POLITECNICO
MILANO 1863

Implementation and Testing Document

CodeKataBattle

Authors

Name	**Surname**	**ID**
Alessandro	Saccone	11013852
Matteo	Sissa	10972783
Sara	Zappia	11016799

Source Code GitHub: Repository Link

Version 1.0 - 04/02/2024

A.Y. 2023/2024

Contents

1 Introduction

1.1 Purpose

The purpose of the Implementation and Testing Deliverable document for the ”Code Kata Battle” app is to outline the detailed plan and approach for the development and testing phases of the software in order to guarantee that the developed software meets the specified requirements and functions as intended in real-world scenarios. This document aims to provide a comprehensive guide for the implementation team, detailing implementing decisions, coding standards, and testing strategies that has been employed during the development of the application.

Building upon the requirements specified in the Requirements Analysis and Specification Document (RASD) and upon the design specified in the Design Document (DD), this document will delve into the technical aspects of how the functionalities described in the previous two documents will be translated into a working software system. It will address the intricacies of integrating with GitHub, establishing tournament and battle functionalities, handling user authentication, and ensuring the platform’s scalability, maintainability, and security.

Moreover, the document will define the testing procedures and methodologies. This includes unit testing, integration testing, and system testing.

1.2 Scope

The Implementation and Testing Deliverable for the ”Code Kata Battle” app comprehends the following key areas:

1. **Authentication and User Management:**
   - Integration with GitHub for user authentication.
   - Creation of user accounts and management of user profiles.
2. **Tournament and Battle Creation:**
   - Educators can create tournaments, defining parameters such as name, description, and registration deadline.
   - Educators can design and publish code kata battles within tournaments.
   - Specification of battle parameters, including deadlines, team size constraints, and consolidation stages.
3. **GitHub Integration:**
   - Automatic generation of dedicated GitHub repositories for battles after the registration deadline.
   - Implementation of GitHub Actions to trigger notifications to the app on code commits.
4. **Battle Participation and Scoring:**
   - Students can join battles individually or form teams within specified constraints.
   - Automated scoring of code solutions based on test cases, time, and external static analysis.

5. **Ranking and Notifications:**
    - Real-time updating of battle and tournament rankings.
    - Notification system to inform users about new battles, updates, and final results.
6. **Badges and Rewards:**
    - Definition of badges and rewards by educators based on student achievements.
    - Automatic assignment of badges upon tournament closure and result calculation.
7. **Architectural Overview:**
    - Description of the implementation of the microservices and database structure.
    - Explanation of RESTful APIs, event-driven communication, and the Model-View-Controller pattern.
8. **Testing Strategies:**
    - Detailed plans for unit testing, integration testing, system testing.
    - Identification of test scenarios and expected outcomes.

By providing a comprehensive scope, this document aims to guide the development in successfully implementing and validating the ”Code Kata Battle” app according to the specified requirements and design decisions.

2 Functions Implemented

For the implementation, the primary focus has been on developing requirements and functions that effectively express communication among all microservices. In doing so, the choice was made to emphasize Sequence Diagrams as described in the Design Document, accurately representing the integration of such communication for the most important functions of the application. More precisely, the selected use cases were:

- **Educator and Student Login:** Assigning a role to users is crucial to distinguish between creating a tournament or a battle and subscribing to them. Ensuring that all users creating and participating in tournaments possess a GitHub Account is fundamental, so the exclusive means of signing in to the Code Kata Battle web app is through an existing GitHub Account.
- **Tournament and Battle Creation:** These functions are fundamental to the application, and their absence would impact meeting the most relevant requirements and goals outlined in the Requirement and Analysis Specification Document.
- **Subscription to a Tournament or a Battle:** These functions are vital to ensuring the proper functioning of the web application, as explained before.
- **Assignment of Scores**. Notwithstanding the fact that these functions are of secondary importance for the basic functioning of the web application, we decided to implement them because they are of great relevance considering the final version of the web application.
- **End of a Tournament or a Battle:** Implementation of this function allows us to manage and implement the Notification Manager's functions and the Apache Kafka framework for asynchronous communication. It also ensures correct management of schedulers controlling parameters, such as the expiration of the submission deadline for a battle, natively offered by Spring.
- **Code Push:** The core of the entire application relies on this function, so its implementation is of maximum importance. Without it, the testing and development useful for assigning the score or for correctly communicating with GitHub APIs wouldn't be done.

