Accademic Year: 2023-2024

POLITECNICO
MILANO 1863

Implementation and Testing Document

CodeKataBattle

SOFTWARE ENGINEERING 2

PROFESSOR DI NITTO ELISABETTA

CONTI ALESSANDRO CODICE PERSONA: 10710583 MATRICOLA: 252665

DI PAOLA ANTONIO CODICE PERSONA: 10717589 MATRICOLA: 956038

Contents

1. INTRODUCTION

1.1. Overview

This Document is intended to provide a comprehensive overview of the CodeKataBattle project. This documentation will be a guide for the reader, enabling them to understand the decisions regarding the design of the application. In particular, the architectural structure and design choices will be detailed, allowing developers, stakeholders, and other team members to gain a clear understanding of how all the features previously outlined informally in the RASD will be implemented.

1.2. Purpose

The purpose of our project, named CodeKataBattle (abbreviated as CKB), is to provide a digital platform that enables students to develop and refine their skills in software development through a collaborative experience. The CKB platform allows students to practice in a programming language of their choice. The exercises offered on this platform are created by educators and require students to complete the code or parts of it, adding the missing components they deem appropriate. The code is then executed and subjected to specific tests created by educators to verify its correctness.

Exercises are organized into tournaments by educators, within which 'battles' are held to evaluate the performance of the students. This process culminates in the creation of performance rankings among all students involved in the battles and tournaments. In this way, CKB encourages friendly competition and the growth of students' skills in the field of software development.

1.3. Scope

The CKB application, as defined in the RASD document, is a web application focused to enhancing software development skills for users enrolled in the Students category. Educators, who have in-depth expertise in the topic, will manage tournaments and battles within the platform to facilitate the improvement of students' skills. Through detailed rankings of battles and tournaments, students can assess their level and monitor progress in enhancing their skills.

1.4. Definitions, Acronyms, Abbreviations

1.4.1. Definitions

- *Commit*: commits are the core building block units of a Git project timeline. Commits can be thought of as snapshots or milestones along the timeline of a Git project. Commits are created with the `git commit` command to capture the state of a project at that point in time.
- *Fork*: a fork is a new repository that shares code and visibility settings with the original “upstream” repository.
- *Code kata*: code kata contains the description of a battle and the software project on which the student will have to work, including also test cases and build automation scripts
- *Test Case*: a test case is a singular set of actions or instructions that the Educator wants to perform to verify a specific aspect of the project pushed by the students. If the test fails, the result might be a software defect that students might have not found.
- *Upload*: upload in which the educator sends to the platform database the code kata for a specific battle.
- *Repository*: a Git repository is the .git/folder inside a project. This repository tracks all changes made to files in your project, building a history over time. Meaning, if you delete the .git/folder, then you delete your project’s history.
- *Branch*: in Git, a branch is a new/separate version of the main repository. Branches allows users to work on different parts of a project without impacting the main branch. That main branch is the one seen as the default one.
- *Push*: the `git push` command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo.
- *Pull*: the git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration work flows.
- *Cross-Site Scripting*: cross-site scripting is a web application security vulnerability that allows an external attacker to add malicious scripts into web pages viewed by other users. These scripts can be executed in the context of the victim's browser, allowing the attacker to steal sensitive information, such as user sessions, or perform malicious actions on behalf of the victim without their consent.
- *Frame*: a frame is a region within a web page that can contain a separate HTML document. The use of frames allows multiple HTML documents to be displayed in the same browser window, allowing a page to be divided into independent sections.
- *Inline Frame*: an inline frame is an HTML element that allows an HTML document to be embedded within another HTML document. Inline frames are often used to embed content from external sources, such as videos, maps, or third-party Web pages, within a page.

1.4.2. Acronyms

- CKB: CodeKataBattle.
- CK: CodeKata, Code Kata.
- GH: GitHub.
- GHA: GitHub Action.
- DAO: Data Access Object.
- RASD: Requirements Analysis and Specification Document.
- DD: Design Document.
- API: Application Programming Interface.
- SMTP: Simple Mail Transfer Protocol.
- MIME: Multipurpose Internet Mail Extensions.
- XSS: Cross-Site Scripting.
- JVM: Java Virtual Machine.

1.4.3. Abbreviations

- [APIn]: the n-th API.
- [FWn]: the n-th framework.
- Repo: Git repository.
- iFrame: Inline Frame.

1.5. Revision history

- Version 1 (2024/02/04).

