Final technical report

Juan Pablo Mosquera

Joshoa Alarcon

This technical report outlines the development of a backend for a Q&A forum, focusing on robustness, scalability, and maintainability. Built with Python and FastAPI, the backend supports key features like user authentication, question and answer submission, voting, and real-time updates.

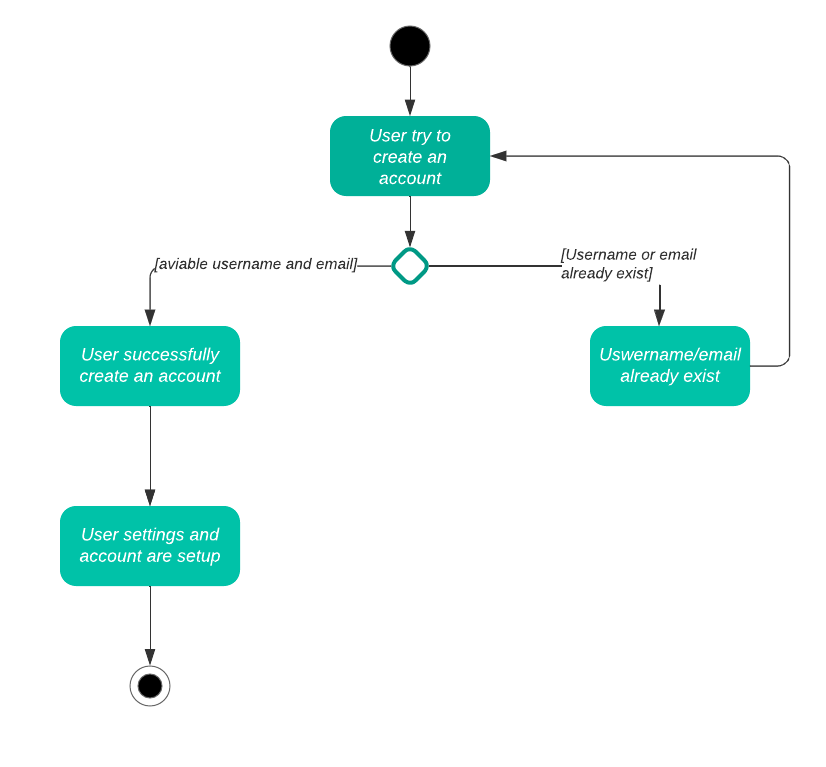
We utilized Pylint and Black for code quality, Poetry for dependency management, and Docker for containerization. The system is designed for scalability. Extensive testing—covering functionality, performance, scalability, and integration—ensured the backend's reliability. This concise report is supplemented with visual diagrams to illustrate the system architecture and development processes.

User stories:

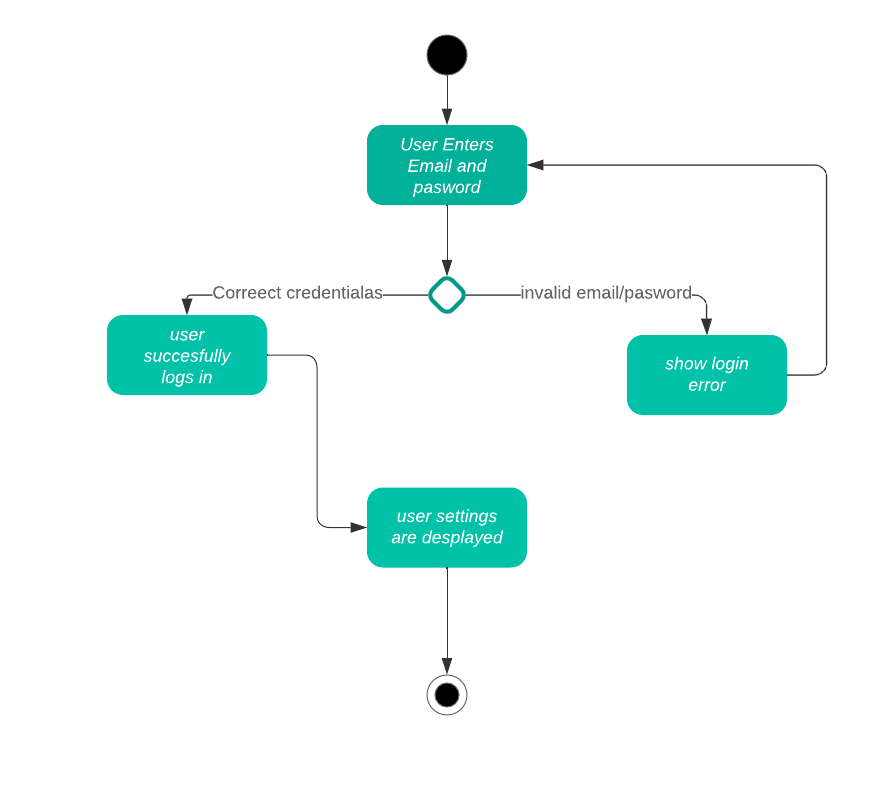
* As a user, I want to be able to post a question, so that I can get help from the community.
* As a user, I want to be able to answer a question, so that I can help other users.
* As a user, I want to be able to vote on a question or answer, so that I can indicate the quality of the content.
* As a user, I want to be able to search for questions and answers, so that I can find the information I need.
* As an administrator, I want to be able to delete inappropriate posts, so that I can keep the forum clean and safe.
* As an administrator, I want to be able to manage user accounts, so that I can control who has access to the forum.

