CodeMaster Final Report

Alex Testa
alex.testa@studio.unibo.it

Riccardo Rambaldi
riccardo.rambaldi4@studio.unibo.it

October 2025

Contents

Introduction

CodeMaster is a distributed system to support collaborative coding challenges, inspired by coding platforms such as LeetCode. With **CodeMaster** every user can create, manage and resolve different coding problems, using different languages.

In fact, its primary goal is to provide an extensible and scalable environment to practice problem solving, evaluate algorithmic solutions, and integrate advanced architectural concepts.

CodeMaster is designed not only as a learning tool, but also as a practical environment to strengthen and refine individual coding skills through continuous practice and feedback. For this reason, the platform is intended not only for experienced programmers, but also for beginners who are approaching the world of computer science for the first time.

The system provides services for solution submission, problem management, evaluation, and user interaction. Each service communicates through lightweight APIs and asynchronous messaging, enabling fault isolation, high availability, and independent scalability. This design choice supports distributed deployment and eases the integration of new features without disrupting the overall system.

From an educational perspective, CodeMaster is not only a coding practice tool but also a case study in modern distributed systems. This demonstrates how principles such as service decoupling, resilience, observability, and scalability can be applied in practice. Furthermore, the project highlights challenges in consistency management, interservice communication, and error handling in distributed environments. By combining practical coding features with advanced architectural design, CodeMaster provides both end users and developers with a robust environment.

1 Glossary

The initial stage of the knowledge-crunching phase involves establishing the project's ubiquitous language. This language serves as a common vocabulary shared among all team members, enabling clear and consistent communication when discussing the project's domain. By aligning terminology across developers, designers, and stakeholders, misunderstandings are minimized and collaboration becomes more effective. To build this shared language, we first identified the key concepts and entities that define the project's domain. Each of these elements was then carefully described and documented in a glossary, which serves as a reference point for the entire team throughout the development process.

Term	Definition
User	A user who utilizes the system and is authenticated. It also contains information regarding the user itself.
System	The application as a whole.
CodeQuest	A programming challenge consisting of a problem description and illustrative examples that can be solved by users. CodeQuests can also be created by other users.
CodeQuest codes	Upon the creation of a CodeQuest, the code generation microservice produces function templates and corresponding test cases for each selected programming language. These test cases are then used to evaluate the correctness of user-submitted solutions.
Solution	A code submission provided by a user to solve a specific CodeQuest. A solution may include multiple implementations across the different programming languages supported for that CodeQuest.
Comment	A textual contribution associated with a CodeQuest, authored by users to share feedback, insights, or discussions.
Language	A programming language that can be used to implement solutions for a CodeQuest. This entity has different domain models depending on the context.
Trophy	An achievement awarded to a user upon executing some tasks, such as successfully solving one or more CodeQuests.
Level	By solving CodeQuests, users gain experience points. As these points accumulate, the player advances to higher levels.

Table 1: Glossary of key system terms

2 Concept

CodeMaster is a distributed web-based platform for solving algorithmic coding challenges. It can be classified as a *web-service application* with a browser-based user interface, supported by a backend designed according to microservices and Domain-Driven Design (DDD) principles.

The platform provides users with the ability to submit code solutions, receive automated evaluation, and track their progress across different challenges.

- Users can access the system remotely from anywhere with an internet connection.
- Interaction is on-demand: users log in whenever they want to practice, typically multiple times per week depending on their learning goals.
- Users interact through a modern web interface, accessible from desktops, laptops,

or tablets.

- The system stores account information (credentials, profile, progress), coding submissions, and challenge metadata. Each microservice is independent and store data in its own database.
- **Roles.** The platform defines at least two roles:
  - *Users*, who create, manage, solve coding challenges (called **CodeQuests**) and review feedback.
  - *Administrators*, who manage, handle moderation tasks, and maintains the platform.

2.1 Use Cases

User authentication

A user who wants to access CodeMaster platform have to:

1. Sign in with username and password, if the user has already an account.
2. Sign up providing a nickname, email and password.

Once the authentication is completed, the user can access the dashboard

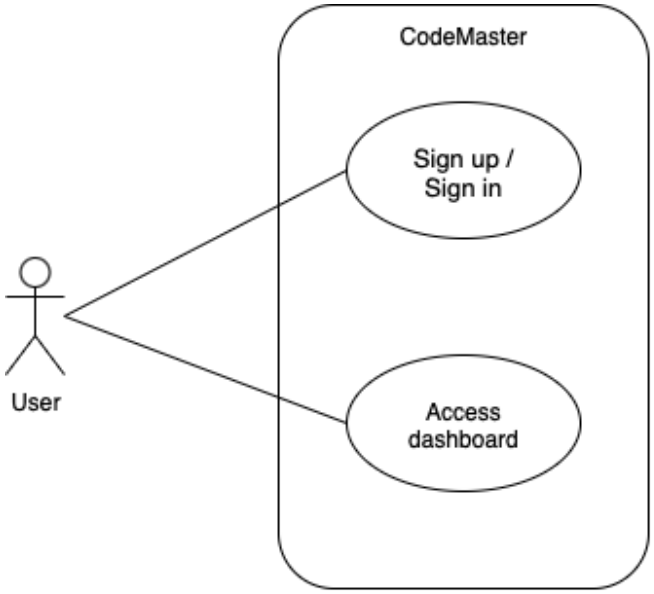

Figure 1: User authentication use case diagram

Profile

From the profile section, an authenticated user can:

1. log out from Codemaster
2. change information from the profile, such us CV, preferred languages, bio and profile picture

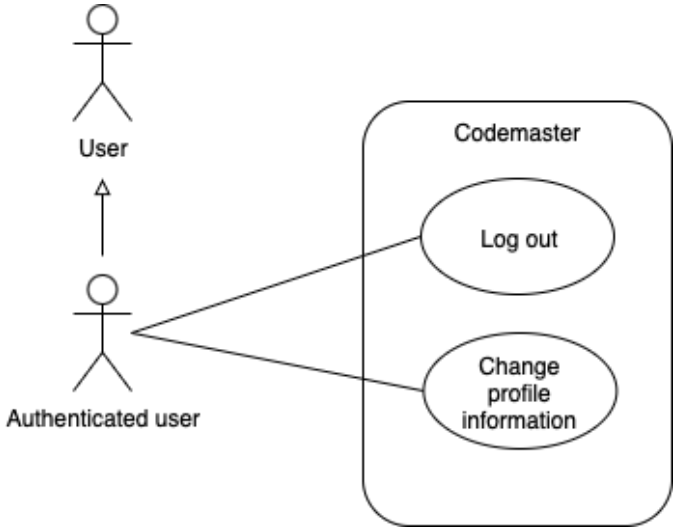

Figure 2: Profile management use case diagram

Codequest

Authenticated users can manage codequests by:

1. creates new codequests that will be visible and solvable by other users
2. deleting codequests only created (previously) by him
3. viewing codequests created by other users

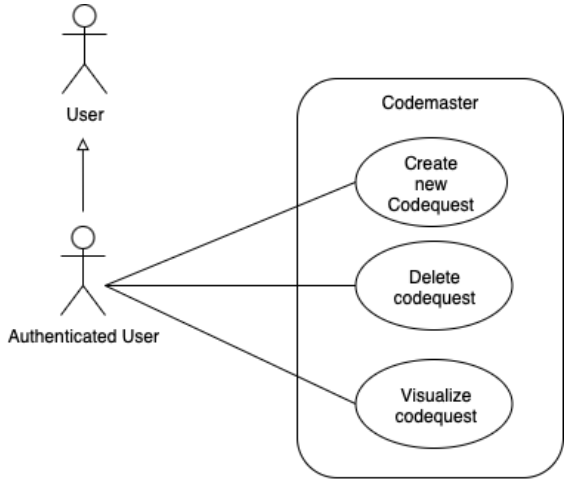

Figure 3: Codequest management use case diagram

Solution

Once an authenticated user opens a codequest, he can:

1. creates a new solution, or opens and modifies an existing one created by him
2. debug solution's code with feedback
3. run the code and checking the solution proposed by submission

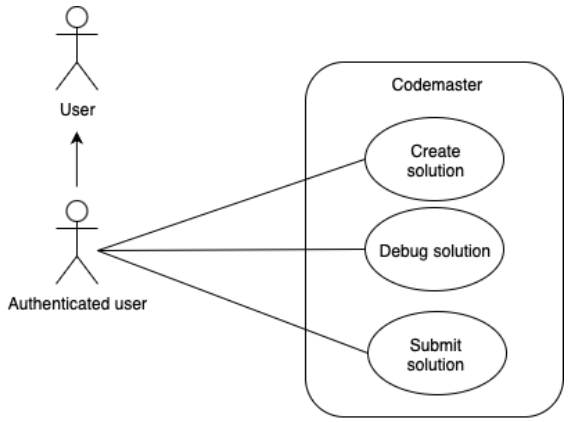

Figure 4: Solution use case diagram

Community

Within the solution section, an authenticated user can:

1. add new comments related to that codequest

2. view all comments added by other users, related to that codequests
3. share solution

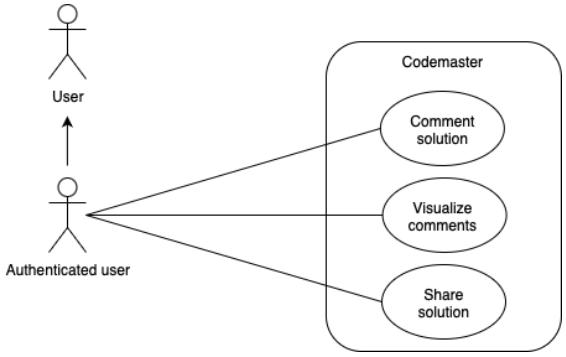

Figure 5: Comments use case diagram

Admin authentication

Administrator users are predefined within the system and granted special privileges, while still authenticating through the same login process as regular users.

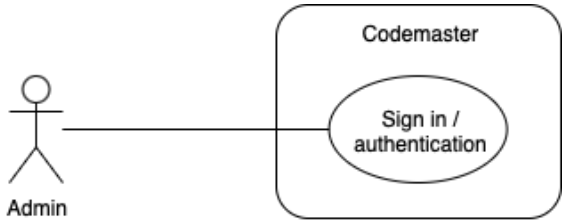

Figure 6: Admin authentication use case diagram

User and codequest management

Once an administrator has successfully authenticated, they can:

1. ban or unban a user
2. delete a codequest from the complete list of codequests
3. log out from CodeMaster

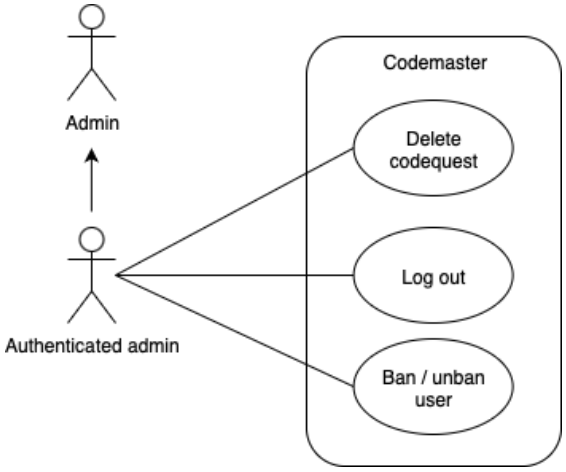

Figure 7: Admin use case diagram

3 Requirements

This section clarifies that all the requirements listed below have been fully satisfied by the developed software. Compared to the initial proposal, no substantial changes, removals, or modifications have been made to either functional or non-functional requirements.

The requirements are presented in two distinct categories, corresponding to two types of users:

- **End User:** represents the final user, a person who accesses and interacts with the platform in the intended manner. This profile is inspired by the user personas that will be described in later chapters.
- **Administrator:** an internal system user with elevated privileges necessary for managing, moderating, and maintaining the platform.

4 End User Requirements

1. Users must be able to register and authenticate themselves.
2. Users must be able to log out.
3. Users must be able to view and edit their profile, including their profile picture (selected from those available on the platform), bio, preferred programming languages, and résumé.
4. Users must be able to create CodeQuests and define the programming languages appropriate for solving them.
5. While creating a CodeQuest, users must specify the tests that validate the solution.

6. Users must be able to edit or delete previously created CodeQuests.
7. Users must be able to submit solutions to both their own CodeQuests and those shared by others.
8. Users must be able to view their current level (e.g., Beginner, Novice, Expert, etc.).
9. Users must be able to see their most recently earned trophy.
10. Users must be able to adjust visual preferences, such as enabling dark mode or light mode.
11. Users must be able to filter CodeQuests by difficulty level.
12. Users must be able to track which CodeQuests they have completed successfully.
13. Users must be able to share a CodeQuest with others.

5 Administrator Requirements

1. Administrators must authenticate to access the system.
2. Administrators cannot self-register and are internal system accounts.
3. Administrators must be able to ban users who violate community guidelines.
4. Administrators must be able to remove CodeQuests that do not comply with platform rules or that cannot currently be solved.

6 Non-Functional Requirements

1. The system must ensure a high level of security: sensitive data must never be stored in plain text.
2. The system must protect against header-based request attacks.
3. The user interface must meet WCAG AA color contrast standards.
4. The system must always provide meaningful feedback to users.
5. The system must be compatible with the following browsers:
   - Firefox
   - Chrome
   - Safari
6. The system should leverage opportunities inspired by competitor solutions such as LeetCode.

7. The user interface must be fully responsive.
8. The system must remain highly reactive and performant.
9. The platform must be intuitive and accessible, even for users new to programming.
10. The system must be easy to install.
11. The interface must be developed following a mobile-first design approach.

6.1 Implementation Constraints

1. The system shall be developed as a set of microservices, following Domain-Driven Design principles. This approach ensures modularity, scalability, and separation of concerns, aligning with the educational goals of the project.

   Acceptance: Each service can be deployed and scaled independently.

2. Most of the microservices shall be realized according to the **MEVN stack** (MongoDB, ExpressJS, VueJS, and Node) using the TypeScript programming language. The MEVN stack is a full-stack JavaScript framework that combines MongoDB, Express.js, Vue.js, and Node.js, enabling developers to build dynamic web applications using a single language across both client and server sides.

   Acceptance: Each microservice implemented with the MEVN stack runs independently, exposes a clear API, and is able to interact seamlessly with other services in the distributed environment.