The only function we decided not to implement, as can be seen in the Design Document, is the possibility for an Educator to give permission. The reasons for this decision rely on the complexity of such a function, considered not of fundamental importance in the basic operations of the application.

3 Adopted Development Frameworks

The most relevant frameworks used for the implementation for the web application are the following:

- **Spring Boot**. Spring Boot has been chosen especially due to several compelling reasons, compatible especially with microservice architecture: indeed it provides a convention-over-configuration approach, reducing the need for developers to write boilerplate code, and this simplicity accelerates the development process for microservices; with embedded servers and default configurations, then, it facilitates quick prototyping and development, accelerating the overall process; moreover it enables the creation of independently deployable and scalable services in an easy and intuitive way; furthermore Spring Boot includes embedded servers like Tomcat, Jetty, and Undertow, simplifying deployment and aligning with the lightweight nature of microservices; also centralized dependency management through starter templates ensures consistency across microservices; finally, Spring Boot Actuator facilitates built-in monitoring and management, contributing to the overall maintainability of microservices.

  The programming language chosen has been Java. This decision stems from several considerations. Firstly, the development team possesses a strong proficiency in Java, making it a natural and efficient choice. While Kotlin was considered, the team's extensive experience with Java provided a familiarity that would expedite development and troubleshooting. Secondly, Java's maturity, extensive ecosystem, and wide community support were crucial factors influencing the decision. Considering furthermore that the language's robustness in building scalable and reliable systems align well with the requirements of microservices architecture, Java emerged as the preferred language for the Code Kata Battle development.

- **Apache Kafka**. Kafka plays a vital role in enabling the asynchronous exchange of messages within the microservices architecture. Specifically, the Notification Microservice heavily relies on Apache Kafka for its effective operation. Indeed, Kafka's significance for the Notification Microservice includes supporting event-driven communication. Instead of direct, synchronous interactions, Kafka establishes a communication channel where producers publish events, and consumers asynchronously process these events. Among the advantages of this approach we include the ensuring of scalability and of load distribution. This allows the microservice to handle varying loads of notification events without compromising performance.

- **Mockito**. Mockito is a crucial framework used for testing within the development process. It facilitates the creation of mock objects, enabling developers to isolate and test components in isolation. In the context of Spring Boot and microservices, Mockito proves invaluable for writing unit tests that verify interactions between different components. By creating mock implementations of dependencies, developers can ensure that each microservice behaves as expected, even before integration testing. Mockito's capabilities for behavior verification, stubbing, and easy mock creation contribute to the overall quality and reliability of the microservices implemented with Spring Boot and Kafka.

- **JGit**. JGit is an open-source, Java library that implements Git version control. Developed by Eclipse, it offers a lightweight, platform-independent solution for Git integration in Java applications. JGit's advantages are related to its pure Java nature, making it easily embeddable and ensuring compatibility across platforms. Its simplicity and intuitive API facilitate straightforward integration, providing developers with an efficient and accessible tool for implementing Git functionality in their Java projects. JGit has been employed in the development of CKB in order to give support to the Github Integration Microservice to communicate in an easy way with GitHub.

- **WireMock**. WireMock is a Java library for HTTP-based service mocking. It's developed to simplify the testing of API-driven applications, and allows developers to simulate HTTP responses. With its lightweight design and pure Java implementation, WireMock provides platform independence, making it easily employable into Java projects. WireMock has been used in the CKB platform to test some functionalities of the microservices that needed to communicate with external API through HTTP requests. In particular, the GitHub Integration Microservice has been tested using the WireMock framework, because it was necessary to provide a confined and isolated environment in order for the test to be consistent and reliable.
- **SonarQube server and Sonar Scanner**. These SonarQube products provide a convenient way of running static analysis on source code stored locally on the system. The SonarQube server gathers the results of the analysis carried out by the Sonar Scanner and offers the possibility of retrieving that data through simple API calls.