Activity diagrams:

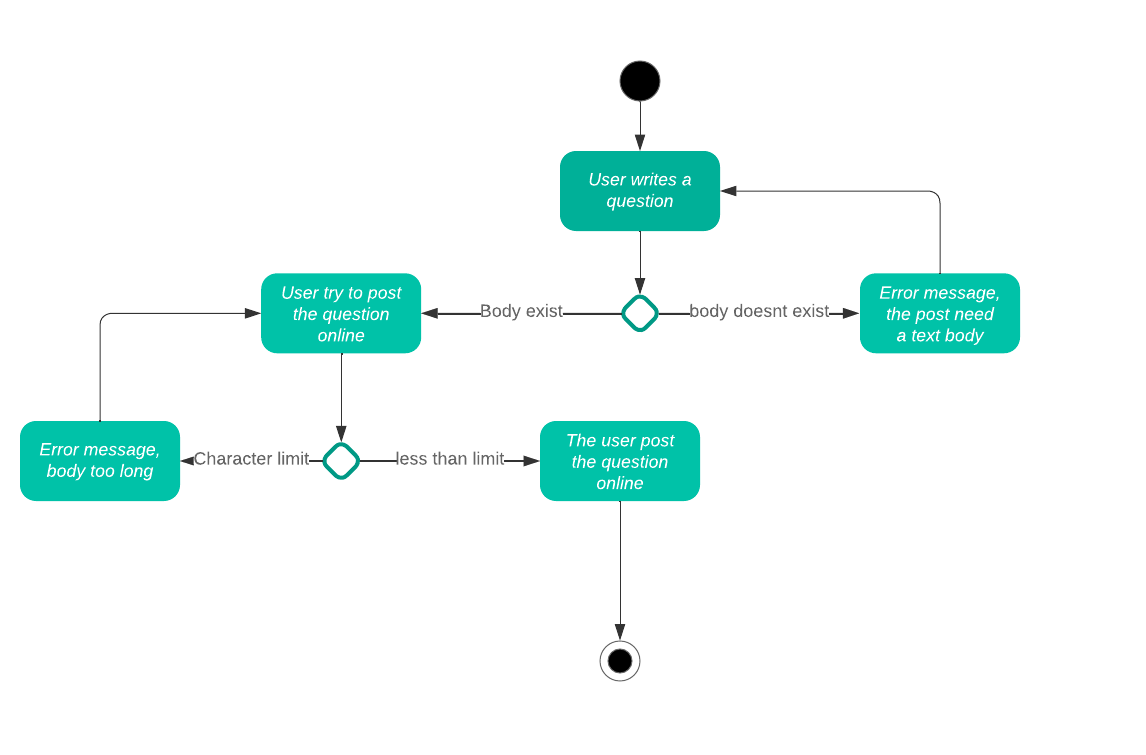
* User registration



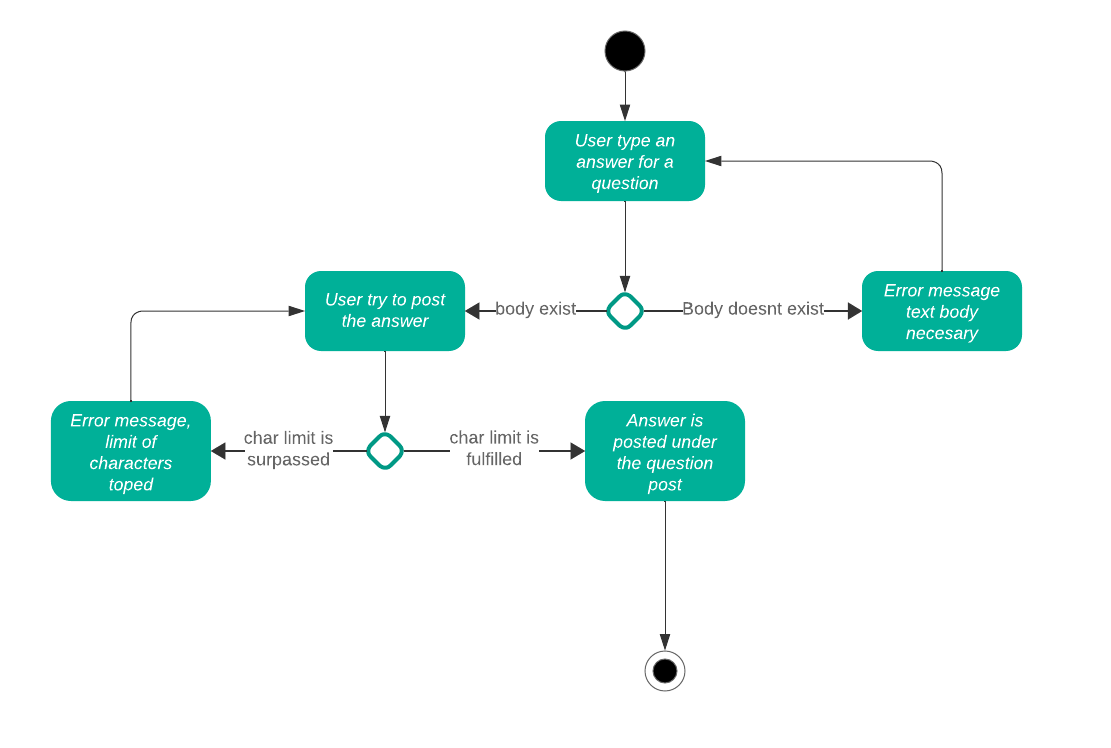