3. The system shall use Docker to containerize each microservice. Containerization guarantees reproducibility, simplifies deployment, and enables scaling across heterogeneous environments. It also eases continuous integration and delivery [15, 23].

   Acceptance: Each microservice can be built and deployed as a Docker container image.

4. The system shall execute user-submitted code inside isolated containers, orchestrated using Kotlin-based services. Kotlin ensures type safety and supports integration with container management libraries. Executing code inside dedicated containers preserves security and scalability when concurrently running multiple code-executing containers [11, 23].

   Acceptance: User code runs inside a dedicated container launched by a Kotlin microservice, and its output is returned to the platform.

5. The system is implemented using TypeScript and Kotlin for the backend. TypeScript provides strong typing and maintainability for JavaScript services, reducing runtime errors, improving developer productivity, and facilitating large-scale application development by enforcing structured code and enabling advanced tooling support [31, 2].

Acceptance: Core services compile and run successfully in the target environments.

6. The system shall store persistent data in a NoSQL database.

   Acceptance: All domain entities remain available across sessions and survive system restarts.

7 Design

7.1 Architecture

In order to implement a distributed and reactive environment for **CodeMaster**, we decided to divide the system into multiple *microservices*. This choice guarantees modularity, fault isolation, and independent scalability, which are fundamental properties for a platform expected to grow in both features and user base.

The architecture is organized around bounded contexts, following Domain-Driven Design (DDD) principles. Each microservice encapsulates a specific business capability, such as user management, problem (CodeQuest) management, code execution, commenting, or achievement tracking. Services communicate through lightweight APIs and asynchronous messaging, ensuring loose coupling while still enabling integration.

We chose this architecture for several reasons:

- **Scalability:** By splitting the platform into independent services, it becomes possible to scale only the components under heavy load (e.g., the code execution service) without affecting the others.
- **Resilience and fault tolerance:** If one service fails, the others can continue operating, reducing the impact of failures on the overall system.
- **Flexibility and extensibility:** New features can be introduced as separate services, without requiring modifications to the core system. This makes the platform adaptable to future needs, such as integrating new programming languages or adding new collaboration tools.
- **Distributed deployment:** Microservices can be deployed across different nodes or environments, which is consistent with the design goals of a distributed system and allows better resource utilization.
- **Alignment with DDD**: Microservices align naturally with the concept of bounded contexts, enabling a clearer mapping between the domain model and the software architecture.

Overall, the chosen architecture supports the educational and collaborative objectives of CodeMaster while adhering to the principles of distributed systems: modularity, reactivity, and scalability.

7.2 Infrastructure

The infrastructure of **CodeMaster** is designed to support a microservices-based architecture, with a clear separation between frontend and backend services. The infrastructure includes the following components:

- **Frontend**: client application served to users through a web browser. It communicates with backend services via HTTPS REST APIs.
- **Backend microservices**: developed with Node.js/TypeScript and Kotlin, each implementing a specific bounded context (e.g., user management, CodeQuest management, code execution, commenting, achievements).
- **Databases**: MongoDB instances are used for flexible storage of user data, problem statements, and submissions. Relational or distributed databases may be used where strict consistency is required.
- **Message broker**: RabbitMQ is introduced as a middleware for asynchronous communication between microservices. It is particularly used for resource-intensive tasks such as code execution, ensuring that requests are queued and processed by worker containers without blocking other services. By decoupling service interactions and supporting reliable message delivery, RabbitMQ enhances scalability, fault tolerance, and overall system responsiveness [25].
- **Reverse proxy**: a reverse proxy is used as an entry point to the system. It routes incoming traffic to the correct backend services, provides load balancing, improves security by hiding internal service details, and enables features such as SSL termination, caching, and request rate limiting. By centralizing these responsibilities, Nginx helps ensure high availability, scalability, and performance for the overall architecture [20].
- **Containers**: All microservices and infrastructure components are packaged into Docker containers, which guarantees portability, reproducibility, and simplifies orchestration.

In the current deployment model, services and infrastructural components are distributed across multiple containers within the same machine. The chosen infrastructure ensures that services remain modular, discoverable, and independently deployable.

7.3 Microservicies

Microservices are an architectural style in which an application is composed of small, independent services. Each microservice encapsulates a specific business capability, can be developed and deployed independently, and typically owns its own database, which ensures decoupling and scalability [19, 35]. In our system, asynchronous communication for cross-service updates, such as deletions, is handled through a message broker provided by the RabbitMQ middleware.

UserService

The UserService manages and stores user account information and trophies.

AuthenticationService

The AuthenticationService handles user login and logout, verifying credentials and maintaining session information.

CodeQuestService

The CodeQuestService manages the creation, storage, and maintenance of codequests submitted by users.

CommunityService

The CommunityService allows users to add comments to codequests.

SolutionService

The SolutionService handles storing and managing solutions submitted by users. Additionally, it executes user code inside isolated and dedicated containers for security and scalability.

CodeGeneratorService

The CodeGeneratorService automatically generates test cases and function templates for solutions based on user-provided examples and specifications when creating a codequest.

FrontendService

The FrontendService provides the client-facing interface of the system and interacts with all backend services via REST APIs.

ApiGateway

The ApiGateway acts as an entry point for client requests, routing them to the appropriate backend services and handling load balancing and security concerns using Nginx.

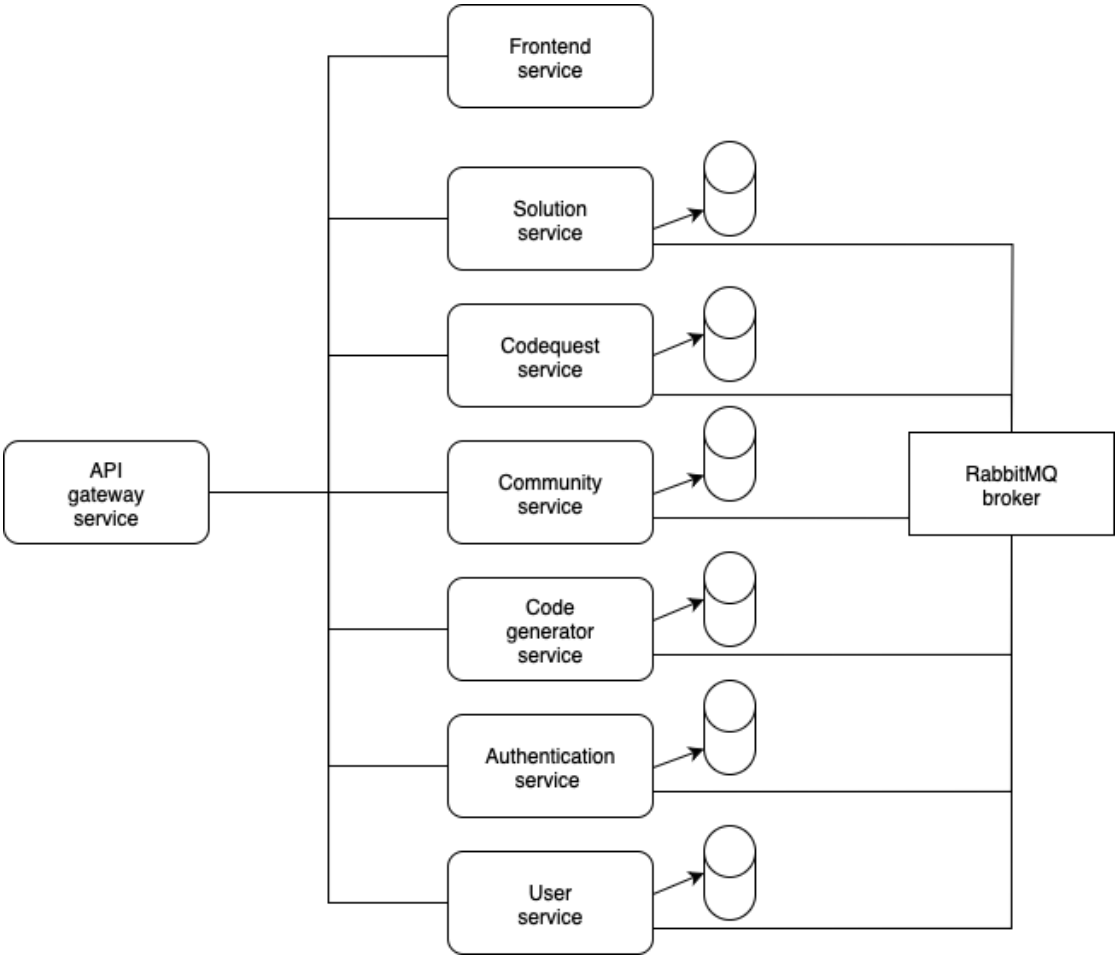

Figure 8: Codemaster infrastructure

Each services is organized, following the Domain Driven Design guide lines, into multiple modules as following:

- **Domain:** Encapsulates the core business logic of the service. This module defines all the domain entities, including aggregate roots, value objects, and entities, which collectively represent the domain model of the microservice. It also provides the corresponding factories responsible for the consistent creation of the entities.
- **Infrastructure:** Provides the technical implementations required by the domain and application layers, such as repositories, messaging, and configurations.
- **Application:** Coordinates the use cases of the microservice. It does not implement business logic itself but orchestrates domain operations, handling transactions, application flows, and interactions between modules.

- **Interfaces:** Exposes the functionalities of the microservice through REST APIs. It serves as the entry point for other systems or users.

7.4 Modelling

Domain Entities

- **CodeQuest:** A programming challenge consisting of a problem description and illustrative examples that can be solved by users. Codequests can also be created by other users.
- **CodeQuest codes:** Upon the creation of a CodeQuest, the code generation microservice produces function templates and corresponding test cases for each selected programming language. These test cases are then used to evaluate the correctness of user-submitted solutions.
- **Solution:** A code submission provided by a user to solve a specific CodeQuest. A solution may include multiple implementations across the different programming languages supported for that CodeQuest.
- **Comment:** A textual contribution associated with a CodeQuest, authored by users to share feedback, insights, or discussions.
- **Language:** A programming language that can be used to implement solutions for a CodeQuest. This entity has different domain models depending on the context.
- **Trophy:** An achievement awarded to a user upon executing some tasks, such as successfully solving one or more CodeQuests.
- **Level:** By solving codequests, users gain experience points. As these points accumulate, the player advances to higher levels.

All domain entities have been systematically mapped to their corresponding infrastructural components, such that each entity is encapsulated and managed by a dedicated microservice.

- User, Trophy, Level → User Service
- Language → Solution Service, Codequest Service, CodeGeneration Service, User Service
- Comment → Community Service
- Codequest → Codequest Service
- Solution → Solution Service

This replication of Language domain entity with different meanings is necessary in Domain Driven Design approaches: preserving independency and a clear separation of responsibility between all microservices is the main principle of these type of software modeling techniques. It promotes a clear separation of bounded contexts, ensuring that each service maintains autonomy over its own domain logic. By structuring the architecture in this manner, the system achieves modularity, maintainability, and scalability.

In the following sections, are presented the class diagrams corresponding to each domain entities. These diagrams provide a detailed view of the internal structures, relationships, and associations of the domain entities, illustrating how they are represented within the system's infrastructural components.

- **User Service**

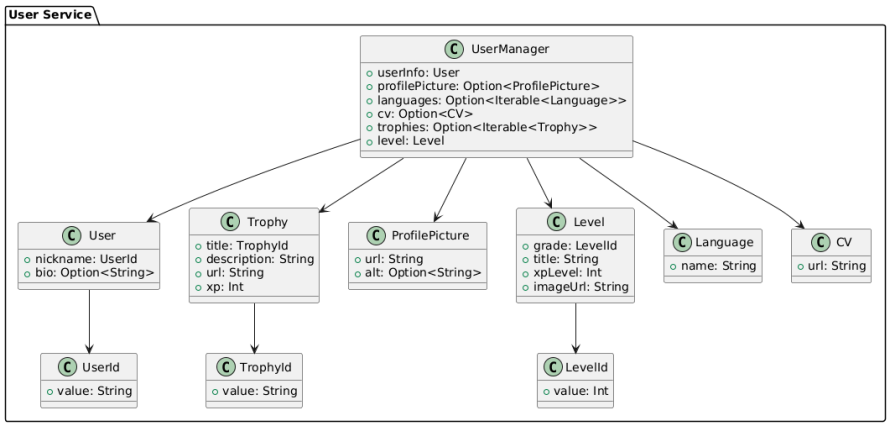

Figure 9: User class diagram

  - UserManager is an aggregate root
  - User, Trophy and Level are entities
  - All other classes are value objects

- **Authentication Service**

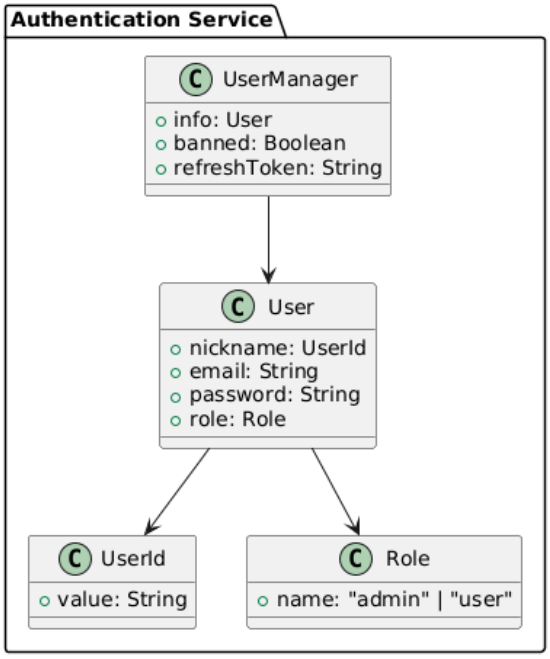

Figure 10: Authentication class diagram

  - UserManager is an aggregate root
  - User is an entity
  - UserId and Role are value objects

