POLITECNICO
MILANO 1863

SOFTWARE ENGINEERING II

CKB – CodeKataBattle

Implementation And Test

Version 1.0

Feraboli Alessandro, Filippini Marco, Lucca Simone

February 05, 2024

Introduction and scope

This ITD document outlines the software implementation process for the CodeKataBattle project, providing an overview of key aspects:

Purpose:

- Describe our implemented features and functionality.
- Detail the development frameworks and their pros and cons.
- Present the code structure and organization.
- Explain our testing procedures and outcomes.
- Provide clear installation instructions.

Scope:

- Covers the implementation phase of CodeKataBattle.
- Offers insights into choices, design, and testing.
- Enables readers to understand our prototype's state.
- Serves as a reference for acceptance testing.
- Our goal is to deliver a well-documented, functional prototype with a focus on code quality, stability, and adherence to project requirements.

Implemented requirements

1. The system allows users to sign-up and log-in.
   Note: this authentication system was not secured.
2. The system allows each educator to create a tournament at any moment.
3. The system updates, upon the end of a battle, the tournament score of the students who have participated in it.
4. The system allows the tournament leader to add other educators to the tournament at any point in time.
5. The system allows the tournament leader to close a hosted tournament.
6. The system allows an educator to create a battle within the context of a tournament.
7. The system allows students to join a battle only if it has not started yet.
8. The system requires students to enter a group of an acceptable size before the battle starts, otherwise they will be expelled when the battle begins.
9. The battle starts when the timer set upon creation expires.
10. The system creates a GitHub repository for the battle.
11. The system pushes in the appropriate GitHub repository the code KATA previously uploaded by the educator.
12. The system is notified about new pushes by the GitHub Action set up by the group leader.
13. The system evaluates the pushed solution giving a temporary score after each push.
14. The ranking is shown at the end of the battle.
15. The system allows group leaders to invite peers in the group if the battle has not started yet. Students can be invited even if they are participating in another group, but they need to choose only one.

16. The system stops students from entering groups that have reached the maximum number of participants.
17. The system registers as the score obtained in a battle the last score calculated (which corresponds to the score of the last push on the repo).

Adopted Development Tools

Programming languages:

Backend: JAVA
Advantages:

1. Platform Independence: Ensures wide accessibility across different systems.
2. Robust Frameworks: Offers powerful tools like Spring Boot for efficient development.
3. Strong Community Support: Provides extensive resources and support.

Disadvantages:

1. Verbose Syntax: May lead to longer development times, mitigated using Lombok.

Frontend: JAVASCRIPT

Advantages:

1. Universal Compatibility: Runs on all modern web browsers seamlessly.
2. Dynamic and Interactive UIs: Enables creation of responsive and interactive user interfaces.
3. Rich Ecosystem: Offers frameworks/libraries like React, for robust development.

Disadvantages:

1. Browser Inconsistencies: Can behave differently across browsers.

Frameworks and tools:

Spring Boot: We adopted the Spring Boot framework to facilitate the development of our backend services. Spring Boot offers a robust and efficient platform for building RESTful web applications, and it greatly simplifies tasks such as setting up the application, handling HTTP requests, and managing dependencies.

Spring Data JPA: Within our microservices, we adopted Spring Data JPA, an integral part of the Spring ecosystem, to simplify data access and persistence. Spring Data JPA provided an abstraction layer over JPA (Java Persistence API) and allowed us to work with databases in a more streamlined and efficient manner. It promoted the use of standard JPA annotations, such as @Entity, @Repository, and @Transactional, to manage data entities and transactions.

Maven: We used Apache Maven as our build and dependency management tool. Maven simplified project configuration, build automation, and dependency resolution. It ensured consistent and efficient project builds across our microservices.

Kafka: Kafka played a pivotal role in enabling real-time data streaming and event-driven architecture within our application. Kafka provided us with the means to efficiently handle asynchronous communication and data synchronization between various components of CodeKataBattle. This choice is in line with the growing importance of event-driven architectures in modern software systems. Apache Kafka was preferred to other alternatives like RabbitMQ for its ability to handle high throughput. Kafka has also a lot of functionalities that enable the implementation of at least once delivery and at most once delivery since the data is persisted in the replicas. In our prototype we did not focus on this type of guarantees but it is good to know this as a point of improvement.