All the APIs used for the implementation have already been discussed within the Design Document.

4 Structure of Code

In order to facilitate the development of different microservices, each logically operating on separate machines, we chose to implement each of them as a separate module inside the same repository. This approach allowed us to assign different ports, set distinct settings, and connect to the shared database for each microservice independently. Specifically, the ports chosen are all on localhost and distributed as follows:

- Battle Microservice on port 8083.
- Tournament Microservice on port 8085.
- User Microservice on port 8086.
- Gateway Microservice on port 8087.
- Notification Microservice on port 8088.
- Score Computation Microservice on port 8089.
- GitHub Integration Microservice on port 8090.

The reason why port 8084 was not chosen is that it is occupied by default on some hardware configurations. Additionally, Apache Kafka runs on port 9092 (but it depends on the configuration of each computer), and the SonarQube server runs on port 9000.

Each microservice has a distinct *pom* file and could ideally have a different version of Spring or other included dependencies. While these files share a similar structure, based on specific needs, each may contain more or fewer dependencies.

The majority of microservices follow a standard architecture: the

`main/java/org/example/name_of_microservice`

folder contains four different packages, specifically

- `Controller` package, containing all the controllers, ensuring the correct handling of GET and POST requests in our application.
- `Model` package, containing all the models mapping to the tables in the database.
- `Repository` package, integrating JPA to make requests directly to the database.
- `Service` package, containing all the business logic invoked by the controller while handling its requests.

The only microservices that deviates from this standard architecture are the Gateway Microservice, the GitHub Integration Microservice and the Score Computation Microservice.

The Gateway Microservice is particularly crucial; it contains all the templates for displaying the views of the MVC model. From these templates, all the actions work in the same way:

- A POST or GET call is made from a button or link; it is up to the Gateway Manager to handle this correctly.

- Depending on the requested action, the controller will redirect the request to the correct microservice, with the correct request parameters always passed by the front-end, through a RESTful request. All business logic then relies on the invoked microservice.
- The invoked microservice will perform its action and return one or more values useful for the GET mapping, or a response useful for understanding the state of the request for the POST one.

The GitHub Integration Microservice handles all communications with GitHub (apart from the authentication of users in the Gateway) and it does that in two ways:

- Through API calls: this method is adopted to create a new empty repository for a battle, and to download a zip file containing the entire content of a student's repository when a new commit is pushed.
- Through git commands: this method is applied when the repository for a new battle is created. Initially, the GitHub Integration Microservice asks GitHub to create a new empty repository with the name of the battle through API calls (as said in the previous point). Then git commands are launched on the local repository that stores the content for a new battle in order to push everything to the remote repository.

This is why inside the code of the GitHub Integration Microservice it is possible to find the Download package, responsible for interacting with GitHub when the download of a student's repository is needed, the Git repository, necessary to run git commands when the repository of the battle is created, and the UnzipUtil package that is a helper package containing static methods to work on files and manipulate them on the local file system. The data flow for this microservice is the following:

- When a new battle starts (registration deadline passes), the Battle Microservice calls the /publishRepo endpoint exposed by the GitHub Controller. The GitHub Controller sends a HTTP request to GitHub to create a new empty repository for the battle. Then, it contacts the Battle Microservice for extracting the test cases and build automation scripts of the battle and places all of that inside a folder under "battles" named with the name of the battle. Then the GitOp class is used to run git commands to push the entire content of the repository to the remote repository created on GitHub.
- When a student pushes some new commits on a branch, the GitHub Actions workflow is set to hit the /newSubmission endpoint of the GitHub integration controller. As a first thing, the controller leverages the GitHubCodeDownloader to pull down all the repository (that is packaged into a zip file) and places it under "code_download". Then the UnzipUtil is used to extract the content. At this point, the control is passed on to the Score Computation Microservice to compute the score.