- **Codequest Service**

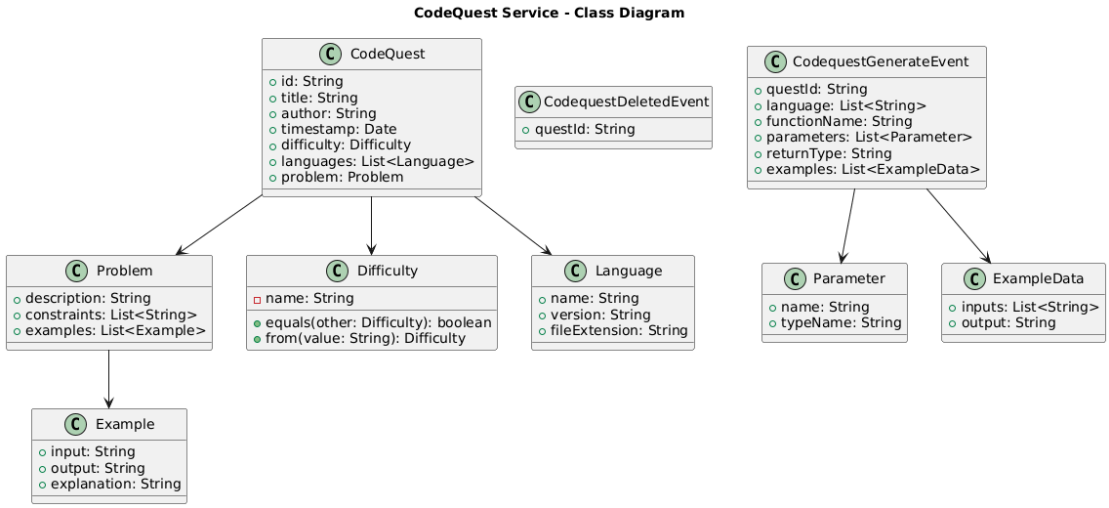

Figure 11: Codequest class diagram

- CodeQuest is the main entity
- User is an entity
- CodequestDeletedEvent and CodequestGenerateEvent are domain events

- **Community Service**

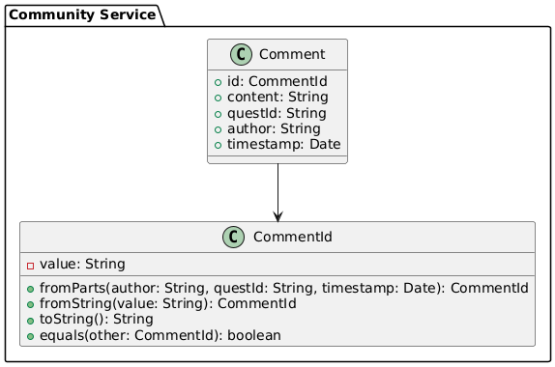

Figure 12: Community class diagram

- Comment is the main entity while Id is a value object

- **Solution Service**

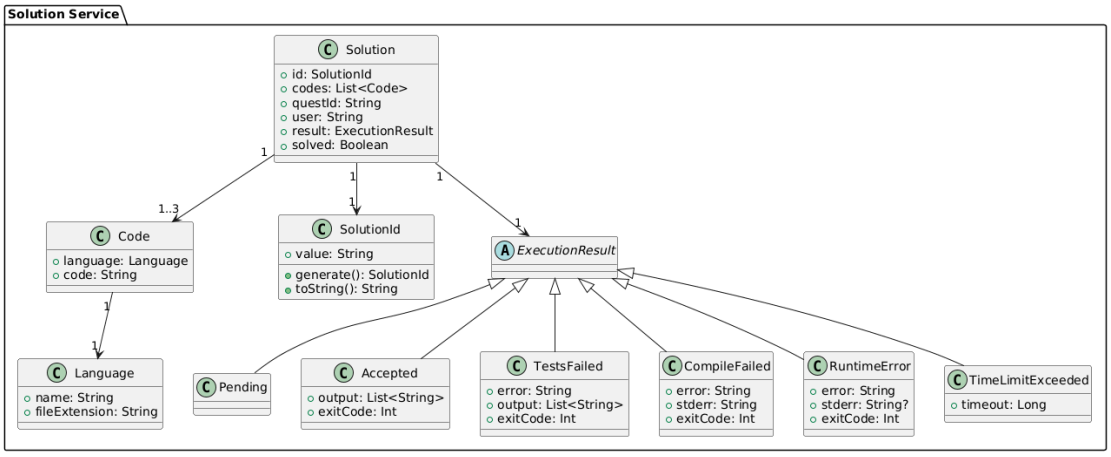

Figure 13: Solution class diagram

- Solution is the main entity
- ExecutionResult is a Result Value Object, encapsulated within the Solution entity. Each type of solution result represent a possible output/error obtained from the code execution

- **CodeGenerator Service**

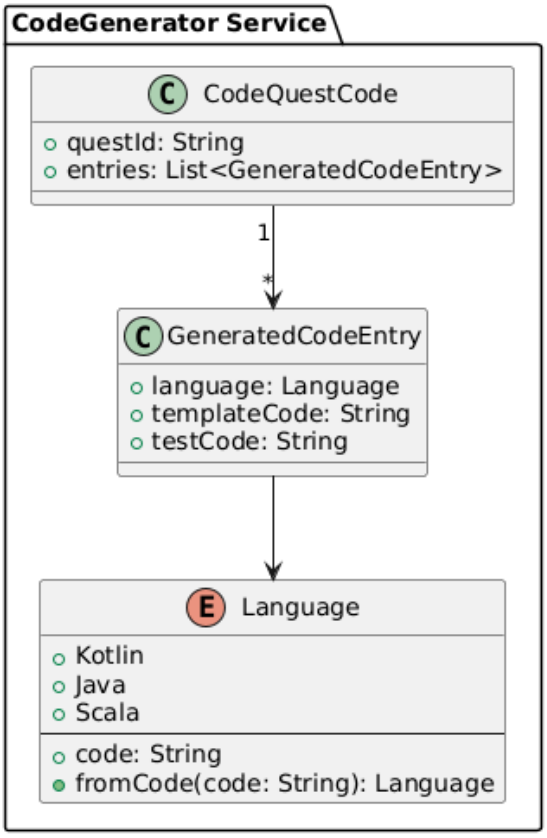

Figure 14: Code generator class diagram

– CodeQuestCode is the main entity that represent, for each codequest, a list of testCode and templateCode for each language available

Domain Events

- **UserRegistered:** Triggered when a user registers and authenticates on the platform for the first time. Represents the creation of a new user in the system.
- **UserDeleted:** Triggered when a user is removed from the system. Represents the removal of all data associated with that user.
- **CodequestDeleted:** Triggered when a codequest is deleted or when its owner is removed. Represents the removal of the affected codequest from the system.

Commands

Messages exchanged within codemaster can be **event** or **commands**. Commands are messages sent directly to a specific destination. The only command we use is **CodequestCodegen**. This command is published from codequest service to code generator

service when a new codequest is created by a user. Once the generator receive the command, it start to generate codequest's template and tests code.

The state of the system comprehends all the persistent domain information managed by the different services. Specifically, it includes:

- **User data**: profile information, authentication credentials, profile preferences.
- **Codequest data**: codequests information, ownership, examples.
- **Community data**: all comments created by users associated to a codequest.
- **Solution data**: user-submitted solutions, evaluation results, tests.
- **Generated code artifacts**: source code and tests associated to codequests.

On the other hand, there are elements that do not form part of the persistent system state. Those elements are transient or temporary and include:

- **Messages in RabbitMQ queues**: they are transient communication artifacts used to propagate events and commands between services
- **JWT tokens**: they are temporary authentication credentials that exist only during a user session and are not persisted as part of the domain state

7.5 Interaction

Within the system, components communicate in different ways depending on the context:

- The frontend service interacts with microservices through their APIs using HTTP requests, triggered by user actions such as registration, login, or submitting a codequest.
- The API Gateway translates incoming HTTP requests to the appropriate backend service URL, effectively functioning as a reverse proxy and routing requests to the correct service.
- Backend microservices communicate asynchronously with each other via the RabbitMQ message broker, typically in response to domain events, commands, or queries that require coordination between services. The communication is asynchronous and event-driven.

The following two sequence diagrams illustrates an example of the activation stack that occurs when a event is triggered or a command is forwarded.

The first diagram illustrates the activation stack when a UserDeleted event is triggered. The AuthenticationService publishes the event to a queue subscribed by UserService, CodequestService, CommunityService, and SolutionService. When a user is deleted, the CodequestService publishes a new event for each removed codequest to a dedicated queue, subscribed by CommunityService, SolutionService, and CodeGenerationService,

ensuring consistent and complete information across all components. Both for user deleted and codequest deleted events, the communication is broadcast.

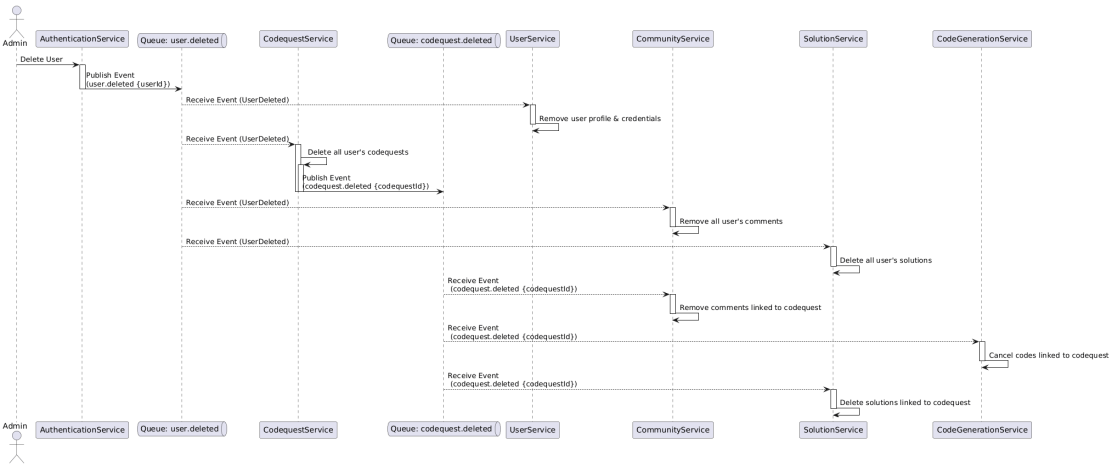

Figure 15: Delete User Sequence Diagram

The last diagram instead, shows the activation stack of a code generation command that starts when a user create a new codequest. This is a point-to-point communication.

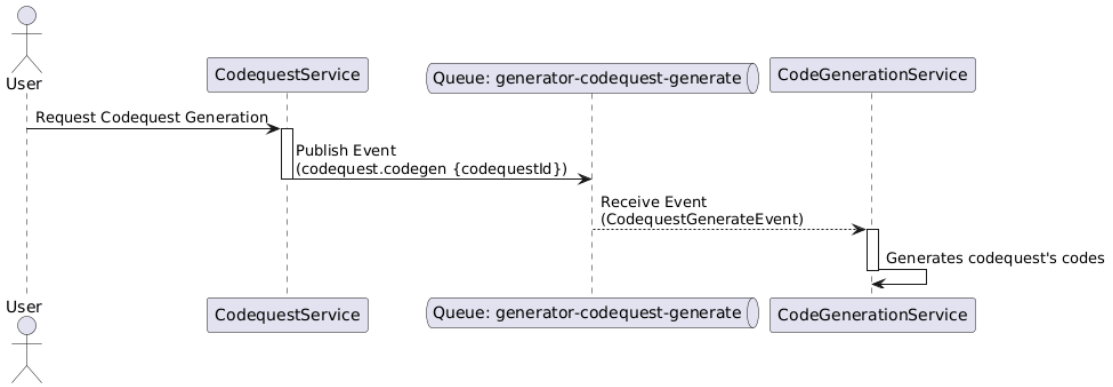

Figure 16: Code Generation Sequence Diagram

7.6 Behaviour

All of the aforementioned services are stateful, as each microservice is entrusted with the management and persistence of a specific domain entity. Accordingly, they are responsible for updating and maintaining the overall system state.

Service	Behavior / Response to Events
Authentication	Publishes events such as `UserRegistered` or `UserDeleted` in response to user actions.
User	Upon `UserRegistered`, creates and persists a new user. Upon `UserDeleted`, removes the user data from persistent storage.
Codequest	In response to `UserDeleted`, deletes all codequests of the user and publishes a `CodequestDeleted` event for each. In response to generation requests, sends messages to `CodeGenerationService`.
Community	In response to `UserDeleted` or `CodequestDeleted`, removes associated data.
Solution	Responds to `UserDeleted` and `CodequestDeleted` by deleting related solutions.
CodeGeneration	Receives `CodequestDeleted` events or generation requests (`CodequestGenerateEvent`) and processes code generation tasks.

Table 2: Individual behavior of system components in response to events or messages.

However, two services can be classified as stateless: the API Gateway Service and the Frontend Service. These components neither manage domain entities nor maintain any form of persistent storage; rather, they are limited to handling transient data solely for the purpose of request routing and user interaction.

7.7 Data and Consistency Issues

Each stateful microservice maintains its own persistent data. This separation and independency guaranteed by the microservice architecture permit to adopt a document-oriented model using MongoDB.

NoSQL database fits naturally with a Domain Driven Design approach. Each aggregate root can be stored as a single document, including its value objects, which avoids complex joins and keeps persistence aligned with the domain model. Moreover, MongoDB offers flexibility when the domain evolves, since its schema-less nature allows incremental changes without heavy refactoring. Finally, isolating one database per bounded context becomes straightforward, and atomic operations on documents ensure consistency while supporting scalability.

Queries are executed by each microservices through its own repository. This component is the database interface that expose main queries to other components in the microservice. Typical operations include:

- authorization and authentication by storing user's credentials in Authentication Service
- profile information retrieval and update in the User Service

- codequest generation, retrieval and deletion in the Codequest Service
- solutions submission, update and evaluation in the Solution Service
- comments retrieval in the Community Service

The system supports both concurrent reads (e.g., many users browsing at once) and concurrent writes (e.g., simultaneous submissions), with consistency guaranteed locally within each service database.

Persistent data is never shared directly, instead services communicate asynchronously through domain events exchanged via RabbitMQ. Only transient data, such as JWT tokens for session management, is passed between services but is not stored persistently.

7.8 Fault-Tolerance

Fault-tolerance in the proposed architecture is achieved through a combination of timeouts, retry mechanisms, error handling strategies, and real-time monitoring. The overall design philosophy is to ensure that failures of individual services do not compromise the entire system, while providing mechanisms for graceful degradation.

Health-checks

First of all, each microservice exposes a dedicated health-check endpoint (`/status`) that simply returns a `200 OK` response when the service is running.