* User login



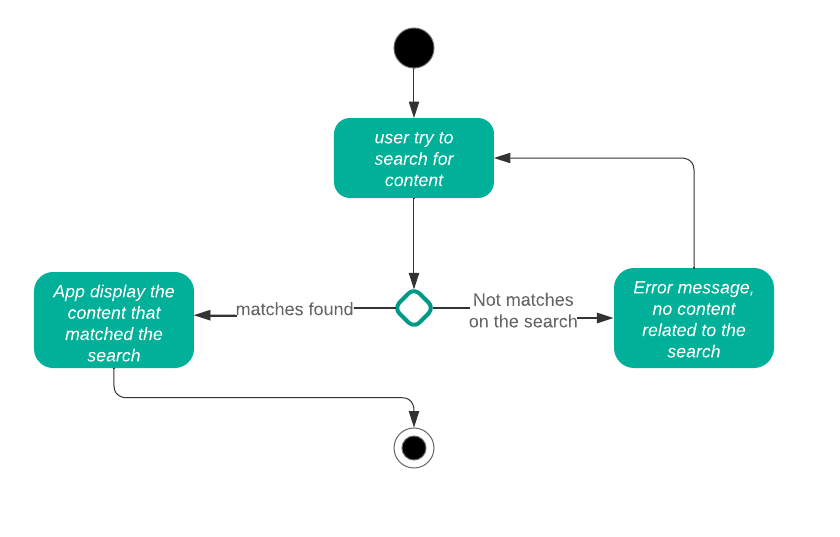
* Posting question



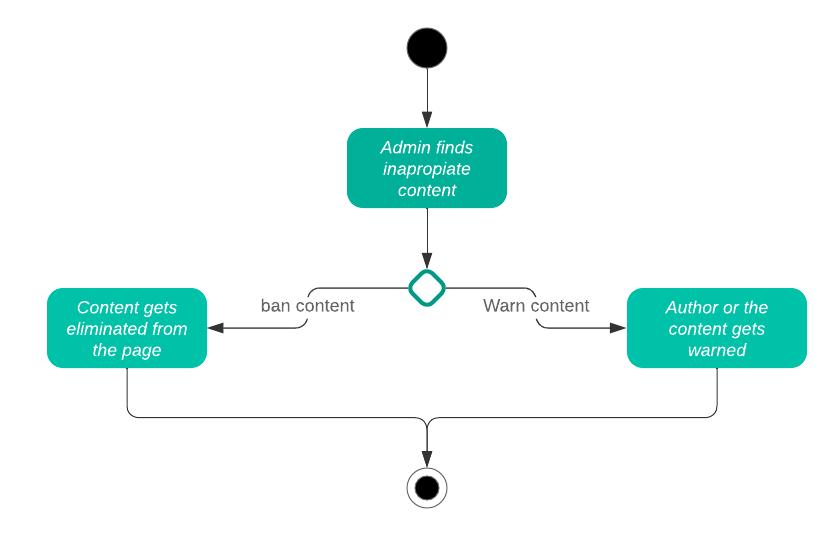