1.6. Reference documents

This document is strictly based on:

- The specification of the ITD and ATD and assignment of the Software Engineering 2 course, held by professor Matteo Rossi, Elisabetta Di Nitto and Matteo Camilli at the Politecnico di Milano, A.Y 2022/2023.
- Slides of Software Engineering 2 course on WeBeep.
- Official link of CodeKata: http://codekata.com/.

1.7. Documents Structure

The rest of the document is organized as follows:

- *Implemented Requirement* (Chapter 2) contains a detailed description of how the requirements and functionality outlined previously in the RASD and DD documents have been implemented.
- *Source Code Structure* (Chapter 3) contains the structure of the implemented source code divided into packages.
- *Testing* (Chapter 4) explains how the tests assessing the fulfilment and correctness of the specified requirements were conducted.
- *Installation Instruction* (Chapter 5) explains how to properly run the implemented application.

2. IMPLEMENTED REQUIREMENT

2.1. Overview

In this chapter, we will explain the implementation of the requirements and functionalities delineated in the previous documents, RASD and DD. We will describe how these requirements and functionalities have been translated into concrete solutions by adopting specific criteria and approaches.

2.2. Requirement

This section illustrates the requirements delineated in the RASD document and their implementation. The functional requirements were all implemented excluding requirement [R21]. which was not required in the implementation specification.

As stated in the DD document, a relational database was used to store all the information necessary for the proper functioning of the application, with the exception of the project-specific data to be implemented during a battle by users enrolled as Users. To establish secure connections with the database, the *mysql-connector* was employed, while read and write operations were handled through *java.sql.PreparedStatement*. These objects, because they represent precompiled SQL statements, helped mitigate the risks of Query Injection type attacks, since the parameters are handled separately from the query, thus reducing the possibility of malicious query execution.

To store the specification of a battle and, if requested by Educator, also a code base for students to work on, GitHub was chosen. Specifically, a new repository was created on GitHub for each battle. When an educator creates a battle, the application saves the zipper folder containing the specification and/or code to be completed on GitHub. At the same time, all other information, such as the name of the battle, the deadlines, and the link to the repository, is recorded on the relational database. At the start of a battle, when the registration deadline has expired, the application automatically creates a repository for each registered team, retrieving all the information previously entered by the educator from the battle repository. Interaction with GitHub was made possible through the use of the *org.eclipse.jgit* API, which allows manipulating repositories on GitHub by implementing Git objects and performing operations corresponding to those performed with Git software from the command line.

Finally, to meet the requirements of automatic email sending, an email API was integrated using *jakarta.mail*. The choice fell on Gmail and the SMTP protocol of smtp.gmail.com to ensure automatic email sending in accordance with the required specifications.

2.3. Function

This section delineates the additional functionality described in the DD document and the related implementation process.

In line with the DD document specifications, the application was built as a web application structured in three distinct layers. The first tier, the presentation layer, was developed through HTML and JavaScript pages displayed when the user interfaces to the application through their browser. The second tier, the business logic layer, was implemented as a Java server, which makes decisions and provides responses in response to requests from the presentation layer. The third tier, the data layer, consists of a relational database, and GitHub is used to store specific data, as previously described. The adoption of this structure naturally led to the use of the MVC pattern. Other patterns implemented include the Observer Pattern, used for automatic email notifications at the creation, closure of tournaments and at the start and conclusion of battles, as well as the Command Pattern, which handles the separation of actions and requests through dedicated classes.

To ensure the security of the application, several controls have been implemented to prevent malicious attacks. The application requires a secure connection via HTTPS, thus encrypting the messages exchanged. Resources displayed in the browser must come exclusively from the server, be interpreted only with their specific MIME type, and cannot be included in external frames or iFrames. In addition, the application actively handles XSS attacks.

In the context of security, the only sensitive data currently encrypted using Triple DES is the application's login password, while other data does not currently undergo encryption. To facilitate running the application on different devices, access keys, such as those for the notification email, GitHub account, and Triple DES encryption key, are not stored as environment variables.

At the current stage, the application was not designed as a distributed system, but was intended to be easily scalable in the future. The use of MySQL as the database was chosen for its suitability for the current computational load, avoiding the adoption of Amazon RDS as initially proposed in the DD.

As additional functions, the ability to change personal information, view notifications received from the server, and receive a confirmation email for account registration will be provided, but these are still being implemented.