This mechanism is especially useful within `docker-compose`, as it allows dependencies to be orchestrated correctly: for instance, services such as the API Gateway and the Frontend Service only start once all backend services are marked as healthy. In addition, even RabbitMQ itself is equipped with a container-level health check, ensuring that the message broker is fully operational before all backend service containers are launched.

Timeouts and Error Handling

All API requests from the frontend to backend services are configured with a 5 seconds timeout. This ensures that, in the event a service becomes unavailable, the request fails gracefully instead of leaving the client in a blocked state.

In such cases, the system simply communicates the unavailability of the services to the user, without exposing raw errors. This strategy allows the platform to remain partially operational even if some backend services are temporarily down, with the notable exception of the Authentication Service, which is essential for user access.

For example, if the user service is unavailable, it is not possible to fetch user's profile data, but the user's still able to create and solve codequests. A special case arises during the creation of new codequests: if the Code Generation Service is unavailable, the Codequest cannot be created, even though the Codequest Service itself is active. This is due to the necessity of generating templates and test code, without which a Codequest would be functionally incomplete.

Retry Mechanism

A retry strategy is also implemented at the reverse proxy level. Each upstream service is configured with a `max_fails` (3) threshold and a `fail_timeout` (30 seconds) window.

This mechanism allows the proxy to temporarily mark a service as unavailable after a number of failed attempts, and then retry after the timeout interval. Such a configuration reduces the impact of transient failures and increases the resilience of client-service communication.

Message Queues

RabbitMQ plays a crucial role in maintaining system consistency. All significant state changes, such as the deletion of a user or a Codequest, are published as events to dedicated message queues.

If a service becomes unavailable, these events remain stored in the queue until the service is restored. Once it resumes, the service consumes the queued messages and updates its state accordingly, ensuring that the system remains consistent across all components. This mechanism guarantees eventual consistency even in the presence of temporary faults.

Service Monitoring

A dedicated dashboard is available within the frontend, providing real-time insights into the health of all microservices. This monitoring capability enables users and administrators to immediately detect faults, verify service availability, and take corrective actions when necessary. Together with the health-check routes and container-level monitoring, this dashboard contributes to proactive fault detection and operational reliability.

The idea work around some key principles. First, no individual backend service (with the exception of the Authentication Service) is strictly necessary for the system to function. This aligns with the principle of resilience through decentralization, as failures in peripheral services do not prevent the rest of the system from operating. Moreover, the use of RabbitMQ ensures eventual consistency. The integration of timeouts, retries, and monitoring mechanisms reflects a design focused on fault isolation, self-healing, and graceful degradation.

7.9 Availability

While fault-tolerance focuses on how the system reacts to service failures and recovers from them, availability specifically addresses the mechanisms that keep the platform responsive and operational under varying conditions, such as workload peaks or temporary network partitions.

Load Balancing

The system leverages the reverse proxy configuration as a basic form of load balancer. Each upstream service is registered with retry and fail-over policies, enabling the proxy

to temporarily exclude unhealthy services and redirect requests accordingly.

Network Partitioning

In the case of a network partition, direct communication between services may fail. However, the use of RabbitMQ as a message broker ensures that events are durably stored in queues until the affected services are reachable again.

In practice, this means that state changes such as the deletion of a user or a Codequest are never lost, but may be asynchronously processed by dependent services after recovery. This trade-off allows the platform to remain operational even under network instability.

7.10 Security

The system's security is ensured through three main pillars: authentication, authorization, and cryptographic measures.

Authentication

User identity is verified to ensure that only legitimate clients can access protected resources. The system relies on token-based authentication to provide stateless and secure session management.

Authorization

Access control is enforced via roles. Each user is assigned a role (e.g., regular user, administrator) which determines their permissions. This ensures that users can only perform actions they are entitled to.

Cryptographic Measures

Sensitive data such as passwords is protected using industry-standard cryptographic techniques. All data transmitted between components is encrypted to maintain confidentiality and integrity.

---

8 Implementation

The system employs both synchronous and asynchronous communication mechanisms. Synchronous interactions between the frontend and backend microservices are performed via **HTTP/REST** requests. Asynchronous communication between backend services is handled through **AMQP** using RabbitMQ as a message broker.

All transmitted data, whether via REST APIs or message queues, is serialized in **JSON** format for simplicity, readability, and cross-language compatibility within the system.

Authentication.

The system uses **JSON Web Tokens (JWT)** to manage user sessions securely. JWTs allow information to be transmitted safely between client and server using a digital signature to guarantee integrity. This approach eliminates server-side session storage, making the architecture more scalable [13].

Password Security.

To protect user passwords, the **bcrypt** library has been used, a widely recognized standard in cybersecurity. Bcrypt generates secure, non-reversible hashes and applies automatic salting for each password. This ensures that even identical passwords from different users produce distinct hashes, making dictionary or rainbow table attacks extremely difficult. The combination of random salting and the continuous updating of the hashing algorithm helps guarantee that credentials are handled securely and in line with industry best practices [22].

Authorization.

Role-Based Access Control (RBAC) defines which operations each user can perform. Roles such as regular user or administrator determine access rights to services and frontend pages, ensuring that sensitive actions are restricted appropriately.

8.1 Technological details

The following section outlines the core technologies employed in building the microservices that form the CodeMaster architecture. These technologies were applied consistently across most services, with the exception of the *Solution-service*, which was implemented separately in Kotlin to address specific requirements related to code execution and validation. All other microservices share a common technological foundation, selected to ensure consistency in development, simplify maintenance, and streamline integration between system components. The chosen technologies range from mature backend frameworks to tools for asynchronous communication and security, reflecting a modern, modular, and scalable design approach.

8.2 MEVN Stack

To implement the CodeMaster architecture, the **MEVN** stack was adopted—an acronym for MongoDB [17], Express.js [8], Vue.js [33], and Node.js [21]. This stack is a widely used solution for full-stack web applications, combining efficiency, scalability, and consistency through the use of JavaScript on both the client and server sides.

For data persistence, **MongoDB** was selected as the primary database. Being a NoSQL, document-oriented system, MongoDB stores information in a JSON-like (BSON) format. Its distributed nature, horizontal scalability, and flexible schema design make it particularly well-suited for a microservices architecture. These features allow the

data model to evolve seamlessly over time, avoiding the rigid constraints of traditional relational databases.

On the server side, all microservices (except the Solution service and Code Generation service) run on **Node.js**, a JavaScript runtime built on Google's V8 engine. Node.js is known for its efficiency in handling asynchronous operations and its event-driven programming model, which makes it highly effective for systems with heavy workloads and high concurrency. Within each service, **Express.js** is used as a lightweight yet powerful framework for building RESTful APIs. Its clear syntax and modular structure for routes and middleware improve code readability and maintainability.

On the client side, the user interface is built with **Vue.js**, a progressive JavaScript framework designed for reactive, component-based applications. Vue's component philosophy enabled the UI to be organized into reusable logical blocks, improving maintainability, ensuring consistent design, and promoting code reuse across different parts of the application.

By leveraging a Single-Page Application (SPA) model, network traffic was reduced at the cost of increased client-side computation. However, Vue's gentle learning curve and extensive documentation made it a particularly suitable choice for an agile, fast-evolving project like CodeMaster.

8.3 Typescript

For the development of the application, **TypeScript** [31] was chosen a super-set of JavaScript that introduces static typing into an otherwise dynamically typed language. This decision was motivated by the need to produce safer, more robust, and maintainable code, while significantly reducing the likelihood of runtime errors. One of the most relevant features of TypeScript is its ability to detect errors already at compile time, thanks to explicit type declarations. This enables developers to identify bugs and inconsistencies before execution, increasing system reliability and reducing the time spent on debugging in later stages.

From a theoretical standpoint, static typing is widely regarded as a best practice in modern programming paradigms. It makes code behavior more predictable and easier to reason about, facilitating teamwork and supporting advanced development practices such as safe refactoring, object-oriented programming, and the automatic generation of documentation through tools like IntelliSense.

Within this project, TypeScript was seamlessly integrated with **Vue.js**, thanks to the framework's native support in recent versions. This allowed the benefits of Vue's component-based model to be combined with the reliability provided by static typing. In other words, the flexibility of front-end development was merged with the safety of a strongly typed language, ultimately enhancing code quality, readability, and the overall development experience.

8.4 Fp-ts

Alongside TypeScript, the project integrates the **fp-ts** library [3], a toolkit that systematically brings functional programming (FP) principles into the JavaScript/TypeScript ecosystem. The decision to adopt fp-ts stems from the goal of making the code more predictable, safer, and declarative by embracing core FP concepts such as immutability, pure functions, composition, and the explicit handling of side effects. Unlike the traditional imperative paradigm, this approach treats data as immutable values and functions as the primary units of behavior.

One of the library's most powerful features is its ability to further strengthen TypeScript's type system through algebraic data types and constructs such as `Option`, `Either`, `Task`, and `IO`. These abstractions make it possible to model error handling, asynchrony, effects, or the absence of values in an explicit and type-safe manner. As a result, error detection shifts from runtime to compile time, significantly improving the robustness of the application.

Moreover, fp-ts encourages writing modular and reusable code through function composition utilities like `pipe` and `flow`, which improve not only readability but also maintainability and testability of the overall system.

8.5 Helmet and JWT

To enhance the security of the project, several practices and dedicated libraries were adopted to safeguard the application against common attacks and vulnerabilities related to HTTP traffic and user credential management.

First, the middleware library **Helmet** was integrated into the Node.js and Express stack. Helmet strengthens HTTP headers in both requests and responses, acting as a defensive layer against well-known threats such as Cross-Site Scripting (XSS), Clickjacking, and MIME-sniffing [9]. By standardizing and hardening HTTP header configurations, Helmet reduces the attack surface of the system and increases its resilience against malicious intrusions and tampering.

For user authentication, the project relies on **JSON Web Tokens (JWT)**, a widely adopted technology for secure session management. JWTs allow information to be transmitted safely between client and server through a digital signature that guarantees integrity and authenticity. This stateless approach eliminates the need for server-side session storage, thereby improving scalability and aligning naturally with a microservices-based architecture.

8.6 Tailwind CSS

For managing the styling and visual aspects of the interface, the project adopted **Tailwind CSS** [14], a utility-first framework designed to simplify and accelerate CSS development. Unlike traditional approaches, Tailwind allows developers to apply CSS classes directly within the HTML markup, creating a clear and immediate connection between structure and style. This approach significantly reduced the amount of custom CSS written manually, streamlining the front-end workflow and increasing overall productivity.

One of Tailwind's distinctive strengths lies in its compositional flexibility. In contrast to frameworks like Bootstrap, which provide ready-made but often rigid components, Tailwind imposes no predefined design. Instead, it offers a large and modular set of CSS utilities that can be freely combined to build highly customized components. This allowed the development of a consistent yet original user interface, while maintaining full control over design and visual identity.

Tailwind also integrates seamlessly with **PostCSS**, enabling automated optimization of stylesheets during the build phase. For instance, unused classes are removed automatically, resulting in leaner CSS bundles and improved client-side performance.

Another key advantage is the built-in support for responsiveness. Tailwind includes ready-to-use responsive classes based on a well-defined breakpoint system. This made it straightforward to adapt the interface across different devices without writing manual media queries. As a result, the project benefits from a modern, responsive layout that is both clean and easy to maintain.

8.7 Linting and Prettier

To ensure high code quality as well as improved readability and maintainability, two widely adopted tools in modern JavaScript/TypeScript development were integrated: *ESLint* and *Prettier*.

ESLint [7] was employed to enforce consistent coding conventions and detect problematic patterns, such as poor practices, potential logical errors, or unintended side effects. This helped preserve stylistic coherence across the different modules of the project while also preventing hard-to-trace bugs.

In parallel, Prettier [26] was used as an automatic code formatter, responsible for unifying the visual style of the codebase—covering aspects like indentation, spacing, and punctuation—based on predefined rules. While ESLint focuses primarily on syntactic correctness and code quality, Prettier deals exclusively with formatting, ensuring the code remains clean and visually consistent.

8.8 Detekt and Kover

To maintain high code quality and ensure comprehensive test coverage in Kotlin microservices, the project adopted *Detekt* [1] and *Kover* [12].

Detekt is a static code analysis tool for Kotlin that helps identify potential code smells, enforce coding conventions, and maintain a consistent style across the project. By integrating Detekt into the build process, developers received immediate feedback on violations, allowing proactive improvements and reducing the risk of maintainability issues.

Kover is a Kotlin-specific code coverage plugin for Gradle that measures the percentage of code exercised by unit tests. This tool provided precise metrics about the effectiveness of automated tests and highlighted areas that required additional testing. The combination of Detekt and Kover ensured that the Kotlin microservices were not only functionally correct but also aligned with best practices for code quality and testing.

These tools were integrated into the CI/CD pipeline, enabling continuous quality checks and automated reporting of coverage metrics, which contributed significantly to the robustness and maintainability of the system.

8.9 Spring Boot for Kotlin Microservices

For the development of Kotlin-based microservices, *Spring Boot* [24] was chosen as the primary framework. Spring Boot provides a comprehensive ecosystem for building production-ready microservices with minimal configuration. Its embedded server model, extensive dependency injection capabilities, and support for RESTful APIs allowed rapid development of reliable and scalable services.

Using Spring Boot ensured that each microservice could be started independently, configured easily through properties files, and integrated seamlessly with other services in the *CodeMaster* system. Additionally, Spring Boot's extensive ecosystem facilitated integration with libraries for data persistence, security, and messaging, such as RabbitMQ, reducing development overhead and improving maintainability.

Spring Boot also provided robust support for testing, including unit and integration testing.

8.10 VitePress

A crucial role in the creation and management of the project's documentation was played by *VitePress* [32], a tool specifically designed for building high-performance static documentation sites. Its native integration with Vue.js proved to be particularly advantageous, especially for a project like *CodeMaster*, where the interface relies heavily on Vue components.

One of the most valuable features during development was undoubtedly *hot reload*: every change made to Markdown or Vue files was instantly reflected in the documentation, without requiring a manual server restart. This enabled a highly fluid workflow and significantly reduced waiting times, thereby improving overall productivity.

8.11 Mongoose

To simplify interactions with the MongoDB database, the project adopted *Mongoose* [18], one of the most widely used and reliable ODM (Object Data Modeling) libraries in the Node.js ecosystem.