Eureka: Eureka is used to facilitate service discovery in a microservices architecture. It serves as a distributed registry where each microservice registers itself, allowing other services to dynamically discover and communicate with each other. This enhances the system's scalability and resilience, as services can be added, removed, or moved without disrupting the overall application.

Spring Cloud Starter Gateway MVC: We leveraged the Spring Cloud Starter Gateway MVC to implement the gateway component of our microservices architecture. This framework enabled us to manage routing, load balancing, and API composition effectively. It plays a crucial role in ensuring the scalability and resilience of our system while also simplifying API management.

Static Analysis API (To Be Completed): We have integrated a static analysis API into our system to ensure code quality and adherence to coding standards. This API, while not specified in detail here, will be a critical component in maintaining code integrity and facilitating efficient code reviews. The tool we use to do this is SonarQube.

GitHub API: We utilized the GitHub API to integrate with the GitHub platform seamlessly. This integration allowed us to create repositories, push code, and interact with GitHub's extensive developer tools programmatically. This approach aligns with our goal of enhancing the development workflow for educators and students.

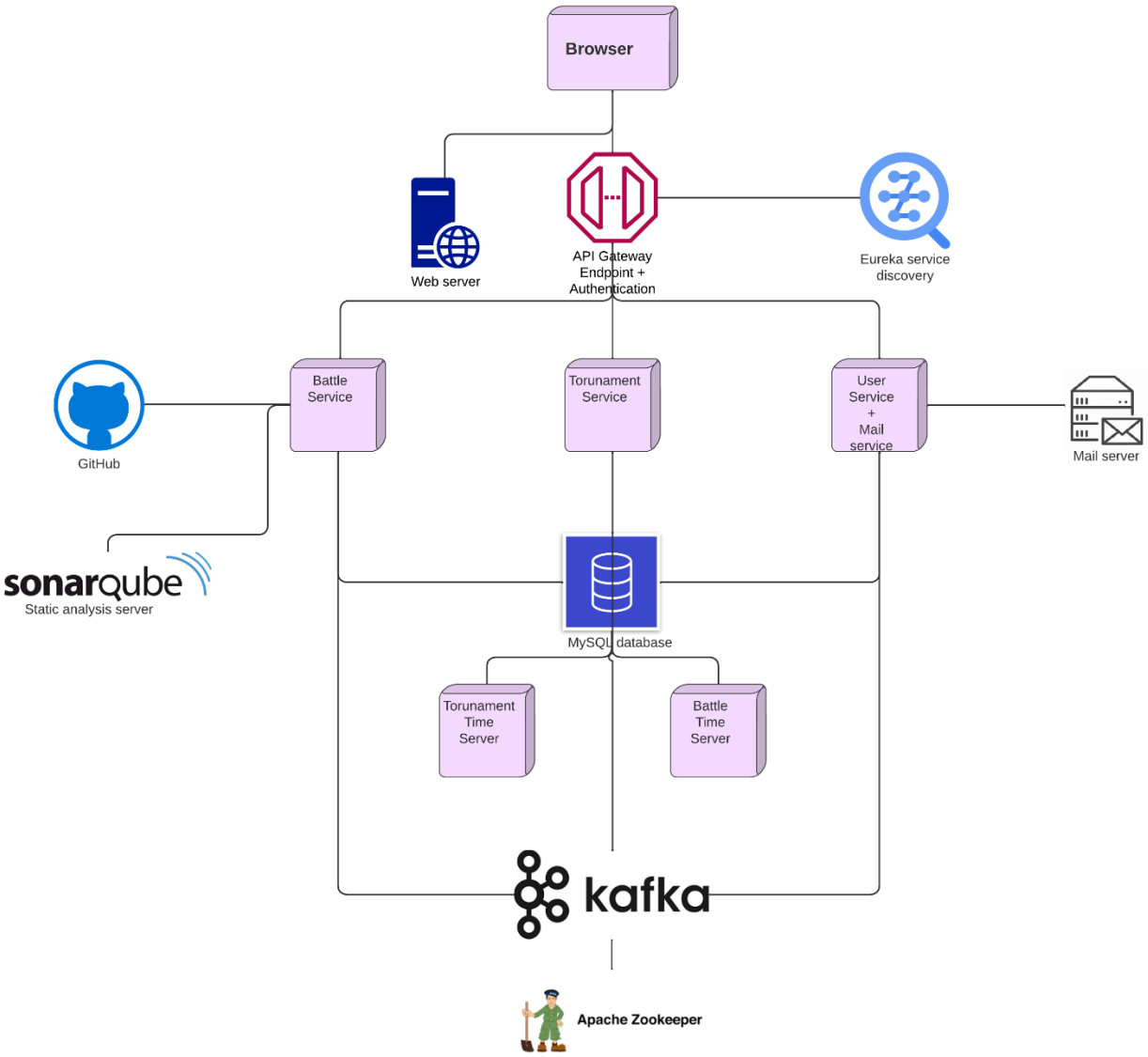
Lombok: Lombok is a valuable library that simplifies Java code by generating boilerplate code, such as getters, setters, and constructors, through annotations. Its usage in our project contributed to cleaner and more concise code, improving code maintainability and readability.

