (low impedance mismatch). The benefits of

MongoDB also include the properties of Consistency and Partition Tolerance, sacrificing Availability.

- **AuthenticationService**: in this service, we opted for a relational database like MySQL, as queries and record saving are mainly based on key and value mapping. While we acknowledge the existence of key-value databases like Redis, which could have been well-suited for this task, we opted for a solution that aligns better with our familiarity and expertise.

4.2. Frontend structure

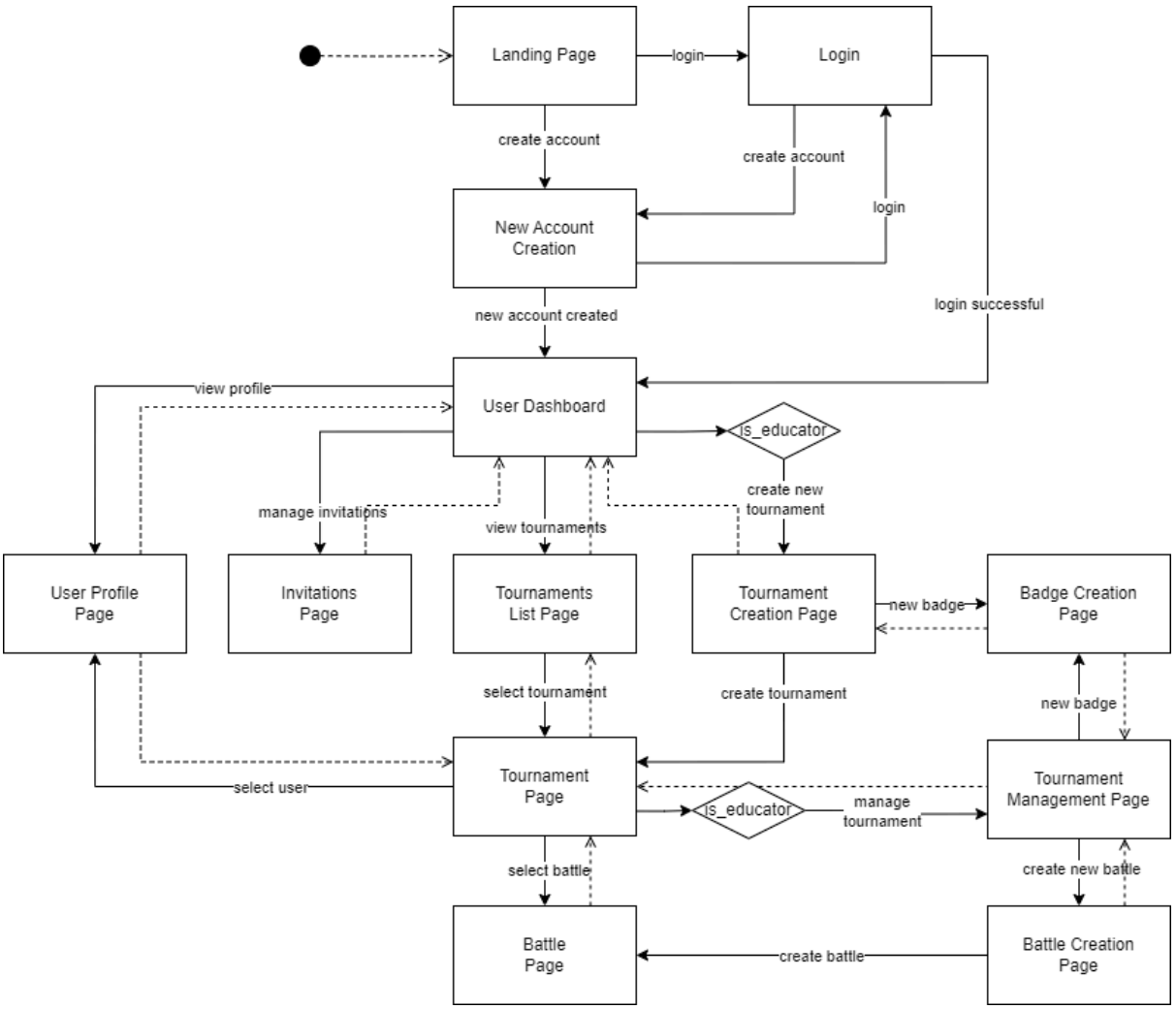
The code for the frontend roughly follows the standard structure of Vue applications and is divided into the following sections:

- **assets**: here are contained static resources such as logos and images, but also the stylesheets shared by all views and components.
- **components**: here are contained the reusable components.
- **router**: here is the logic behind the routes of the web application. Routes are separated into their files that can be found in **/routes**.
- **store:** here is the implementation of the client-side VueX storage used for storing the user's JWT and data, along with the store for the invites.
- **util:** here we can find all the type declarations and general-purpose functions used throughout the views.
- **views:** here are located the main views of the application. Most views are associated with a route and generally include components.

Both views and components are structured following the CompositionAPI of Vue 3 and are divided into 3 main sections:

- **setup (script):** encapsulates all the logic used by the view/component (i.e. requests to the server) along with the data used for binding with HTML elements.
- **template:** structure of the view/component. It contains standard HTML tags, some of which use Vue's tags for data binding, and custom component instances.
- **(scoped) style:** view-specific CSS styling.

As for the routes, the application follows the navigation diagram already presented in the Design Document which is repeated below for convenience. All routes implement a whitelist on the account type to prevent unauthorized access (see **/router/index.ts**).



Note that the User Profile Page, as well as the Badge Creation Page, have not been implemented since they do not provide any utility in the context of the prototype.

5. Testing

In the authentication service, tests were performed on all layers of the server: API, service, and data layer.

For the backend (the application server) tests were performed on the service layer and the data layer due to the very simple nature of the API controllers which delegate most of the work to the lower layers.

During development, we frequently used Postman to test interactions with the routes defined by the controllers of the server while waiting for the front-end to be developed. Unit tests were performed using a mocking library called Mockito which allowed us to mock all the dependencies of the service we were testing by injecting stub implementations of the methods that we could configure to return specific values. For services that update the database we used a memory-embedded database to check the state of the data before and after the functionality we were testing was run to ensure the updates were correct. Line coverage was the main metric we were using to determine the completeness of the tests.

The main features that we tested are:

- Battle creation, enrollment, start and closing
- Score computation and update
- Notification sending
- Invite sending and accepting
- Tournament creation

The functionalities concerning repository creation, fetching from groups repositories and automatic evaluation were tested manually very thoroughly since the interaction with GitHub could not be mocked in a sensible way.

6. Installation

6.1 Frontend

To start using the frontend it is necessary to have NPM (Node Package Manager) installed. Once NPM has been installed, move to the "**`/ckb_webview`**" directory and install the dependencies by typing "**`npm install`**" or "**`npm i`**".

Once all the necessary dependencies have been installed, the frontend is almost ready to be run. First, make sure that the "**`VITE_APP_API_BASE`**" URL in the "**`.env`**" file found in "**`/ckb_webview`**" reports the correct value pointing to CKBApplicationServer.

Once the URL is set, start the application by running the command "**`npm run preview`**" and go to the given localhost URL (http://localhost:4173/ by default).

6.2. Backend and microservice

To enable GitHub Actions to send requests to the backend, it is essential to deploy the server in a manner that makes it publicly accessible. This requirement extends to the microservice and the databases, necessitating their deployment in a publicly reachable server as well. For this purpose, Docker is strongly recommended in case one wants to deploy the servers on a cloud service. Dockerfiles and docker-compose scripts for building images are available in the project's repository along with further instructions and tips in the README.