Mongoose provides a structured, high-level interface for working with MongoDB, enabling the definition of strongly typed document schemas while also supporting relationships, validation, middleware, and many other features.

By relying on Mongoose, the project avoided the need to manually implement repetitive or complex tasks, such as query handling, nested data management, or format conversions. Furthermore, the ability to explicitly model data through well-defined schemas significantly improved code readability, robustness, and long-term maintainability.

In the context of a microservices-based architecture such as *CodeMaster*, Mongoose played a strategic role by standardizing database access. This not only reduced the

likelihood of errors but also increased consistency across different services interacting with MongoDB.

8.12 Cookie Parser

For handling user registration and authentication, the project employed the *cookie-parser* library [4], a simple yet highly effective tool for managing HTTP cookies in a Node.js environment.

Integrating this library streamlined access to information stored within cookies, thereby simplifying session tracking and enhancing the overall login experience. During registration or authentication, a secure cookie containing a token (such as a JWT) is generated and automatically sent to the client.

This mechanism allows users to remain authenticated across multiple sessions without re-entering their credentials each time they access the platform. In practice, the cookie serves as a bridge between sessions, making the login flow smoother and more transparent while preserving security.

Additionally, cookie-parser made server-side cookie management more intuitive. Thanks to its automatic parsing of HTTP request cookies, the relevant data became immediately accessible within Express code, greatly reducing implementation complexity.

8.13 Animate on Scroll (AOS)

To enhance the user experience and deliver a smoother, more modern, and visually appealing interface, the project integrated the *Animate on Scroll* (AOS) library [28].

AOS is a lightweight and highly configurable tool that allows developers to apply professional-grade animations to HTML elements as they enter the viewport during scrolling, without requiring complex CSS or JavaScript.

This approach helped avoid a static or rigid interface by introducing natural and fluid transitions at strategic points in the layout, such as the appearance of sections, cards, or buttons.

The use of lightweight, contextual animations made user interactions more engaging, reinforcing the perception of responsiveness and modernity in the application, without compromising performance or accessibility.

8.14 SweetAlert2

To improve communication between the system and its users, particular attention was given to the design of visual feedback mechanisms. For this purpose, the project integrated *SweetAlert2* [29], a modern and highly customizable library for building modal dialogs and toast notifications.

SweetAlert2 proved especially effective because it enables the rapid creation of informative, confirmation, or error popups with just a single line of JavaScript code, eliminating the need for additional HTML markup or complex logic.

Thanks to its intuitive syntax and high degree of configurability, the library allowed the delivery of immediate, visually engaging responses, which contributed to an overall impression of smoothness and attention to detail throughout the platform.

8.15 Docker

For managing the deployment process, the project adopted *Docker* [5], one of the most widely used and reliable containerization technologies in modern software development. This choice was deliberate, as Docker offered an elegant and scalable solution for isolating and distributing each individual microservice within the project.

Every microservice was "dockerized," meaning it was encapsulated in a dedicated image containing all the required dependencies, configurations, and runtime environments. This approach provided several advantages:

- Users and evaluators of the project were not required to manually install libraries, runtimes, or external tools for each individual part of the system.
- Each microservice could be executed independently, which simplified development, debugging, and maintenance.
- The entire ecosystem could be replicated on any machine, ensuring consistency across development, testing, and production environments.

Another key benefit of Docker is its intelligent dependency optimization. When multiple containers share similar components (e.g., Node.js versions or common packages), Docker avoids unnecessary duplication by managing images through an efficient layer-based system.

This mechanism not only reduces download times but also improves overall performance during build and deployment, making the system faster to start up and lighter to maintain.

9 Validation

9.1 Automatic Testing

Automatic testing was implemented to ensure the reliability, correctness, and maintainability of the *CodeMaster* system. The testing approach covered unit tests, integration tests, and end-to-end tests, using modern and widely adopted JavaScript/TypeScript tools.

- **Unit Testing:** Individual components and modules were tested using *Jest* [10]. Unit tests focused on verifying the behavior of each function, class, or module in isolation. Mocking was employed to isolate dependencies, ensuring that each unit behaved as expected under controlled conditions.

For Kotlin-based services, only unit tests were implemented using the *Mockk* library [16], which allowed mocking of service dependencies and verification of behavior at the module level.

  - *Rationale:* To detect logic errors, validate edge cases, and prevent regressions in individual modules.
  - *Automation:* All unit tests were automated via Jest's CLI.
  - *Execution:* Run using `jest --coverage` to generate coverage reports.
  - *Requirements:* Tests were linked to functional requirements such as user registration, authentication, and CodeQuest creation.

- **Integration and Communication Testing:** Interactions among microservices and API endpoints were tested using *Supertest* [30], simulating HTTP requests and verifying responses. These tests ensured correct integration between frontend and backend, as well as proper behavior of dependent services like RabbitMQ.

  Only for TypeScript-based services, *Mongo Memory Server* was used to provide an in-memory MongoDB instance for testing purposes. This approach eliminated the need to start a full MongoDB server for every test, significantly improving test execution speed and preventing resource overload in continuous integration pipelines on GitHub.

  - *Rationale:* To validate that communication among components works as expected in a microservices architecture.
  - *Automation:* Executed automatically within Jest test suites.
  - *Execution:* Included in the same Jest workflow as unit tests.
  - *Requirements:* Related to data exchange, API correctness, and session management.

- **End-to-End Testing:** The full system was tested in a *production-like* environment using Docker and docker-compose [5, 6]. Running `docker-compose up` launches all services in the correct order and network configuration, allowing end-to-end verification of the entire platform, including user flows from login to CodeQuest submission.

  - *Rationale:* To confirm that all components function together as intended, and to detect potential crashes or errors in complex workflows.
  - *Automation:* The deployment itself acts as a test harness in a production-like environment.
  - *Execution:* Launch the complete stack with `docker-compose up` and run automated scripts or manual verification as needed.
  - *Requirements:* Ensures compliance with functional requirements for user authentication, code submission, and scoring.

Recall that *deployment* **automation** is commonly used to *test* the system in a *production-like* environment.

Tests include corner cases, such as crashes, errors, or invalid inputs, to guarantee robust behavior.

9.2 Acceptance Testing

As part of the manual acceptance testing, were evaluated the system's behavior under conditions where some microservices were unavailable. This type of scenario cannot be fully tested through automated tests, as it involves simulating faults and partial service outages in a controlled manner while observing user-facing behavior.

During these tests, we deliberately disabled specific services and verified that *CodeMaster* responded according to the designed fault-tolerance mechanisms. The main objective was to ensure that the platform remained usable, that data consistency and integrity were preserved, and that users could continue their activities without encountering unexpected errors or inconsistencies.

For example, if a dependent service was temporarily down, the system gracefully handled requests by providing appropriate feedback to the user, queuing actions where necessary, and maintaining the integrity of ongoing workflows. This approach confirmed that *CodeMaster* is resilient to partial failures, and that users can safely interact with the platform even when certain services are unavailable.

In addition, were performed other tests to ensure usability, visual accessibility, and user experience, focused on user interface elements, accessibility, and responsive design. In particular was tested visual contrast, responsive layout, font readability, accessibility attributes (`alt` text for images), and correct rendering across devices. Visual aspects and accessibility require human judgment to assess legibility, clarity, and user-friendliness. Automated tools such as *Responsively App* [27] and *WebAIM Contrast Checker* [34] assisted but could not fully replace human evaluation.

10 DevOps

The build system encompasses the collection of tools and procedures that automate the process of compiling, testing, and packaging the project. It ensures that the software can be consistently built and deployed across different environments, reducing manual effort and potential errors.For this project, Gradle was chosen as the build system. Gradle offers a flexible and efficient way to manage dependencies, define build tasks, and streamline the overall development workflow, making it well-suited for projects that require scalability and maintainability.

10.1 Project Structure

Given the decision to structure the system according to a microservices architecture, we opted for a mono-repository and multi-project setup. This approach allows all com-

ponents of the system to be maintained within a single repository, simplifying code management, version control, and continuous integration, while still preserving the independence and modularity of each microservice. The subprojects that make up the overall architecture are as follows:

- **authentication-service**: based on node, contains the authentication service;
- **user-service**: based on node, contains the user service;
- **codequest-service**: based on node, contains the codequest service;
- **community-service**: based on node, contains the community service;
- **solution-service**: based on jvm, contains the solution service;
- **codegenerator-service**: based on jvm, contains the codegenerator service;
- **frontend-service**: based on node, contains the frontend service;
- **api-gateway**: based on nginx, contains the server for microservice.

10.2 Dependency

Each microservice manages its own dependencies independently. A deliberate architectural decision was made to ensure that every microservice remains fully self-contained, even when multiple services rely on the same external libraries or components.While this approach introduces a small overhead since common dependencies need to be downloaded separately, it ensures complete isolation between microservices. As a result, updates or changes to one service do not directly affect the others, thereby improving the overall stability, maintainability, and scalability of the system.Each microservice, designed to be fully isolated and independent, includes its own `build.gradle.kts` file, which specifies the set of dependencies required for its operation. This approach allows every microservice to autonomously manage its configuration and dependencies, preserving flexibility and independence from the rest of the system.Dependency management also varies according to the underlying technology: for microservices developed in Node.js, the npm package manager was adopted, whereas Spring was chosen for those running on the JVM, leveraging its robust ecosystem for dependency management and application lifecycle configuration.This technological diversity makes it possible to exploit the strengths of each platform while maintaining consistency and standardization across the overall build process.

10.2.1 Node Plugin

Some of the microservices are built using Node.js, which introduces specific requirements such as downloading modules or executing script commands.To handle these needs consistently within the overall build infrastructure, the Node Gradle plugin was adopted, available at the following link.This integration made it possible to define custom tasks

for each microservice, tailored to its individual development needs. As a result, dependencies can be installed automatically, build scripts can be executed, and tests can be run directly through Gradle, ensuring a coherent and centralized management of the entire build process.

10.2.2 Docker Plugin

Another plugin integrated into the project, specifically used by the microservice responsible for generating containers in which user-submitted code is executed, is the Docker plugin for Gradle. This tool allows the management of the entire Docker container lifecycle directly from the build system, including image creation, configuration, and execution.Through this integration, the microservice can dynamically create and launch isolated containers to safely run the code produced or uploaded by users, ensuring a consistent, secure, and reproducible execution environment. The plugin is available at the following link.

10.3 Version Control

The Version Control System (VCS) is a fundamental tool that tracks and manages changes made to files throughout the lifecycle of a project. It maintains a detailed history of modifications, allowing developers to restore previous versions of individual files or even revert the entire project to an earlier state when needed.Beyond simple version tracking, a VCS provides powerful collaboration features: it enables team members to work on different parts of the codebase simultaneously, compare revisions, and identify when and by whom specific changes were introduced. This facilitates debugging, improves accountability, and enhances the overall transparency of the development process.For this project, Git was chosen as the Version Control System. Git's distributed architecture, branching model, and integration with modern development platforms make it an ideal choice for managing collaborative, large-scale, and continuously evolving software projects.

10.3.1 Semantic Versioning

The project uses Semantic Versioning for software release versioning.

10.3.2 Conventional Commits

The project uses Conventional Commits for commit messages.
To ensure consistency and clarity in the commit history, the project enforces the Conventional Commits standard. This convention defines a structured format for commit messages, making it easier to understand the nature of changes, generate changelogs automatically, and maintain a clean and meaningful version history.To automate this enforcement, a Git hook was implemented that validates each commit message before it is accepted. This hook ensures that every commit follows the predefined format

and prevents non-compliant commits from being added to the repository.For the automatic installation and management of this hook, the project integrates the Gradle plugin `org.danilopianini.gradle-pre-commit-git-hooks`, configured as follows:

```
plugins {
    id("org.danilopianini.gradle-pre-commit-git-hooks") version "2.0.25"
    id("org.gradle.toolchains.foojay-resolver-convention") version "1.0.0"
}

gitHooks {
    commitMsg { conventionalCommits() }
    preCommit { tasks("npmTest", "test") }
    createHooks(true)
}
```

10.3.3 Semantic Release

To streamline and automate the release workflow, the project integrates the Semantic Release plugin. This tool automatically determines the next version number, generates a changelog, and publishes a new release whenever a commit is pushed to the main branch.Semantic Release analyzes commit messages to infer the type and scope of changes introduced into the codebase. By following the Conventional Commits standard, the system can automatically decide whether the update represents a patch, a minor enhancement, or a major change, and assign the appropriate semantic version accordingly.This approach eliminates the need for manual version management and minimizes human error. The main branch thus becomes the single source of truth for the project, ensuring that every release is consistent, traceable, and accurately versioned. Furthermore, this level of automation promotes a smoother continuous integration and delivery (CI/CD) pipeline, increasing both development efficiency and release reliability.

10.4 Repository Management

The project uses the branch `main` as default branch, where semantic versioning and release are applied.
Other main types of branches used in the project are:

- `feature/<feature-name>`: branch for a new feature development.
- `fix/<microservice-name>`: branch for a bug fix.
- `refactor/<microservice-name>`: branch for refactor code in a service.
- `hotfix/<microservice-name>`: branch for immediatly bug fixing.
- `ci/<feature-name>`: branch for add feature in ci.
- `docker/<microservice-name>`: branch for dockerization.

10.4.1 Merging

The team decided to adopt different integration strategies for the project's branches to balance historical traceability and clean development history. For the `main` branch, merge commits were chosen to preserve a complete record of how features and fixes were integrated over time, providing a transparent and detailed history of the project's evolution. Conversely, the `develop` branch uses the rebase technique to maintain a linear and readable commit history, making it easier to follow the progression of changes and reducing unnecessary merge clutter. This approach combines the advantages of both methods: a clean and comprehensible development timeline, along with a fully traceable integration history on the main branch.

10.4.2 Pull Requests

The project relies on GitHub Pull Requests as the primary mechanism for merging changes into the `main` branch, as well as for integrating updates from feature branches back into `develop`. This strategy ensures a controlled and transparent integration process, where every change undergoes review before being incorporated into the main codebase. By using Pull Requests, each team member stays informed about ongoing developments, can discuss improvements, and help identify potential issues early in the process. Furthermore, each Pull Request requires a code review from at least one other contributor, ensuring that all technical decisions are collectively reviewed and agreed upon. This workflow promotes collaboration, enhances code quality, and minimizes the risk of introducing regressions or inconsistencies into the project's primary branches.