These adopted frameworks and tools were selected to enhance the efficiency, scalability, and maintainability of our CodeKataBattle implementation. Each framework serves a specific purpose within our architecture, ensuring that we meet the requirements of our project while adhering to best practices in software engineering.

Structure of the source code

In the CodeKataBattle project, our microservices are organized into separate Spring Boot projects, each adhering to a consistent directory structure and organization. This section provides an overview of the source code structure for each microservice, highlighting the common elements and patterns we have followed.

In the following image, each component represents a process or an api that must be active for the correct function of the system. Note that, even if the database was deployed on a single process, the database schemas are different, thus enabling the easy deployment of the database on different machines.



Directory structure:

Each microservice project follows a similar directory structure for consistency and ease of navigation. Below is a high-level overview of the common directories and their purposes:

src/main/java: This directory contains the Java source code for the microservice.

com.polimi.ckb.service-name: The root package for the microservice, following Java package naming conventions.

- config: Configuration files and classes specific to the microservice.
- controller: Controllers responsible for handling incoming HTTP requests.
- service: Service classes that encapsulate business logic.
- dto: data transfer objects, classes that serve the purpose of communication.
- repository: Data access classes and repositories for database interaction.
- entity: jpa entities related to database tables.
- exception: custom exceptions .
- utility: Entity classes representing data structures.

src/main/resources: Configuration files and resources required by the microservice.

src/test: Contains unit and integration tests for the microservice.

Testing

Unit Testing:

For our unit testing, we chose JUnit. This tool helped us write and run tests easily, checking if different parts of our code worked as expected. We wrote test cases, which are specific checks to see if our code gives the right output for given inputs. JUnit made it simple to spot where our code had issues, so we could fix them quickly. It also worked well with our development tools, making the whole testing process smooth and giving us fast feedback on our code's quality.

Integration Testing:

Integration testing was performed to validate the interactions and collaborations between different components within the application. Key integrations, such as those involving Spring Boot controllers, services, and database access, were thoroughly tested to ensure seamless communication and data flow. We also utilized the Spring Test framework to facilitate integration testing in a controlled environment.

Acceptance Test Case: Educator Creates Tournament

Test Objective:

1. Confirm that educators can create tournaments in the CodeKataBattle application.

Preconditions:

1. The educator is logged into their account.

Test Steps:

1. Navigate to the "Tournaments" section.
2. Click on the "Create Tournament" button.
3. Fill in the tournament details, including name (unique), date.
4. Click the "Create" button.

Expected Result:

1. A new tournament should be created and listed in the "Tournaments" section.

Acceptance Test Case: Subscription to a Tournament

Test Objective:

1. Verify that users can successfully subscribe to a tournament in the CodeKataBattle application.

Preconditions:

1. The CodeKataBattle prototype is installed and running.
2. The user has logged into their CodeKataBattle account.

Test Steps:

1. From the application's main dashboard, navigate to the "Tournaments" section.
2. Browse the list of available tournaments and select one that the user wishes to subscribe to.
3. Click on the selected tournament to view its details.
4. Find the "Subscribe" or "Join" button and click on it.
5. Confirm the subscription when prompted.
6. Verify that a confirmation message is displayed indicating successful subscription.
7. Navigate back to the user's dashboard.

Expected Results:

1. In Step 6, a confirmation message should appear, confirming successful subscription.
2. The subscribed tournament should now be listed in the user's dashboard or profile.