Many environment variables must be set to allow the application to correctly function:

- Main application database URI (must be a MongoDB database)
- Username and application key of a gmail account (for sending notifications as emails)
- Microservice URL (and port)
- Github API token and owner (for creating the repositories of battles)
- Microservice database URI (must be a MySQL database)
- Microservice database username and password

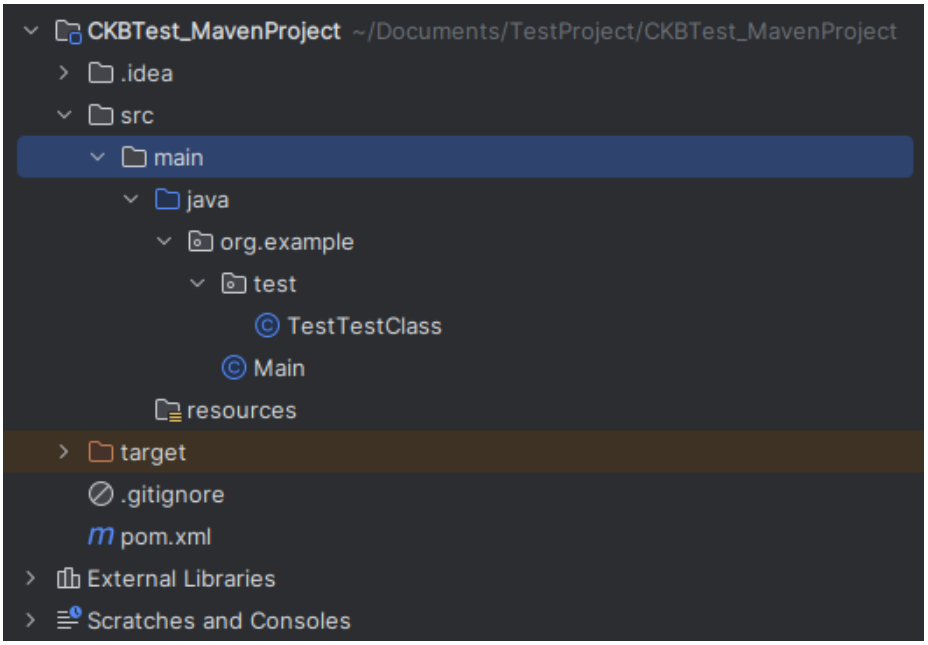
Other properties such as the ports of the servers can be chosen freely (defaults are 8080 for the application server and 8081 for the microservice server) as long as they are

configured correctly everywhere (including the frontend). All these environment variables are configurable in the two *application.properties* files located in the src/main/resources/ folders of the servers and in the *.env* file of the frontend.

6.3. Notes on use

The prototype allows to upload and perform tests on Java projects, although to a limited capacity. To ensure the correct functioning of the features related to code analysis, the uploaded project and files need to follow some requirements:

- The build script file must be called "***build.sh***" and located at the root of the uploaded zipped project.
- The (single) test file must be compiled into the output jar when running the build script. This can be achieved in different ways such as creating a "fat jar" by including the test scope in the compiled files, however, the easiest solution is to put the test file in the same package as the project files. You can see an example below.
- The project directory must contain only one jar file. This is due to how the code performs "file discovery" to find the compiled project.



Example of project structure to include the test file into the jar.

7. Effort Spent

Bersani Michele	
Activity	**Effort spent**
Backend structure	30 min
Database structure	30 min
Total individual effort	**1h**

Chiappini Paolo	
Activity	**Effort spent**
UI framework description	1h
Notes on use	1h
Frontend structure	30 min
Implemented requirements	2h
Total individual effort	**4h 30min**

Fraschini Andrea	
Activity	**Effort spent**
Backend structure	1h
Testing and backend installation	1:30h
Backend structure diagram	30m
Total individual effort	**3:00h**

8. References

- Microsoft Azure cloud services: https://azure.microsoft.com/
- Postman: https://www.postman.com/
- Swagger: https://swagger.io/
- Vue & Vuex: https://vuejs.org, https://vuex.vuejs.org
- JSON Web Token: https://jwt.io
- MySQl: https://www.mysql.com
- MongoDB: https://www.mongodb.com