POLITECNICO DI MILANO

Software Engineering 2

CodeKataBattle - Improve your programming skills online

Implementation & Testing

Document

Authors:

Nicolò Giallongo - 10764261

Giovanni Orciuolo - 10994077

Giuseppe Vitello - 10766482

Table of Contents

1 - Introduction

1.1 - Purpose

The purpose of this document is to present the initial Minimum Viable Product (MVP) of the CodeKataBattle application. First, the scope of the prototype implementation is discussed, then installation instructions are provided to allow testing on a local machine.
Furthermore, the adopted development and testing frameworks are described, along with an in-depth explainer of the structure of the source code, and how the design approach of this prototype differs from the official Design Document.

1.2 - Scope

For this MVP, only a small subset of functions are implemented and tested (more details on the list of requirements are available in section 1.7).
Crucially, the stability and quality of the codebase is preferred, rather than an implementation of a large quantity of features. This initial prototype should only serve as a stepping stone, facilitating further developments to reach the final desired state of the product (as described in the RASD and DD) while having a viable product to present to clients and potential stakeholders interested in the application.
Nice to have features such as: managing friends and notifications, making and joining teams in battles, are disregarded for the moment. The focus and scope of the MVP is to have a functioning system where educators are capable of hosting a simple tournament, students can join it and submit entries in the various available battles, competing for the highest score in the tournament leaderboard. We have collectively decided that this can be considered the bare minimum of a functioning CodeKataBattle system.

1.3 - Installation Instructions

In order to install the application and run it in a local environment, the following steps are required (instructions are also available in the source code directly inside the README files):

- If you have the zip containing the source code, extract it in a folder of your liking. Otherwise, clone the repository containing the source code with the command: `git clone https://github.com/codekatabattle-polimi/OrciuoloVitelloGiallongo.git`. The source code is included inside the directory “IT”. The backend source code is inside the directory named “codekatabattle”.

The frontend source code is inside the directory named "codekatabattle-frontend".

- Install version 17 or 21 of the JDK (Java Development Kit, downloadable here).
- Install PostgreSQL on your machine or run it through the Docker image.
- On PostgreSQL, run the following command to create the database: `CREATE DATABASE codekatabattle;`
- To install and run the backend, use Maven to install dependencies and execute scripts to run the Spring Boot application:
  - `./mvnw install`
  - `./mvnw spring-boot:run`
- Alternatively, it is possible to use an IDE like IntelliJ IDEA.
- The backend will be available on port 8000. Documentation will be served at path `/docs`.
- To install and run the frontend, it is needed to have Node.js (version 20.11.0 LTS downloadable here: https://nodejs.org/en) installed. Then, run the following commands:
  - `npm install`
  - `npm run dev`
- The frontend will be available on port 5173.

1.4 - Revision History

Version	Date	Changelog
1.0	29/01/2024	First version of the document
1.1	30/01/2024	Add Section 1.8, adjust implemented features
1.2	01/02/2024	Add explanation of Testcontainers in Section 3.1

1.5 - Reference Documents

- The specification document Assignment IT AY 2023-2024.pdf
- Clean Code: A Handbook of Agile Software Craftsmanship - Robert C. Martin
- Hands-On RESTful API Design Patterns and Best Practices - Harihara Subramanian and Pethuru Raj
- The API-First Transformation - Kin Lane

1.6 - Document Structure

This document is divided in 3 main parts:

1. **Introduction**: introduces the scope of the MVP and provides installation instructions for a local machine.

2. **Adopted Development Frameworks**: Lists the chosen frameworks and methodologies for both the backend and the frontend, along with a rationale for each choice. Furthermore, the codebase structure is described.
3. **Adopted Testing Frameworks**: List the chosen testing methodologies for both the backend and the frontend.
4. **Effort Spent**: contains information about the number of hours each group member has worked on this document.
5. **References**: contains information about the resources used to redact this document.