Pass Criteria:

1. The confirmation message is displayed.
2. The user is subscribed to the selected tournament.
3. The subscribed tournament is visible in the user's dashboard or profile.

Fail Criteria:

1. The confirmation message is not displayed.
2. The user encounters an error during the subscription process.
3. The subscribed tournament does not appear in the user's dashboard or profile.

Reasoning:

This test case ensures that users can easily subscribe to a tournament within the CodeKataBattle application. Successful completion of this test case demonstrates that the subscription feature is functioning correctly, allowing users to participate in tournaments, which is a key functionality of the system.

Acceptance Test Case: Update Tournament Score

Test Objective:

1. Verify that the system updates the tournament scores of participating students at the end of a battle.

Preconditions:

1. A battle within a tournament has ended.

Test Steps:

1. View the tournament leaderboard or scores.

Expected Result:

1. The scores of participating students should be updated based on their performance in the battle.

Acceptance Test Case: Add Educators to Tournament

Test Objective:

1. Confirm that the tournament leader can add other educators to a tournament at any time.

Preconditions:

1. The educator is a tournament leader.
2. The tournament is ongoing.

Test Steps:

1. Navigate to the tournament details.
2. Click on the "Add Educator" button.
3. Enter the email of the educator to add.
4. Click the "Add" button.

Expected Result:

1. The added educator should have access to the tournament and its details.

Acceptance Test Case: Close Tournament

Test Objective:

1. Verify that the tournament leader can close a hosted tournament.

Preconditions:

1. The educator is a tournament leader.
2. The tournament is ongoing.

Test Steps:

1. Navigate to the tournament details.
2. Click on the "Close Tournament" button.

3. Confirm the action.

Expected Result:

1. The tournament should be in the closing state, no new battles can be started.
2. Eventually when the battle ends it will close

Acceptance Test Case: Create Battle

Test Objective:

1. Confirm that educators can create battles within the context of a tournament.

Preconditions:

1. The educator is logged into their account.
2. A tournament is ongoing.

Test Steps:

1. Navigate to the tournament details.
2. Click on the "Create Battle" button.
3. Fill in the battle details, including name, description, and timer settings.
4. Click the "Create" button.

Expected Result:

1. A new battle should be created and listed within the tournament.

Acceptance Test Case: Student Joins Battle

Test Objective:

1. Verify that students can join a battle only if it has not started yet.

Preconditions:

1. The student is logged into their account.
2. A battle is available for joining, and it has not started.

Test Steps:

1. Navigate to the list of available battles.
2. Click on the "Join" button for a battle.
3. Confirm the action.

Expected Result:

1. The student should be successfully added to the battle participants.

Acceptance Test Case: Group Size Requirement

Test Objective:

1. Confirm that students are required to enter a group of an acceptable size before the battle starts, or they will be expelled when the battle begins.

Preconditions:

1. The student is logged into their account.
2. The student is part of a group.
3. A battle is available for joining, and it has not started.

Test Steps:

1. Attempt to join a battle with a group size that meets the requirements.
2. Verify that the student is added to the battle.

Expected Result:

1. The student should be successfully added to the battle if the group size meets the requirements.

Preconditions (Expulsion Test):

1. The student is logged into their account.
2. The student is part of a group.
3. A battle is available for joining, and it has started.

Test Steps (Expulsion Test):

1. Attempt to join a battle with a group size that does not meet the requirements.

Expected Result (Expulsion Test):

1. The student should not be allowed to join the battle

Acceptance Test Case: Battle Timer

Test Objective:

1. Verify that a battle starts and ends when the timer set upon creation expires.

Preconditions:

1. A battle has been created with a specific registration and submission deadline.

Test Steps:

1. Wait for the timer to expire.

Expected Result:

1. The battle should automatically start and end once the time is reached.

Acceptance Test Case: GitHub Repository Creation

Test Objective:

1. Confirm that the system creates a GitHub repository for a battle.

Preconditions:

1. A battle has been created.

Test Steps:

1. View the battle details.
2. Check for the presence of a GitHub repository link.

Expected Result:

1. A GitHub repository link associated with the battle should be visible.

Acceptance Test Case: Pushing Code KATA to GitHub

Test Objective:

1. Verify that the system pushes the Code KATA previously uploaded by the educator to the associated GitHub repository.

Preconditions:

1. A GitHub repository is linked to the battle.
2. The educator has uploaded Code KATA for the battle.

Test Steps:

1. Monitor the GitHub repository for code updates.

Expected Result:

1. The system should automatically push the educator's Code KATA to the GitHub repository.

Acceptance Test Case: GitHub Action Notification

Test Objective:

1. Confirm that the system is notified about new pushes by the GitHub Action set up by the group leader.

Preconditions:

1. GitHub Actions are set up for the battle's GitHub repository.
2. Code changes are pushed to the repository.

Test Steps:

1. Monitor the system for notifications or updates related to the pushed code.

Expected Result:

1. The system should receive notifications or updates indicating new pushes by the GitHub Action.

Acceptance Test Case: Pushed Code Evaluation

Test Objective:

1. Verify that the system evaluates the pushed solution and gives a temporary score after each push.

Preconditions:

1. Code changes are pushed to the GitHub repository.
2. Evaluation criteria are defined for the Code KATA.

Test Steps:

1. Monitor the system for score updates or evaluations after each push.

Expected Result:

1. The system should automatically evaluate the pushed solution and provide a temporary score.

Acceptance Test Case: Battle Ranking Display

Test Objective:

1. Confirm that the ranking is shown at the end of the battle.

Preconditions:

1. The battle has ended.
2. Evaluation and scoring have been completed.

Test Steps:

1. Access the battle's results or ranking page.

Expected Result:

1. The ranking of participants should be displayed based on their performance in the battle.

Acceptance Test Case 16: Group Leader Invites Peers (R18)

Test Objective:

1. Verify that group members can invite peers to join their group if the battle has not started yet.

Preconditions:

1. The student is logged into their account.
2. A battle is available for joining, and it is in PRE_BATTLE status.
3. At least another student is registered to the same battle.

Test Steps:

1. Navigate to the group management section.
2. Click on the "Invite Peers" button.
3. Select the peer(s) to invite.
4. Send the invitations.

Expected Result:

1. Invitations should be sent to the specified peers, allowing them to join the group.

Acceptance Test Case: Battle Score Registration

Test Objective:

1. Verify that the system registers as the score obtained in a battle the last score calculated (which corresponds to the score of the last push on the repo).

Preconditions:

1. The submission-deadline of battle has expired (battle in CONSOLIDATION status).

Test Steps:

1. Access the battle's results or scoring page.

Expected Result:

1. The system should display the last score calculated as the final score for the battle.
2. Tournament service is notified through a Kafka message of the new definitive score.

Installation Setup

Backend using docker compose:

1. Ensure to have docker installed in your system and the docker daemon running.
2. Open the terminal and go in the /implementation directory of the ckb project.
3. Run docker-compose up sonarqube
4. Open the browser
5. Go to “http://localhost:9000”
6. Access SonarQube – Default user name and password are “admin”
7. Click on the user logo on the right side of the screen
8. Go on security
9. Generate a new token – Choose as token type “user-token”
10. Copy the token an put it in the docker-compose.yaml as the value of “SONAR_TOKEN”
11. Now go to “github.com”

12. Login into your account
13. Generate a new token
14. Copy the token in the docker-compose.yaml file as the value of “GITHUB_API_TOKEN”
15. Set the value of “GITHUB_API_USERNAME” your github name
16. Run docker-compose up – This will start the backend API with all the services associated to it.

Web server

1. Install node.js.
2. Use the terminal to move to the /implementation/ckbfronted directory.
3. Run the command “npm install react-scripts”.
4. Run “npm start” – this will start the web server on port 3000.

To use the application just use a browser to connect to <web server ip>:3000.

Effort Spent

	Feraboli	Filippini	Lucca
Implementation&Testing	143	141	155

References

1. GitHub API: https://docs.github.com/en/rest?apiVersion=2022-11-28
2. SonarQube: https://docs.sonarsource.com/sonarqube/latest/
3. SonarQube API: https://next.sonarqube.com/sonarqube/web_api/api/alm_integrations
4. Spring Boot: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/
5. Kafka: https://kafka.apache.org/documentation/
6. Netflix Eureka: https://cloud.spring.io/spring-cloud-netflix/reference/html/