Project documentation for the portfolio exam

Tim Knüttel (5123101) Adrian Stelter (5123031)
Laurin Neubert (6824016) Jakob Hämmelmann (5123113)

February 15, 2025

1 Introduction of the Project Topic

1.1 Domain

The domain of our project is a social media platform like X (formerly Twitter), Threads or Bluesky. Our goal for the project is to create a platform that enables users to interact with one another through means of text posts. The main objective here is to give users the opportunity to interact not only by posting their own posts, but also to give feedback through likes and comments. Tags help the users to find posts based on interests or topics.

1.2 Use Cases

1.2.1 Posts

The central and most important use case for our project is the ability for users to post text posts. When posting, users should be able to fill their post with content, also it should be visible which user posted at what exact time.

1.2.2 Replies & Likes

Users should be able to interact with posts directly by replying to and "liking" them. This gives users the ability to converse on the platform and discuss their favorite topics. Additionally, replying keeps all the different posts together, so there can be an easy understanding of which post is a new topic and which post was written in response to an already existing topic.

1.2.3 Tags

Tags can be used to link a topic to the created post. By searching for posts tagged with the desired topic, users can easily find posts that are relevant to what they wish to converse about. This is designed to increase the usability and also user retention.

2 Description of the Software Architecture

The hexagonal architecture is generally divided into two main components: adapters and the application. In our case, the adapter component consists of an API adapter and a persistence adapter, as our application does not rely on external APIs.

The API adapter contains the web controllers responsible for handling API calls and requests, several helper classes in the utils directory that provide functionalities such as caching and AuthorizationBinding, as well as the models used within the API. The persistence adapter also has its own models, as each adapter in a hexagonal architecture should depend only on its own defined models. Additionally, each model has a dedicated repository responsible for storing and managing entities.

The application component consists of three main elements. First, the domain component, where the business logic resides. This includes mappers that handle conversions between the models of both adapters and the corresponding domain models. Additionally, there are predefined classes with methods for generating specific models, which are useful for testing the application. Second, the ports, which contain the interfaces for communication between the API and persistence adapters. In a well-structured hexagonal architecture, the application should only interact with the adapters through these interfaces to ensure a clear separation of concerns. Lastly, the service directory, another key part of the application component, where the core business logic is implemented. Here, parsed requests from the web controllers are processed and forwarded to the appropriate repositories in the persistence component.

One of the most challenging aspects of implementing a hexagonal architecture is designing and maintaining each component as truly independent. It is easy to unintentionally couple components, for example, by directly using repository methods within the corresponding services. Adhering strictly to the architectural principles ensures better modularity, maintainability, and testability of the application.

3 Explanation of the API technology

For this project, we have chosen a Quarkus API architecture. This decision is based on several factors. REST is a widely adopted industry standard that significantly simplifies integration with other systems. Additionally, its stateless communication enables high scalability, which is particularly beneficial for large-scale applications. Furthermore, REST offers great flexibility, as it is compatible with various frontend technologies and mobile applications. Another crucial advantage is its ease of implementation: Quarkus provides comprehensive support for REST endpoints, authentication, and persistence, making development more efficient.

This technology brings several benefits to our project. REST is simple and lightweight, as it relies on JSON over HTTP, which facilitates both implementation and debugging. The stateless architecture also ensures efficient resource utilization. Moreover, REST integrates seamlessly with Quarkus' built-in extensions for database access, enabling a smooth connection with the database.

Despite these advantages, some limitations and trade-offs need to be considered. One of the biggest challenges is the lack of real-time communication. Unlike WebSockets or GraphQL subscriptions, REST does not provide built-in support for real-time updates.

Despite this limitation, the benefits of a Quarkus API outweigh the drawbacks for our use case, ensuring a robust, flexible, and highly scalable solution for our project.

4 Implementation Details

As a framework we chose Quarkus and therefore REST Easy. Already collecting experience with Quarkus is a benefit for the members of our project. Furthermore, from a future-proofing perspective, Quarkus is ready for the use as a base for micro-services, meaning the application being split into multiple small parts. To implement mapping we went with MapStruct. We choose it because it is a modern approach to mapping as well as being generated instead of manually written, this reduces the possibilities of user error in the implementation. For authentication, we chose to implement a custom middleware that interacts with the THWS Authentication System and takes the Bearer token it generates. This decision relies on the ever increasing popularity of using bigger providers like Google or GitHub as the sole authentication source.

5 Testing Strategy

This testing strategy describes the approach to ensuring software quality for the Quarkus project. It includes different testing levels to ensure the functionality, security, and scalability of the application. The tests are divided into four main categories: unit tests, integration tests, database tests, and CI/CD tests.

Unit tests ensure that individual modules, particularly services and repositories, function correctly in isolation. JUnit 5 is used for this purpose. These tests run in an in-memory environment without a real server to provide quick feedback on code changes.

Integration tests verify the interaction of multiple modules, particularly REST APIs and services, using RestAssured and Quarkus-Test. These tests run in a Quarkus test instance with an in-memory database.

Database tests ensure the correct functionality of CRUD operations. An H2 database is used to simulate a realistic environment.

The implementation of the tests covers all key components of the application. Unit tests validate individual methods, while integration tests check API endpoints and their interactions. Database tests secure CRUD operations, and CI/CD tests ensure continuous quality assurance.

Overall, this strategy guarantees a stable, high-performance, and error-free application. By combining different testing methods, it ensures that both individual modules and the entire system function reliably.

6 Learning Outcomes and Reflection

Laurin's notes: next time, more time planning before we start coding (esp. when the API-technology is not yet picked); smaller goals that can be achieved quicker so we can support earlier in the process if someone needs help; utilising Git to it's full potential (namely creating milestones and issues;)

Caption

This article was drafted and refined using GPT-4 based on an outline containing related information. The GPT-4 output was reviewed, revised, and enhanced with additional content. It was then edited for improved readability and active tense, partially using Grammarly.