1.7 - Implemented Requirements

The following is a list of implemented requirements in this first prototype. Please refer to the RASD for the list of requirements and the full description for each code:

- SR3, SR4, SR7 *(only through website)*, SR9, SR11
- ER1, ER2, ER7, ER8, ER9, ER10, ER12, ER15, ER17, ER20, ER22
- OR1, OR2, OR8, OR9, OR10, OR11, OR14, OR16, OR17

1.8 - Other Design Decisions

One of the implemented features is automatic execution of the provided tests for a given battle based on submitted commits (called “entries”). This strictly implies RCE (Remote Code Execution) which is a common security vulnerability. This situation is unavoidable given the requirement.

However, it is possible to reduce the attack surface by adopting a technique called “sandboxing”: the custom code submitted by participants is executed in a “sandboxed” environment (which is an execution environment completely separated and closed off from the main server where the backend is running).

This can be concretely implemented by automatically provisioning Docker containers when the code needs to run, and then shutting them down automatically and gracefully. This sandbox environment does not have write privileges on itself (it is only possible to write to stdout stream), so the possible attack surface is greatly reduced. Even if a malicious RCE is performed, the environment makes it more difficult to “breach” the container (doing so would require the exploit of a vulnerability inside Docker itself). This has been deemed secure enough for the scope and purpose of the project (at least in MVP stage).

2 - Adopted Development Frameworks

2.1 - Backend Framework

The framework of choice for the backend part of the application is **Spring Boot** (https://spring.io/). Boot makes the process of creating, running and managing dependencies of a Spring-based project simpler and smoother, which is ideal for quick prototyping and iterating. The language of choice is **Java**. Java and Spring have been chosen mainly because they are both “battle-tested” (which is to say that they have a lot of history and the backing of major organizations). Java is a portable language which can run everywhere a JVM can run. This makes it easy to containerize and distribute the application through Docker images (and, thus, Kubernetes on a real production deployment).
Spring offers a lot of tooling to easily make a RESTful API while following SOLID principles.

The transport layer of choice for the API is JSON. JSON payloads are easy to parse both for humans and for machines, which is a huge advantage in terms of developer experience (DX) and makes working with the API as an external consumer easier as a result. The trade-off is that JSON is schema-less: this problem is partially solved by adopting OpenAPI.

The design of the entire application is “API-first”, which means that the whole application is usable only through the API layer, without requiring a frontend. This makes integrations with external systems smoother, while also providing a myriad of other advantages. The frontend is just a thin layer which does not have a lot of logic, and exists mainly as a presentation layer to allow non-technical users to enjoy the system’s various functionalities without having to invoke the APIs themselves. The frontend can also aggregate data coming from the RESTful APIs in various ways, but must not implement anything related to business logic. The business logic must reside entirely within the backend: separation of concerns is the key.

More details about the API-first approach can be found here. In short, the backend stipulates an API contract through the usage of OpenAPI. The OpenAPI document is not manually written: it is generated by a Spring plugin at runtime, and can be explored through Swagger UI at the following path: http://localhost:8000/docs (assuming the backend is running on localhost).

The repository layer uses Hibernate to communicate with the underlying RDBMS (PostgreSQL). Hibernate is the default ORM (Object-Relational Mapper) of choice for Spring Boot, and it allows us to easily define entities as Java classes and manage transactions in a transparent way.

2.2 - Backend Source Code Structure

The following is the directory tree of the CKB backend along with a description of each package:

- `src/main`: Where the main application packages are located.
  - `java/it.polimi.codekatabattle`:
    - `config`: Classes defining application configurations.
    - `controllers`: Classes defining API endpoints.
    - `entities`: Classes defining database entities (Hibernate).
    - `exceptions`: Classes defining custom exceptions.
    - `models`: Classes defining DTOs.
    - `repositories`: Classes defining repositories to access data.
    - `services`: Classes defining service interfaces.
      - `impl`: Implementation of the service interfaces.
    - `CodeKataBattleApplication.java`: The entry point of the application.
  - `resources/application.properties`: Defines application runtime properties in key-value format.
- `src/test`: Where the application test packages are located.
- `.gitignore`: Defines files ignored by Git.
- `pom.xml`: Defines application version, dependencies and plugins.