2.4. Development Framework

2.4.1. Programming language adopted

To implement our application, we chose to use Java as a programming language motivated by several factors that we believe are crucial to the success of the project.

First, Java is an easily portable programming language-in fact, a Java web application can run on a wide range of devices that support the Java Virtual Machine (JVM). This feature gives us flexibility and a wide range of compatibility.

An additional advantage that steered us toward choosing Java is the robust built-in security offered by the language. The JVM's secure memory management and security system provide a solid infrastructure for dealing with security challenges, which is of paramount importance in the context of Web applications.

Finally, the last determining factor was the widespread industry-wide adoption of Java; in fact, the many resources available greatly simplify troubleshooting.

We prefer to put on the back burner any considerations of Java's syntactic complexity and the memory overhead associated with running the JVM. We believe that the benefits associated with portability, security, and community support far outweigh these issues, providing a solid foundation for long-term application development and scalability.

2.4.2. API adopted

[API1]. *org.eclipse.jgit*: it is a Java library that provides an API for interacting with Git repositories.

[API2]. *jakarta.mail*: it is a Java API that provides functionality for managing and sending email.

[API3]. *javax.servlet*: it is a Java package and API that provides the classes and interfaces needed to develop Java servlet.

2.4.3. Framework adopted

[FW1]. *junit*: it is a unit test framework for the Java programming language that provides an environment and set of methods for writing and executing unit tests.

[FW2]. *org.mockito*: it is a java based mocking framework, used in conjunction with other testing frameworks such as JUnit. A mock object returns a dummy data and avoids external dependencies. It simplifies the development of tests by mocking external dependencies and apply the mocks into the code under test.

3. SOURCE CODE STRUCTURE

This chapter will explain the structure of the source code.

3.1. Bean

src/main/java/it/polimi/SE2/CK/bean/Battle.java

src/main/java/it/polimi/SE2/CK/bean/Ranking.java

src/main/java/it/polimi/SE2/CK/bean/SessionUser.java

src/main/java/it/polimi/SE2/CK/bean/Team.java

src/main/java/it/polimi/SE2/CK/bean/Tournament.java

src/main/java/it/polimi/SE2/CK/bean/User.java

3.2. DAO

src/main/java/it/polimi/SE2/CK/DAO/BattleDAO.java

src/main/java/it/polimi/SE2/CK/DAO/TeamDAO.java

src/main/java/it/polimi/SE2/CK/DAO/TournamentDAO.java

src/main/java/it/polimi/SE2/CK/DAO/UserDAO.java

3.3. Servlet

src/main/java/it/polimi/SE2/CK/servlet/AddCollaborator.java
src/main/java/it/polimi/SE2/CK/servlet/CheckBattleRanking.java
src/main/java/it/polimi/SE2/CK/servlet/CheckTournamentRanking.java
src/main/java/it/polimi/SE2/CK/servlet/CloseTournament.java
src/main/java/it/polimi/SE2/CK/servlet/CreateBattle.java
src/main/java/it/polimi/SE2/CK/servlet/CreateTournament.java
src/main/java/it/polimi/SE2/CK/servlet/GetUser.java
src/main/java/it/polimi/SE2/CK/servlet/JoinBattleAlone.java
src/main/java/it/polimi/SE2/CK/servlet/JoinBattleAsTeam.java
src/main/java/it/polimi/SE2/CK/servlet/JoinTeam.java
src/main/java/it/polimi/SE2/CK/servlet/JoinTournament.java
src/main/java/it/polimi/SE2/CK/servlet/LoginManager.java
src/main/java/it/polimi/SE2/CK/servlet/LogoutManager.java
src/main/java/it/polimi/SE2/CK/servlet/ModifyGrade.java
src/main/java/it/polimi/SE2/CK/servlet/SearchTournament.java
src/main/java/it/polimi/SE2/CK/servlet/ShowAddCollaborator.java
src/main/java/it/polimi/SE2/CK/servlet/ShowBattleInfo.java
src/main/java/it/polimi/SE2/CK/servlet/ShowBattles.java
src/main/java/it/polimi/SE2/CK/servlet/ShowJoinTournament.java
src/main/java/it/polimi/SE2/CK/servlet/ShowTeam.java
src/main/java/it/polimi/SE2/CK/servlet/ShowTeamInBattle.java
src/main/java/it/polimi/SE2/CK/servlet/ShowTournament.java
src/main/java/it/polimi/SE2/CK/servlet/ShowTournamentInfo.java
src/main/java/it/polimi/SE2/CK/servlet/ShowUserTeam.java
src/main/java/it/polimi/SE2/CK/servlet/SignInManager.java