10.5 Quality Assurance

Quality Assurance (QA) represents a fundamental aspect of the software development lifecycle, aimed at ensuring that the final product is reliable, maintainable, and aligned with the established requirements. It is a systematic and continuous process that involves verifying whether the software behaves as expected, adheres to quality standards, and remains free from defects that could impact functionality or user experience. By integrating QA practices throughout the development process, potential issues can be detected and resolved early, significantly reducing the risk of defects reaching the end users. This proactive approach not only improves software stability but also increases team confidence in each release.
In this project, the Quality Assurance (QA) process focuses on the following key areas:

- **Linting & Formatting**: to validate functionality and enforce consistent coding standards;
- **Code Coverage**: to measure how thoroughly the tests exercise the codebase and identify untested areas;
- **Documentation**: to ensure that every component of the system is well-documented, up-to-date, and accessible to all contributors.

Together, these practices establish a solid foundation for delivering a high-quality, maintainable, and trustworthy software product.

10.5.1 Linting & Formatting

The project leverages a variety of tools and technologies to ensure code quality through comprehensive testing and linting practices. Specifically, for code formatting and linting, the Node.js-based microservices adopt ESLint and Prettier.

10.5.2 Code Coverage

Code coverage is a key quality metric that quantifies how much of the codebase is executed during the automated testing process. It provides valuable insight into the effectiveness of the test suite by identifying untested sections of the code that might still contain hidden bugs or unverified logic. For this project, a minimum code coverage threshold of 70% was established. This benchmark ensures that a significant portion of the application's logic is validated through tests, without imposing excessive constraints during early development stages. Code coverage is evaluated separately for each subproject using Jest for Node and Spring for Kotlin, which automatically runs the defined test suites within each package and generates detailed coverage reports. To centralize and visualize these metrics, the project integrates Codecov, a platform that tracks the overall coverage statistics and publishes the results. This provides the team with continuous visibility into testing progress and helps maintain consistent quality standards across all components.
The following section presents the code coverage report generated for the project.

10.5.3 Documentation

Code documentation is a fundamental component of the software development process. It serves as a bridge between the written code and the developers who interact with it, providing clear explanations of the system's structure, components, and intended behavior. Good documentation not only helps new contributors quickly understand how different parts of the codebase work and how to use them, but it also improves long-term maintainability, facilitates debugging, and supports knowledge sharing within the team. In essence, well-crafted documentation transforms the code from something that merely works into something that is understandable, reusable, and sustainable over time. The project makes use of OpenAPI, a widely adopted standard for designing and documenting APIs. By following this specification, the development team ensures a consistent and easily understandable structure for all API definitions, improving interoperability between microservices and simplifying both maintenance and external integration. Using OpenAPI also allows the API specifications to be expressed in a clear and formal way, enabling the automatic generation of documentation and even client or server stubs when needed. All APIs within the project adhere to the CRUD (Create, Read, Update, Delete) standard, ensuring a uniform and predictable approach to resource management across the entire system.

10.5.4 Quality Control

The quality control process, that ensures the rules described above are followed, is performed at least in two phases:

- **Pre-Commit**: The quality control process takes place before any code is committed to the repository, ensuring that potential issues are detected as early as possible in the development workflow. This process involves a lightweight yet effective set of checks, including code formatting, static code analysis, and compilation validation, all aimed at maintaining a consistent and error-free codebase.By running these verifications prior to committing, developers can identify and correct minor issues immediately, reducing the likelihood of introducing technical debt or breaking changes into the shared repository.In agreement with the team, the pre-commit phase also includes the execution of automated tests. Although this approach slightly slows down the commit process, it significantly enhances the reliability, robustness, and overall stability of the project, ensuring that only well-tested and verified code is integrated into the main development branches.
- **CI/CD Pipeline**: The comprehensive quality control process is executed as part of the project's CI/CD (Continuous Integration and Continuous Deployment) pipeline. This automated workflow encompasses all the previously mentioned verification steps, including code formatting, static code analysis, unit testing, integration testing, and code coverage evaluation. Each of these stages plays a crucial role in maintaining a reliable, maintainable, and high-quality codebase. The pipeline is automatically triggered on every push to the repository and on every pull request, ensuring that any new contribution is thoroughly validated before being integrated. If any of the checks fail, the pipeline itself will also fail, effectively preventing non-compliant code from being merged. This mechanism enforces the project's quality standards by requiring that all checks must pass successfully before the code is merged into the main branch. By automating the entire verification process, the CI/CD pipeline not only safeguards code quality but also enhances team productivity, reduces manual intervention, and fosters a culture of continuous improvement throughout the development lifecycle.

10.6 CI/CD

Continuous Integration (CI) and Continuous Deployment (CD) are fundamental pillars of the DevOps software development lifecycle, enabling teams to deliver software more efficiently, reliably, and consistently. During the Continuous Integration phase, the code is automatically built, tested, and analyzed each time a developer pushes new changes to the repository. This automation ensures that issues such as syntax errors, failed tests, or integration conflicts are detected early in the development process, keeping the codebase stable and always in a deployable state. By integrating testing and analysis into every commit, the CI process promotes collaboration, reduces integration risks, and accelerates feedback loops among developers. Once the code passes all quality checks,

the Continuous Deployment phase takes over, handling the automated release and deployment steps needed to deliver new versions of the software safely and efficiently. To implement this workflow, the project leverages GitHub Actions, a powerful and flexible automation tool built directly into the GitHub platform. GitHub Actions allows the team to define and orchestrate the entire CI/CD pipeline within the repository itself, automating tasks such as building, testing, deploying, and even notifying developers, all in a consistent and reproducible manner.

10.6.1 Action

The file **.github/workflows/CI-CD.yml** contains the configuration for the entire CI/CD process. This action performs build, test, release, delivery and deployment tasks.

10.6.2 Build And Test

Conceptually, the first task in the pipeline, named `build-and-test`, is responsible for building the project and running the complete test suite. This job is executed across all branches of the repository and employs a matrix strategy that runs tests on multiple operating systems: Ubuntu, macOS, and Windows (all in their latest versions). Such an approach ensures that the application remains consistent and functional across different environments, thereby improving both its portability and overall reliability. When the job is triggered on the `main` branch, the task additionally uploads the code coverage reports to Codecov, the platform mentioned earlier. This integration enables continuous tracking of test coverage and helps maintain high standards of code quality throughout the development lifecycle.

10.6.3 Release

The second job executed in the pipeline is responsible for automatically generating a new project release, managed through the Semantic Release bot. This job runs only if the previous `build-and-test` job completes successfully and the workflow is triggered on the `main` branch. At this stage, Semantic Release analyzes the latest commit messages to determine the type of changes introduced, such as fixes, new features, or breaking changes and uses this information to automatically compute the next semantic version number. It then creates a new tag in the repository and publishes the corresponding release, including an updated changelog that reflects the new modifications. This process fully automates the release cycle, removing the need for manual intervention, minimizing the risk of human error, and ensuring that every software version is generated in a consistent, transparent, and traceable manner.

10.6.4 Delivery Docker Image

Once the release job completes successfully and the workflow is running on the `main` branch, the pipeline proceeds with the automatic upload of Docker images for each microservice in the project. These images are built and published to Docker Hub, the

platform used for container distribution, ensuring that every service is readily available for deployment or testing across different environments. Each image is tagged with the current version number, which is automatically generated by the Semantic Release Bot during the previous stage of the pipeline. This practice provides full traceability between the source code, the released version, and the corresponding Docker image, simplifying release management and ensuring consistency across all runtime environments. As a result, the release process becomes fully automated and reproducible, minimizing manual effort and supporting a more efficient and reliable DevOps workflow.

10.6.5 Deploy Docs

The final stage of the pipeline is dedicated to the automatic publication of the API documentation, generated through OpenAPI and visualized using Swagger. If the previous release job completes successfully and the workflow runs on the `main` branch, the pipeline proceeds to generate and publish the latest version of the documentation via GitHub Pages. This ensures that the API documentation is automatically updated with every new release, accurately reflecting any changes introduced in the microservices' interfaces. As a result, developers and stakeholders always have access to an up-to-date, consistent, and easily accessible version of the project's API documentation. This approach minimizes manual maintenance effort, improves transparency, and strengthens communication across the entire development team. The latest version of the documentation is available at the following link: OpenAPI Swagger Documentation.

10.6.6 Renovate Bot

Renovate is an automated dependency management tool designed to help development teams keep their projects up to date and secure. It continuously scans the codebase for outdated libraries, frameworks, or packages and suggests or applies updates as needed. In this project, Renovate is configured to automatically create pull requests whenever a dependency update is available. These pull requests clearly indicate what has changed, allowing developers to review and validate the updates before merging. When certain conditions are met, such as passing all automated tests and compliance with predefined rules. Renovate can also automatically merge the pull request, further streamlining the maintenance process. We chose to use a custom configuration, accessible via the **renovate.json** file.

11 Deployment

The deployment phase of *CodeMaster* was carefully designed to ensure that the system could be installed and run from scratch with minimal effort, even on machines not previously configured for development. Given that the application consists of multiple microservices, each with its own dependencies and runtime requirements, a reliable and reproducible deployment process was essential.

To install the system, the user must first ensure that *Docker* and *Docker Compose* are installed on their machine. No additional dependencies or manual setup of databases, message brokers, or runtime environments is required. Once Docker is installed, the deployment process consists of the following steps:

1. Clone the project repository from the official source.
2. Navigate to the root directory of the project.
3. Execute the following command to build and start all services:

   Listing 1: Gradle task to execute project

   ```
   .\gradlew dockerCompose
   ```

   The choice of a Gradle command, rather than directly invoking `docker-compose up`, was made to ensure that several preliminary tasks are executed beforehand, such as building the container image responsible for running the code.
4. Once the containers are running, access the frontend using any web browser at:

   Listing 2: Project URL

   ```
   http://localhost:5173
   ```

Following these steps, Docker automatically handles all tasks necessary to run the system:

- It builds Docker images for each microservice, including the backend, frontend, RabbitMQ message broker, and database.
- It creates and launches containers in the correct order, respecting dependencies between services.
- It establishes a shared network to ensure interoperability between services.

The system becomes immediately operational, and users can interact with the platform without manually installing or configuring individual components. Thanks to this setup, *CodeMaster* achieves high portability and reproducibility: the same deployment process can be executed on development, testing, staging, or production environments with identical results.

11.1 Additional Notes

Each microservice image is also published on *Docker Hub* [6], allowing users to pull prebuilt images and skip the local build step if desired. This further simplifies deployment, particularly for users who are not familiar with the development environment or who wish to quickly run the system without managing dependencies manually.

Overall, this deployment strategy ensures that *CodeMaster* is easy to install, immediately testable, and ready for use in a professional and distributed setup.

12 User Guide

To run *CodeMaster*, the first step is to clone the GitHub repository. Before proceeding, make sure the following dependencies are installed and running:

- Docker Desktop
- MongoDB

Once the environment is ready, navigate to the project's root directory and execute the following command:

Listing 3: Project startup with Gradle task

```
./gradlew dockerCompose
```

After the system has started successfully, simply open any web browser and visit `http://localhost:5173`. On the initial access, the system ask the user to either sign in or sign up, depending on whether they already have an account.

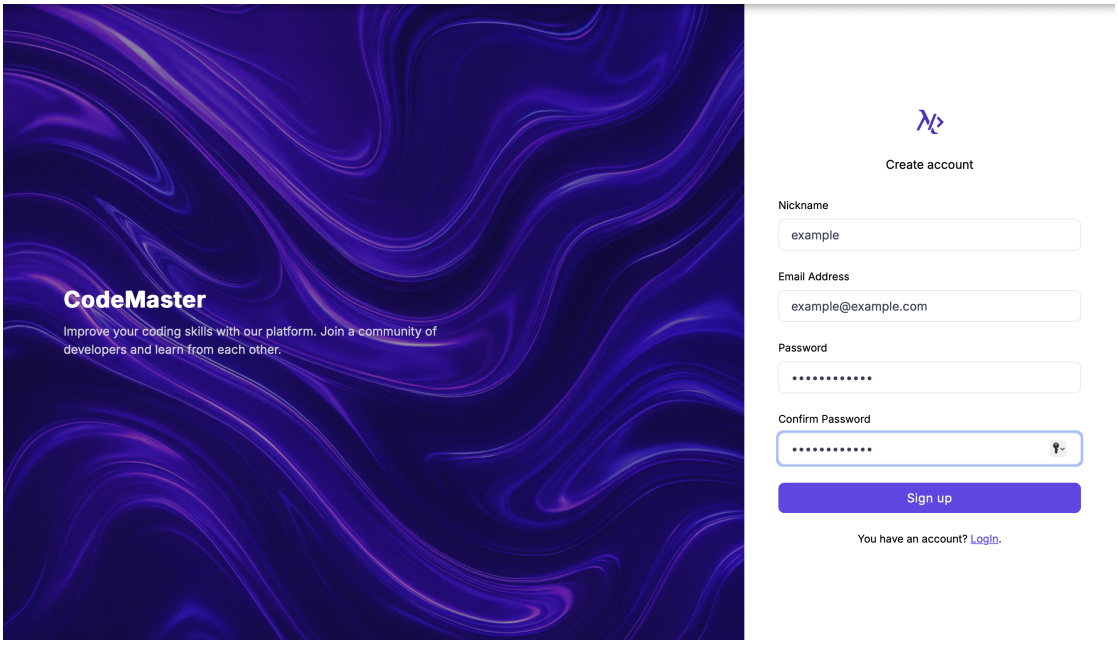

Figure 17: SignUp page

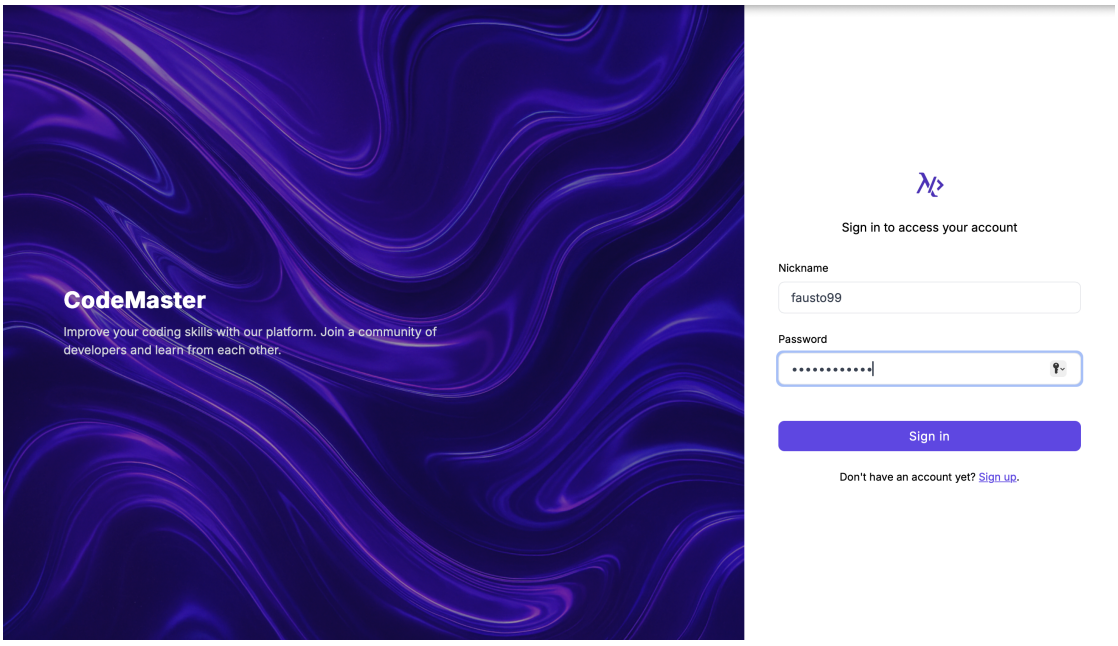

Figure 18: SignIn page

Once authentication is successfully completed, the user is redirected to the **dashboard**, which serves as the central access point for the platform. From here, the user can navigate to:

1. Their personal profile
2. Their own codequests as well as those created by other users

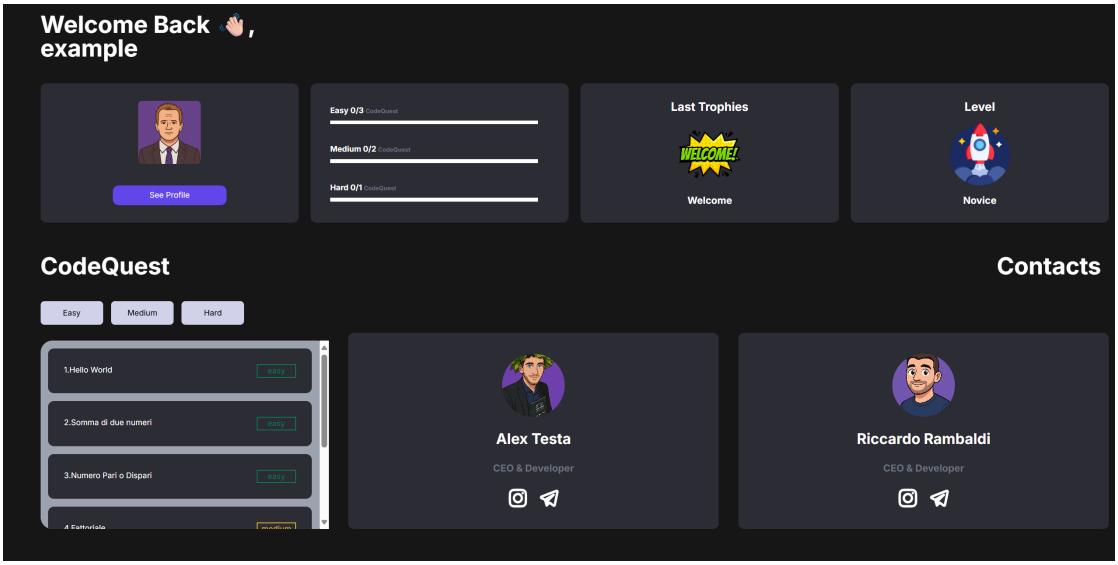

Figure 19: Dashboard page

Profile Page

From the profile page, users can update and personalize their information, such as their biography, preferred programming languages, and CV.

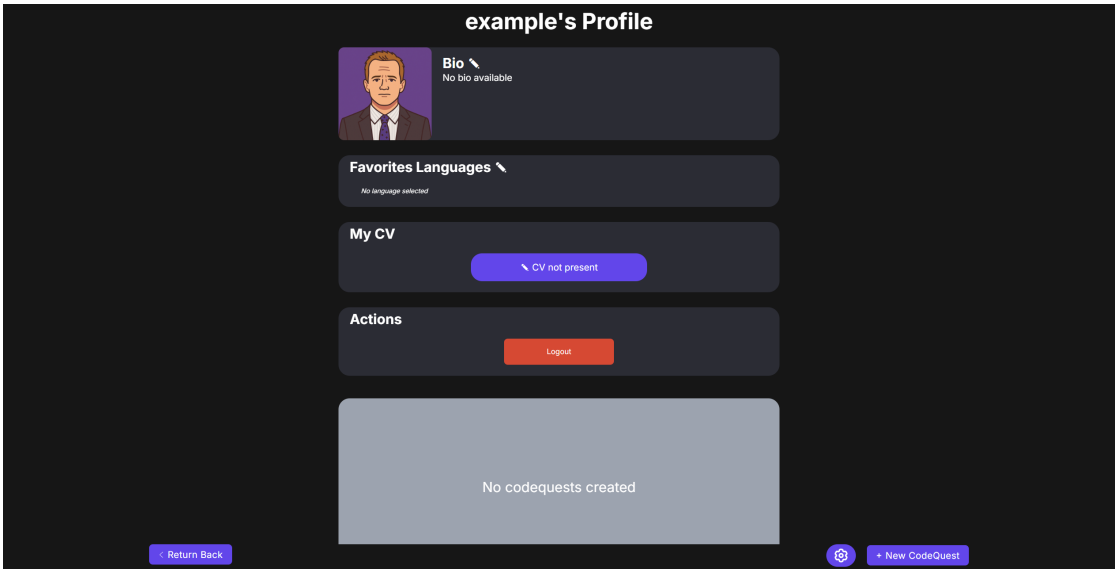

Figure 20: Profile

In addition, the profile page provides access to key actions such as signing out, opening the settings, creating a new codequest, or managing existing ones (viewing or deleting).

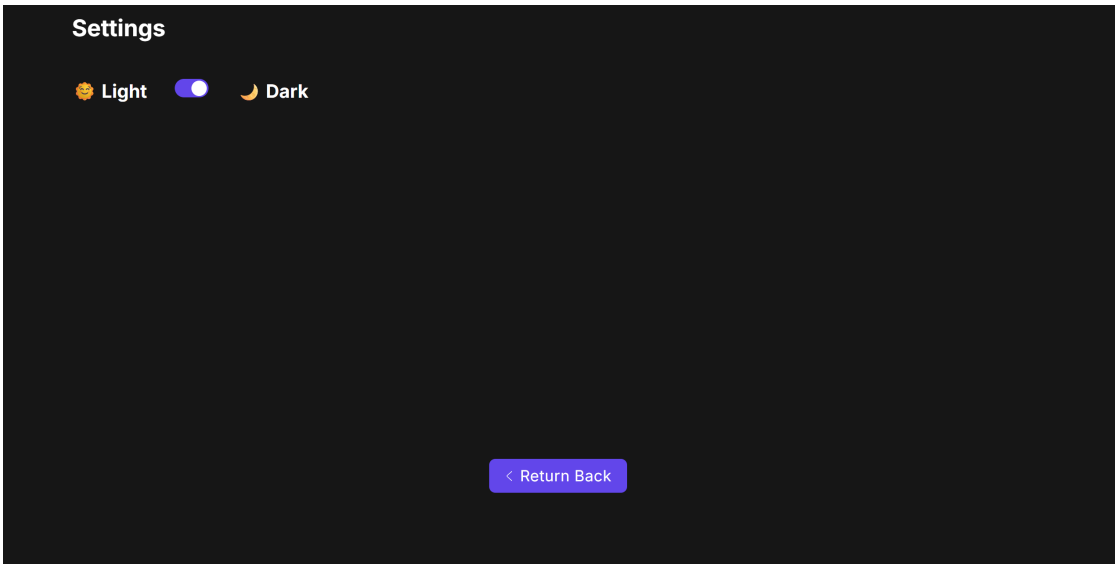

Figure 21: Settings page

When creating a new codequest, the user is redirected to a dedicated creation page where several details must be provided, including:

- Title
- Description
- Constraints (restrictions or hints)
- Supported programming languages
- Difficulty level
- Function name
- Number of parameters, along with their names and types
- Return type of the function

These specifications, particularly the function signature, are essential for generating the function template that will be used to solve the codequest.

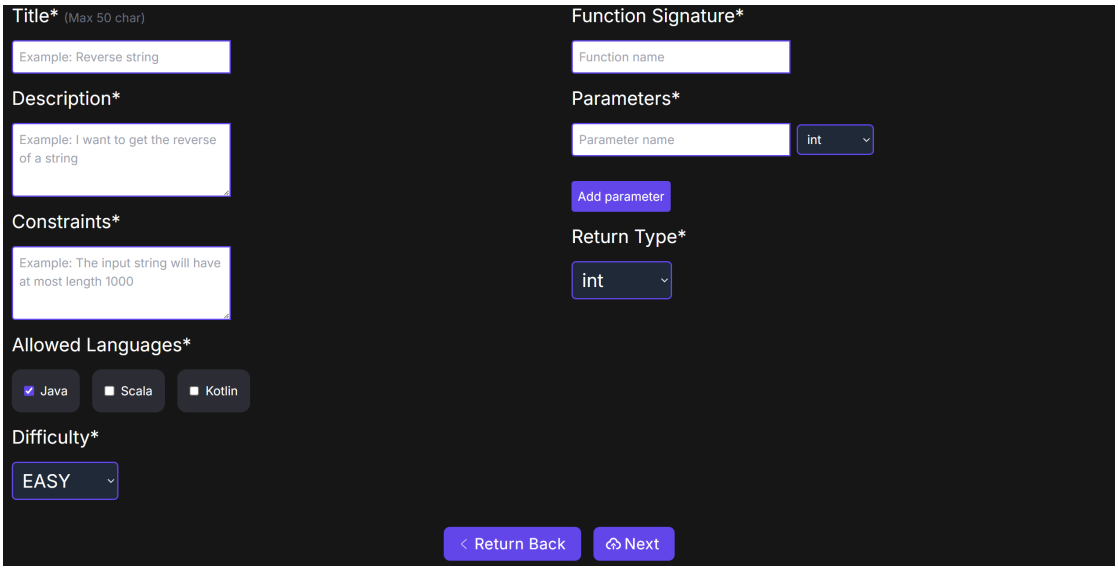

Figure 22: Codequest creation page

Once all fields are completed, the user proceeds to the **examples section**. Here, they can provide a variable number of input–output examples, which will be used to automatically generate the test cases executed against the submitted function.

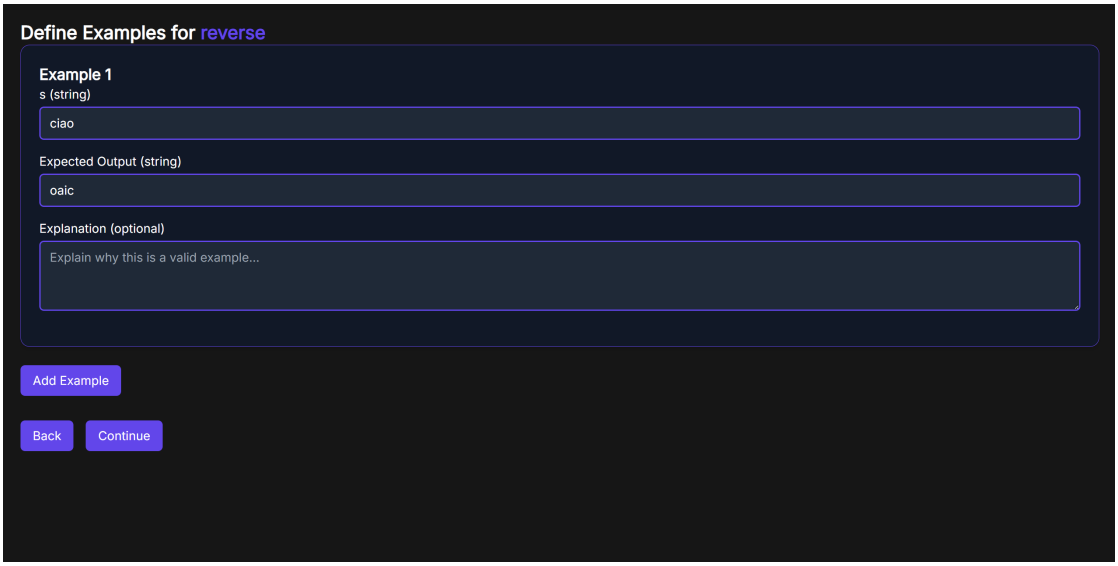

Figure 23: Examples page

After completed all required fields, The user can proceed and, if everything has been completed successfully, they are redirected to their profile page.

Codequest

From the dashboard, the user can browse all available codequests (both their own and those published by other users) and select one to open. Accessing a specific codequest displays a dedicated page containing the challenge description along with a built-in code editor pre-loaded with the function template.