2.3 - Frontend Framework

The framework of choice for the frontend part of the application is **React** (https://react.dev/), an open source library made by Meta to simplify frontend development. The main objective is to allow developers to easily define “reactive components”, that is: components which are automatically rendered when the underlying state of the program changes. This is notoriously difficult to do in plain JavaScript. As such, React is used by the industry a lot and (just like Spring) is thoroughly battle-tested.

The language of choice is TypeScript. TypeScript is a superset of JavaScript which allows defining static types. This greatly increases the maintainability of the application, and also allows to define models which “mirror” those defined in the backend. To keep these models (and the services calling the API endpoints) in sync with the backend updates, an OpenAPI code generator is used.

The bundler of choice is Vite. Vite is an opinionated tool which allows to easily bundle a frontend application in various frameworks. It is stable and thoroughly tested in a lot of production environments.

The CSS is managed through Tailwind CSS. Tailwind defines utility classes which can be composed to create complex layouts without worrying about CSS inheritance and ordering. As such, it is a great choice for projects dominated by asynchronous work by different people or even teams. Daisy UI is a framework-agnostic library of reusable components written using Tailwind.

2.4 - Frontend Source Code Structure

The following is a description of the directory tree of the CKB frontend:

- `src`:
  - `assets`: Static assets (e.g. images).
  - `components`: Reusable React components which can be composed to create page views.
  - `context`: React contexts are used to allow components to enjoy "shared state". For example, the AuthContext defines information about the currently logged in user which can be accessed by all components.
  - `routes`: Routes are React components which describe page views. The application is a SPA (Single Page Application), so this is just simulated routing.
  - `services/openapi`: Classes generated by the openapi command. Must not be manually edited, always generated through the `npm run types:openapi` command.
  - `AppWrapper.tsx`: A React component exposing contexts and wrapping the underlying application.
  - `main.css`: Main CSS file, used to set up Tailwind.
  - `main.tsx`: Application "entry-point". Defines the router.
  - `vite-env.d.ts`: TypeScript Vite compatibility config.
- `.env`: Defines public environment variables. Must not contain secrets, the variables are readable by anyone.
- `.eslintrc.cjs`: Defines ESLint configuration.
- `.gitignore`: Defines files ignored by Git.
- `index.html`: Defines the "skeleton" index.html where the SPA resides.
- `package.json`: Defines application version, dependencies, and scripts.
- `package-lock.json`: Dependencies pinned versions, must be indexed by VCS to ensure that every developer has the correct versions of the application dependencies.
- `postcss.config.js`: Defines PostCSS config (used to compile Tailwind classes into CSS files cross-compatible between browsers)
- `tailwind.config.js`: Defines Tailwind CSS configuration.
- `tsconfig.json`: Defines TypeScript configuration.
- `vite.config.ts`: Defines Vite configuration.

3 - Adopted Testing Frameworks

3.1 - Backend Integration Testing

Backend tests are performed in an automatic way by using the tooling provided by Spring Boot. Tests are run against the API contract, and set up different scenarios to test responses, both for OK and KO responses.
Before deploying on the production environment, all tests are automatically run to ensure that no regressions are introduced for the services covered by the automated tests.

The implemented tests are integration tests: that is, they require a physical database running on the system to run, it is not mocked. Mocking is avoided in the integration tests to emulate as much as possible the actual application environment.
Automatic provisioning of the PostgreSQL instance is accomplished through the usage of Testcontainers for Java, which relies upon Docker to create, connect to, and destroy the provisioned database. After all tests are run, the database is destroyed.

3.2 - Frontend Acceptance Testing

The frontend does not integrate automatic testing for the moment, since it has been deemed that integration tests against the OpenAPI schema on the backend are enough for acceptance at the MVP stage.

4 - Effort Spent

Group Member	Effort Spent (hours)
Nicolò Giallongo	1h
Giovanni Orciuolo	6h
Giuseppe Vitello	1h

5 - References

- Version Control System: GitHub (https://github.com)
- Document written with: Google Documents (https://docs.google.com)