The Score Computation Microservice takes care of calculating the score to assign to a students new solution. This microservice is "activated" by the Github Integration Microservice thanks to the endpoint /computeScore. The Score Computation Microservice takes for granted that the GitHub integration microservice has already downloaded the student's folder under the "code_download" folder and these three important points are executed:

- The build script is run on the "CODE" folder inside the downloaded repository (all repositories must have the same structure, see installation instruction for more info). The build script is also responsible for running the test cases and returning a result.

- The time passed from the beginning of the battle is calculated.
- Sonar Scanner is started on the ”CODE” folder in order to provide static analyses, that are sent directly to the Sonar Server listening on port 9000. Then, the microservice uses simple API calls to retrieves the results of these analyses and computes the final score.

At the end, the Score Computation Microservices asks the Battle Microservice to update the score with the results and deletes the folder that was downloaded from github under ”code_download”

5 Testing Details

In our testing phase, we employed JUnit 5 and Mockito to ensure the reliability and functionality of our microservices. JUnit 5, a robust testing framework for Java, and Mockito, a powerful mocking framework, were utilized to thoroughly test the user, battle, and tournament microservices. Our approach involved initially testing service layer methods by mocking the repository and subsequently extending the scope to controller layer methods, effectively mocking the services. This layered testing strategy ensured systematic verification of microservices functionality in isolation.

The testing classes can be found in the development environment under the IMPLEMENTATION folder, there is a dedicated directory for each microservice and under the src/test/main, the test classes can be found.

6 Installation Instruction

In order to make the entire server work correctly, many details are important.

6.1 Git

The user should have git installed and should be authenticated with a GitHub account.

6.2 Apache Kafka

Apache Kafka has to be installed on the local system and before starting CKB it is mandatory to run the zookeeper and the kafka server. In order to avoid any problem with the topics, it is also recommended to add the following line at the end of the configuration file for the server, that it is possible to find under config/server.properties: auto.create.topics.enable=true.

6.3 SonarQube server

SonarQube server Community edition has to be installed on the local system and run before starting CKB. It is very important to follow carefully the installation instructions that are provided on the official web page. More specifically, as of now (February 2024), SonarQube server cannot be run with a version of Java newer than 17. It is also very important to change the default configurations of the Sonarqube server that can be found in the zip folder downloaded from the official website under config/sonar.properties. More specifically, there is an issue with the fact that the default configuration of the SonarQube server runs an embedded database (h2) on port 9092 of the local system. But this is in conflict with the default configuration of Apache Kafka, whose server also runs by default on that port. In order not to have any problem it suffices to change the default configurations for the database connections for the SonarQube server and redirect them to a local database of any kind (for instance Postgres). The interested properties to change are:

- sonar.jdbc.username=sonarqube_user
- sonar.jdbc.password=sonarqube_password
- sonar.jdbc.url=jdbc:postgresql://localhost:5432/...

The official webside provides a very well explained roadmap for the installation and how to change these properties effectively. Once this is done, it is possible to verify the installation with the script files in the directory where the SonarQube server is downloaded (on Windows run sonarstart). At this point the SonarQube server should be started and listening on port 9000 of the local machine. The first time connecting to the server at http://localhost:9000, the username and password are both admin. In order for CKB to collaborate with this server, it is fundamental to pass to the CKB instance a personal access token generated in the SonarQube server, so that CKB has access to the analyses that are stored on the server. From the home page of the server, in the Account section, Security subsection it is possible to generate a new **User** token. This will be one of the parameters to pass to the script to launch CKB. Notice: the SonarQube server has to stay up and running for the entire functioning of CKB.

6.4 Sonar Scanner

CKB has to run static analysis on the code downloaded from GitHub, and to do that, it leverages a Sonar Scanner. It is very easy to download at: https://docs.sonarsource.com/sonarqube/latest/analyzing-source-code/scanners/sonarscanner/. Once the download is complete, it is very important to make sure to place the sonar-scanner executable under the PATH variable of the system, otherwise CKB won't find it at runtime.