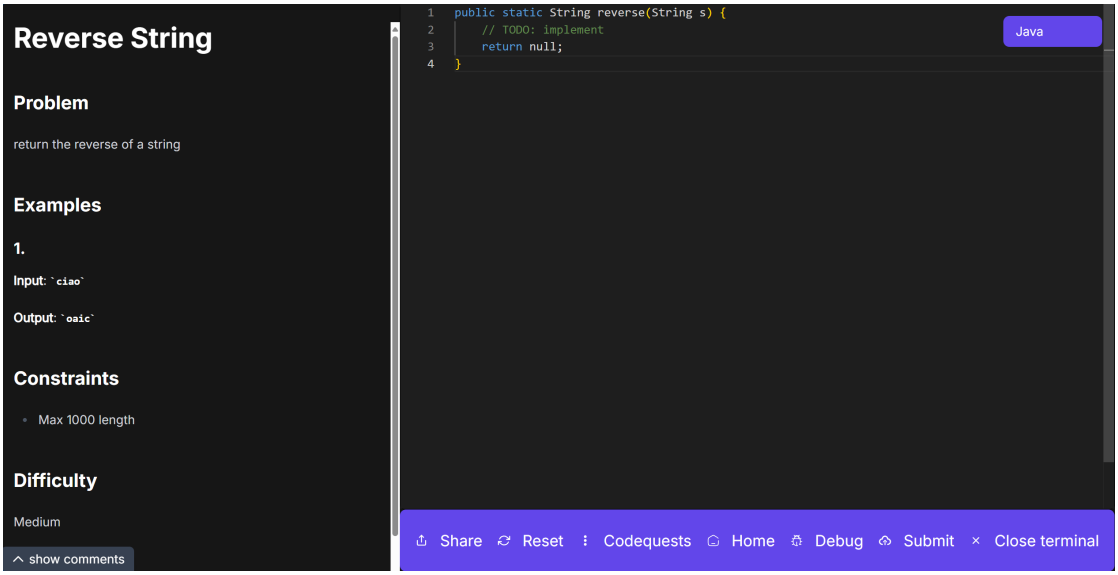

Figure 24: Codequest page

Within this environment, the user has several options:

- Edit the function, debug it (via the *debug* button), or run tests (via the *submit* button)
- Browse a list of other codequests and quickly navigate to them (via the *codequest* button)
- Switch between different programming languages
- Reset the code editor
- Share the codequest
- Open the comments section

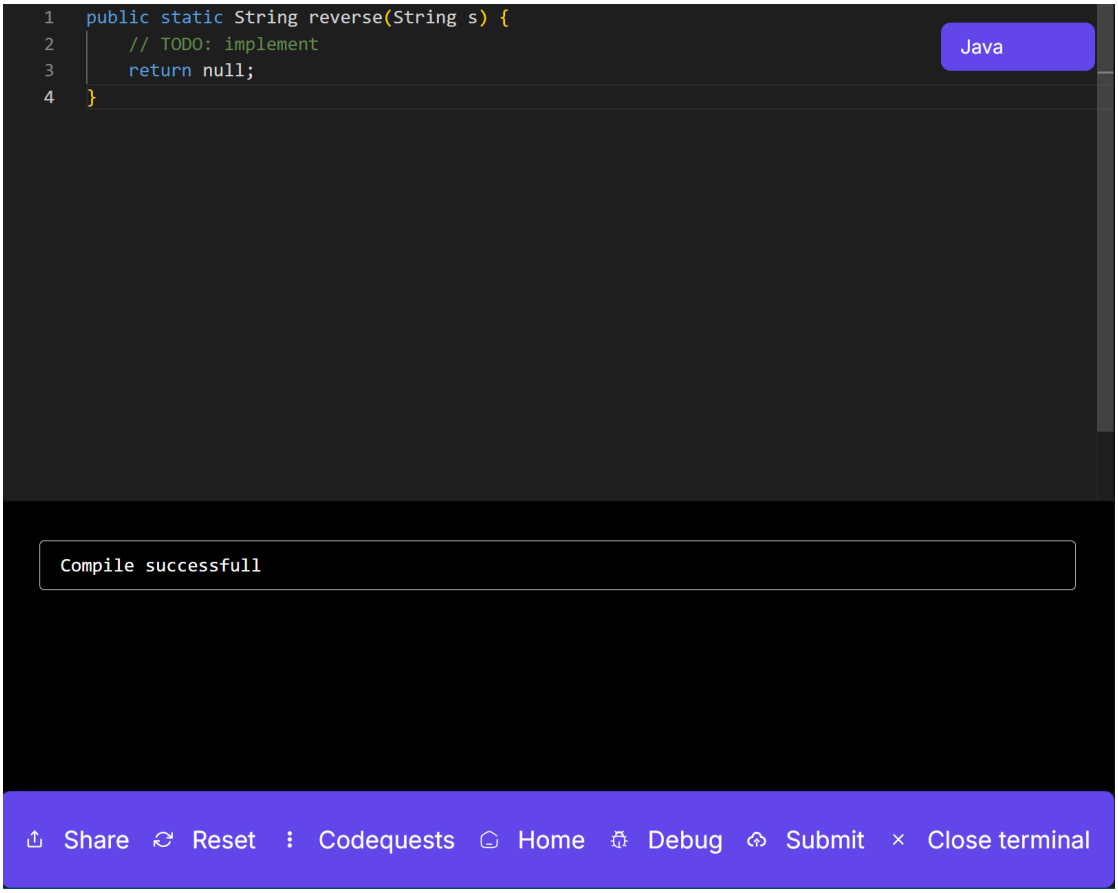

Figure 25: Submit section

If there are no compilation errors, the debug will show "Compilation Successfull", otherwise the terminal show a detailed error stack trace.

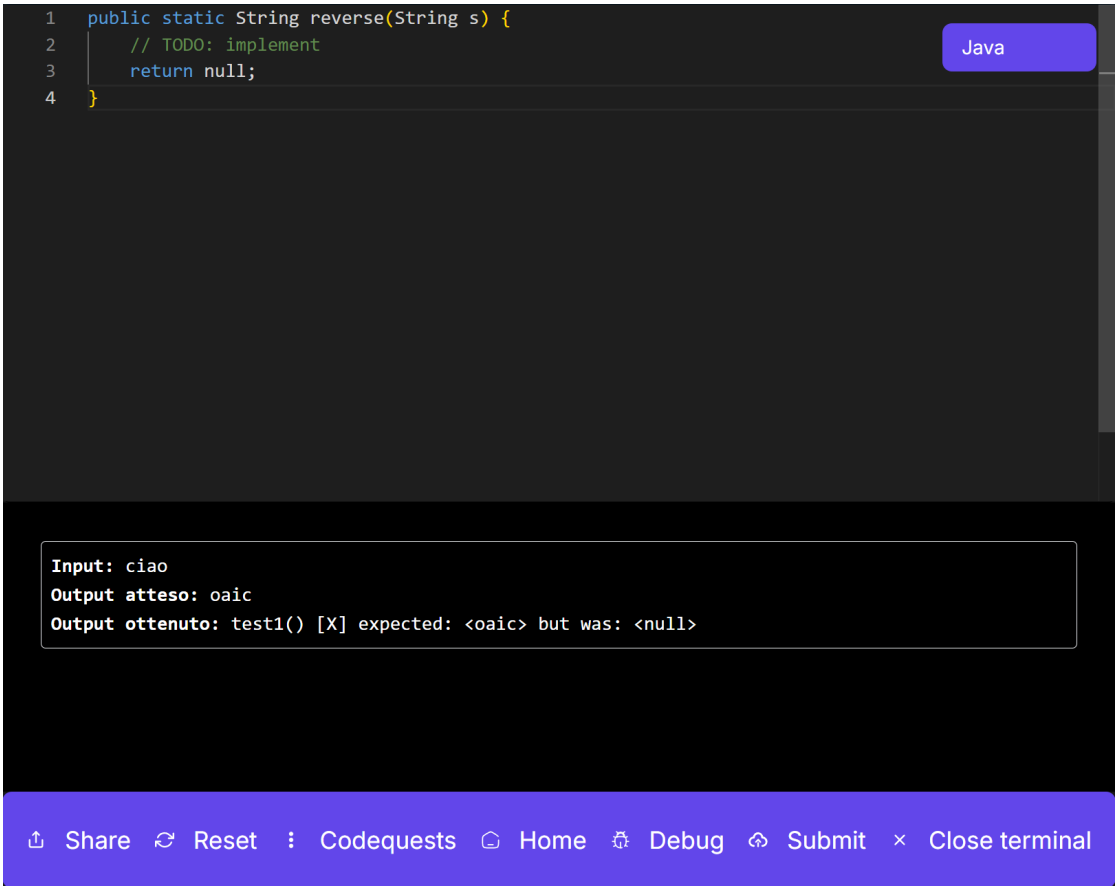

Figure 26: Submit section

Same thing goes for tests. Plus, if tests are not successful but there is no compilation error, the terminal show each actual and expected output for every test that was executed.

Figure 27: Sidebar with navigation options

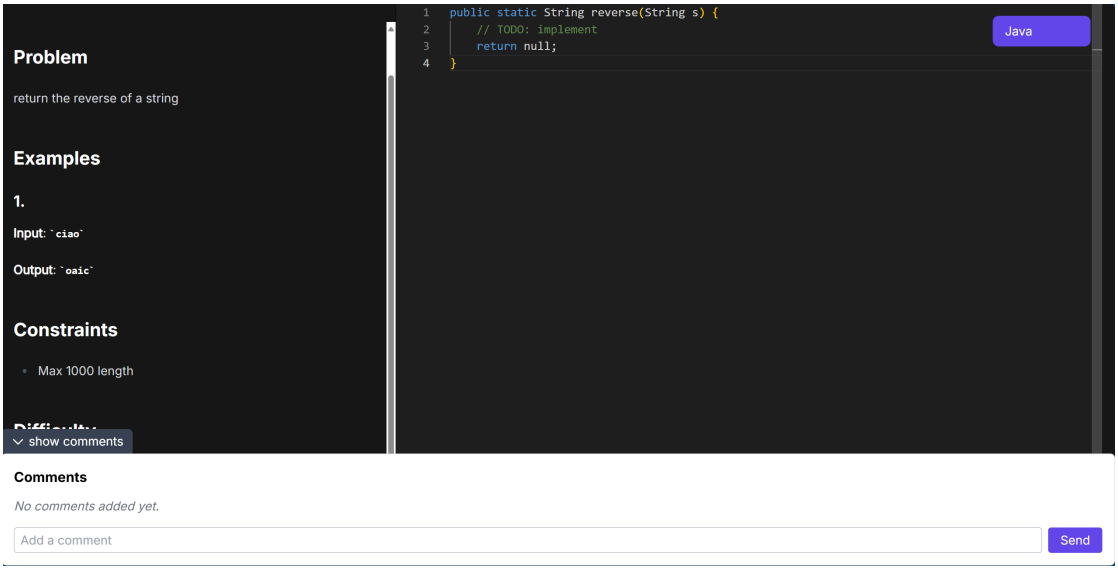

Figure 28: Comments section

If the user is an admin, the access is the same as all other users. Once an admin is authenticated, he can access the admin page.

Figure 29: Admin section

From this page, an admin can ban/unban users, or delete Codequests that violate the guideline rules.

13 Self-evaluation

- An individual section is required for each member of the group
- Each member must self-evaluate their work, listing the strengths and weaknesses of the product
- Each member must describe their role within the group as objectively as possible.

It should be noted that each student is only responsible for their own section

References

[1] Artur Bosch et al. *Detekt – Static code analysis for Kotlin*, 2025. Accessed: 2025-09-28.

[2] Gavin Bierman, Martin Abadi, and Mads Torgersen. Understanding typescript: A language for javascript application scale. In *Microsoft Research Technical Report*, 2014.

[3] Giulio Canti. *fp-ts: Functional Programming in TypeScript*, 2017. Accessed: 2025-09-26.

[4] cookie parser. *cookie-parser: Parse Cookie Header and Populate req.cookies*, 2014. Accessed: 2025-09-26.

[5] Docker Inc. *Docker: Empowering App Development for Developers*, 2013. Accessed: 2025-09-26.

[6] Docker Inc. *Docker Hub – The world's largest library and community for container images*, 2025. Accessed: 2025-09-26.

[7] ESLint. *ESLint: Pluggable JavaScript Linter*, 2013. Accessed: 2025-09-26.

[8] Express.js. *Express.js Guide*, 2025.

[9] Helmet.js. *Helmet: Help Secure Express Apps with HTTP Headers*, 2013. Accessed: 2025-09-26.

[10] Jest. *Jest: Delightful JavaScript Testing Framework*, 2014. Accessed: 2025-09-26.

[11] JetBrains. *Kotlin Documentation*, 2025.

[12] JetBrains. *Kover – Kotlin code coverage Gradle plugin*, 2025. Accessed: 2025-09-28.

[13] JWT.io. *JSON Web Tokens (JWT)*, 2025. Available at: `https://jwt.io`, Accessed: 2025-09-26.

[14] Tailwind Labs. *Tailwind CSS: Rapidly Build Modern Websites Without Ever Leaving Your HTML*, 2017. Accessed: 2025-09-26.

[15] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.

[16] Mockk Contributors. *Mockk – A modern mocking library for Kotlin*, 2025. Accessed: 2025-09-28.

[17] MongoDB. *MongoDB Manual*, 2025.

[18] Mongoose Authors. *Mongoose - elegant MongoDB object modeling for Node.js*. Mongoose, 2025. `https://mongoosejs.com/`.

[19] Sam Newman. *Building Microservices*. O'Reilly Media, 2015.

[20] NGINX, Inc. *Nginx Documentation*, 2025.

[21] Node.js. *Node.js Documentation*, 2025.

[22] npmjs.com. *bcrypt - a library to help you hash passwords*, 2025. Available at: `https://www.npmjs.com/package/bcrypt`, Accessed: 2025-09-26.

[23] Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.

[24] Pivotal Software. *Spring Boot – Framework for building production-ready applications in Java and Kotlin*, 2025. Accessed: 2025-09-28.

[25] Pivotal Software, Inc. *RabbitMQ Documentation*, 2025.

[26] Prettier. *Prettier: Opinionated Code Formatter*, 2017. Accessed: 2025-09-26.

[27] Responsively App Contributors. *Responsively App – A modified browser for responsive web development*, 2025. Accessed: 2025-09-28.

[28] Michal Snik. *AOS: Animate On Scroll Library*, 2015. Accessed: 2025-09-26.

[29] SweetAlert2 Contributors. *SweetAlert2: Beautiful, Responsive, Customizable Replacement for JavaScript's Popup Boxes*, 2015. Accessed: 2025-09-26.

[30] TJ Holowaychuk et al. *Supertest – HTTP assertions for testing Node.js APIs*, 2025. Accessed: 2025-09-28.

[31] TypeScript Team. *TypeScript: JavaScript that scales*, 2025.

[32] VitePress. *VitePress: Static Site Generator Powered by Vite*, 2020. Accessed: 2025-09-26.

[33] Vue.js. *Vue.js Official Documentation*, 2025.

[34] WebAIM. *Contrast Checker – WebAIM*, 2025. Accessed: 2025-09-28.

[35] Wikipedia contributors. *Microservices*, 2025.