3.4. Utils

src/main/java/it/polimi/SE2/CK/utils/EmailManager.java
src/main/java/it/polimi/SE2/CK/utils/Evaluation.java
src/main/java/it/polimi/SE2/CK/utils/GitHubManager.java
src/main/java/it/polimi/SE2/CK/utils/ReceiveFromGitHubServlet.java

3.4.1. Enumeration

src/main/java/it/polimi/SE2/CK/utils/enumeration/TeamState.java

src/main/java/it/polimi/SE2/CK/utils/enumeration/TeamStudentState.java

src/main/java/it/polimi/SE2/CK/utils/enumeration/TournamentState.java

src/main/java/it/polimi/SE2/CK/utils/enumeration/UserRole.java

3.4.2. Folder

src/main/java/it/polimi/SE2/CK/utils/folder/FolderManager.java

src/main/java/it/polimi/SE2/CK/utils/folder/ZipFolderManager.java

3.4.3. Starting

src/main/java/it/polimi/SE2/CK/utils/starting/StartBattle.java

src/main/java/it/polimi/SE2/CK/utils/starting/StartTournament.java

4. TESTING

During the implementation of the application, it was decided to adopt a rigorous testing practice so as to promptly detect errors made during implementation and correct them in a timely manner. Testing was performed whenever a new feature was finished, before integrating it with the working system, so as to ensure its reliability and system functionality even after it was added.

It was decided to automate the testing process, and to do this, the JUnit framework was chosen and used. This decision allowed us to efficiently create test cases, automatically including assertions to verify that the calculated results matched expectations.

Along with JUnit we used another framework, Mockito, which specializes in the creation and management of mock objects. These objects allowed us to isolate specific parts of the code, enabling us to verify, independently of the actual dependencies within the system, the behaviours of individual components. Using this framework thus made it easier for us to simulate the behaviour of classes that have dependencies.

The joint use of these two frameworks, JUnit and Mockito, allowed us to test anomalous behaviours with respect to normal expectations, enabling us to evaluate the behaviour of the system when it receives unexpected stresses. For example, through the use of Mock, it was possible to simulate scenarios in which the server received requests that did not conform to expectations, thus evaluating the system's reactive behaviour under such circumstances.

5. INSTALLATION INSTRUCTION

The project was implemented as a Java web application, requiring a web server and SQL database to run. To implement it, Apache Tomcat was used as the web server, IntelliJ as the development environment and MySQL Workbench as the database.

The following steps are recommended for proper execution: install IntelliJ and import the project, then configure it properly. This includes adding the project as a Maven module and configuring Tomcat as a web server. Next, you need to import the provided database into the repository using MySQL Workbench.

Also, to import the required modules from IntelliJ: File -> Project Structure -> Modules -> + -> Import Module -> select the "Code" folder in the project -> follow the instructions until the "Create" button is displayed -> click "Create" -> click "Apply" and "OK". After that, import the project as a Maven project by right-clicking the "pom.xml" file and selecting "Add as Maven Project."

Configure Tomcat as a web server in IntelliJ by going to Edit Configuration -> + -> Tomcat Server Local -> configure Application Server -> Deployment -> + -> Artifact -> Code: war exploded. Now, IntelliJ is properly configured to run the Java web application.

Next, install MySQL Workbench and create a new connection to the database. After testing the connection, import the database by following the instructions: click Open a SQL Script File in a new query tab -> execute. Now the database is ready to be used.

Before actually running the project, some changes need to be made. In the FoldeManager.java file, add the directory where the files to be uploaded to GitHub will be saved. Also, change the OS path and enter the password to access the SQL database in web.xml and on TestUtils.java.

Now the project is fully configured and ready to run.

If you want to run tests and need to add the password to access the database in the TestUtils class as well.

6. EFFORT SPENT

In the following tables we will summarize the effort spent by each member of the team on the ITD Document

- Conti Alessandro

Chapter	*Effort (in hours)*
1	2
2	3
3	1
4	2
5	2

- Di Paola Antonio

Chapter	*Effort (in hours)*
1	1
2	1
3	1
4	1
5	1

7. REFERENCES

- IDE was used to implement the system: *IntelliJ*
- To implement the database was used: *MySQL Workbench*
- As a web server was used: *Apache Tomcat*