6.5 Database

Another important part of the CKB application is the database integration. CKB needs to interact with the underlying database to store persistent information, so it has to know the coordinates of the database to contact and have the necessary permissions. For CKB, Postgres has been chosen among the various providers. In order for the database interactions to work seamlessly, it is mandatory to setup a local database with name ckb_db, username (with permissions) ”postgres” and password ”Dolphin!”. The final url of the database should result as jdbc:postgresql://localhost:5432/ckb_db.

6.6 Folder Structure

Some constraints are imposed on the way educators should upload the build automation scripts and test cases:

- CKB manages to handle and run analysis only on Maven projects. This means that, when an educator wants to create a new battle, s/he should create a folder called ”CODE” and inside that folder initiate a new Maven project (respecting the conventions for the directory structure). Inside the test folder, the test classes can be designed by the educator. The src/ folder will be filled in with the solutions by the students. This ”CODE” folder can then be compressed into a ZIP format and uploaded on CKB. Notice: No other directory hierarchy or naming conventions are admitted.

- As for the build scripts, the educator should create a ”SCRIPTS” folder, with two sub-folders ”WINDOWS” and ”UNIX-LIKE”. Then, inside the ”WINDOWS” folder, the file ”build.bat” has to be provided for building the project and inside the ”UNIX-LIKE” folder the ”build.sh” can be supplied for Unix-like systems. Notice: the build automation scripts should handle the building process and also start the tests. Since CKB only deals with maven projects, a simple way to do that is using a script running ”mvn test”.

6.7 Using Ngrok

In order to fully test the behavior of the application, an automated GitHub workflow has to be written by the student and placed under the ”.github/workflows” folder on the forked GitHub repository (with a customary name). This workflow should contact the CKB server running on the local machine in some way. Since CKB doesn't have a public IP address, it is possible to use services like Ngrok to make it possible for a machine external to the local network to send an HTTP request to the server where CKB is run. The installation process is very simple on the official website. The first time ngrok is used, it is necessary to create a new account and generate a token on the platform. That token is used on the local machine where CKB runs to authenticate the user and ask for a publicly visible endpoint that an external machine can contact. Once the authentication is done, it is necessary to run on the terminal ”ngrok http 8090”, which creates a public endpoint, that will be redirected to http://localhost:8090, where the GitHub integration microservice is running. This endpoint has to stay active for the entire period in which CKB is up and running. A possible github workflow is provided in the following.

```yaml
name: Send HTTP Request on Push

on:
  push:
    branches:
      - main

jobs:
  send-http-request:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Send HTTP Request
      run: |
        curl -X GET \
        -H "User-Agent: Custom" \
        "https://be06-176-206-221-239.ngrok-free.app/newSubmission/
            sissamatteo29/testBattle?tour=testTournament&battle=testBattle&
            teamName=testTeam"
```

Listing 1: Your YAML code

Notice that:

- The public endpoint in the final HTTP request has to be customized with the one displayed by ”ngrok http 8090”.
- The endpoint to correctly contact the GitHub Integration Microservice has to be constructed in this way:

  /newSubmission/{github_username}/{github_repository}?tour=

  tournament_name}&battle={battle_name}&teamName={team_name}

  All fields have to be filled in based on the specific situation of the student.

6.8 Run the application

Once all external elements are up and running, executing the CKB platform can be done with the scripts provided under the bin/ folder. These scripts require 3 parameters to be passed in:

- The absolute path to the folder that hosts the project on the local file system

- The GitHub access token, which is statically assigned to be ghp_d82tMinTc2CpxPkUwLc6MCI7iAttB42ZA (it is a token for accessing the ckb github account)
- The sonarQube token, which has been described earlier for accessing the SonarQube server from CKB.

7 Effort spent

Studente	Hours spent
Alessandro	100
Matteo	100
Sara	100

Table 1: Time spent on the Implementation and Testing

8 References

- From Youtube: Rest API and Microservices — Microservices Unit Testing using Spring Boot + JUnit5 + Mockito
- Spring Framework Documentation
- Java Documentation
- JavaScript Documentation
- CSS and HTML Documentation