* Posting answer



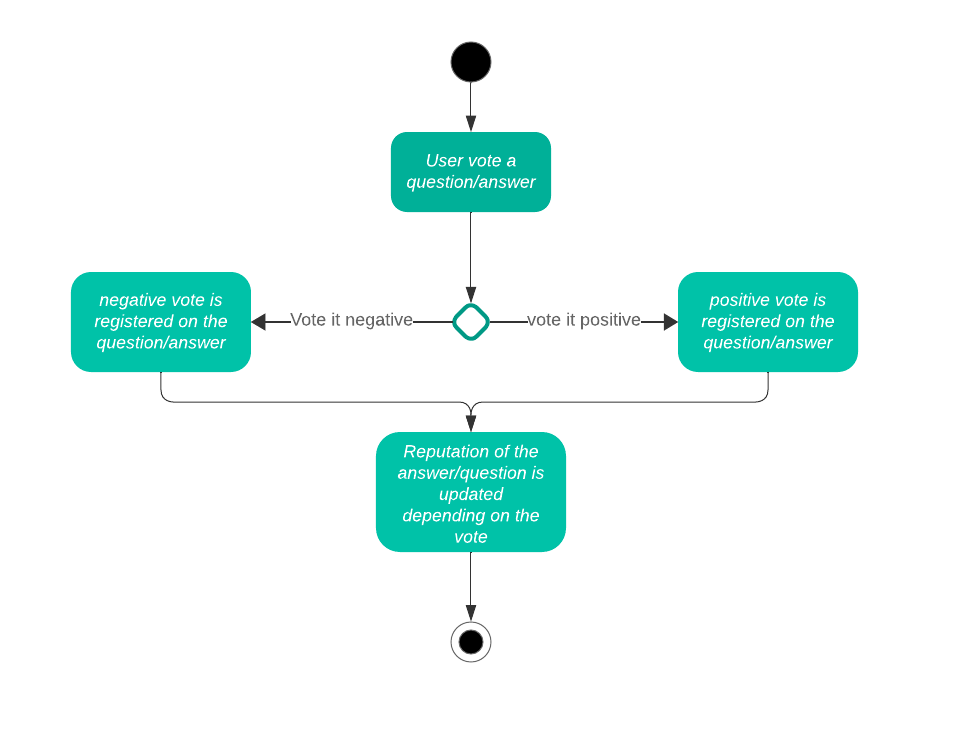
* Searching content



* Moderating content

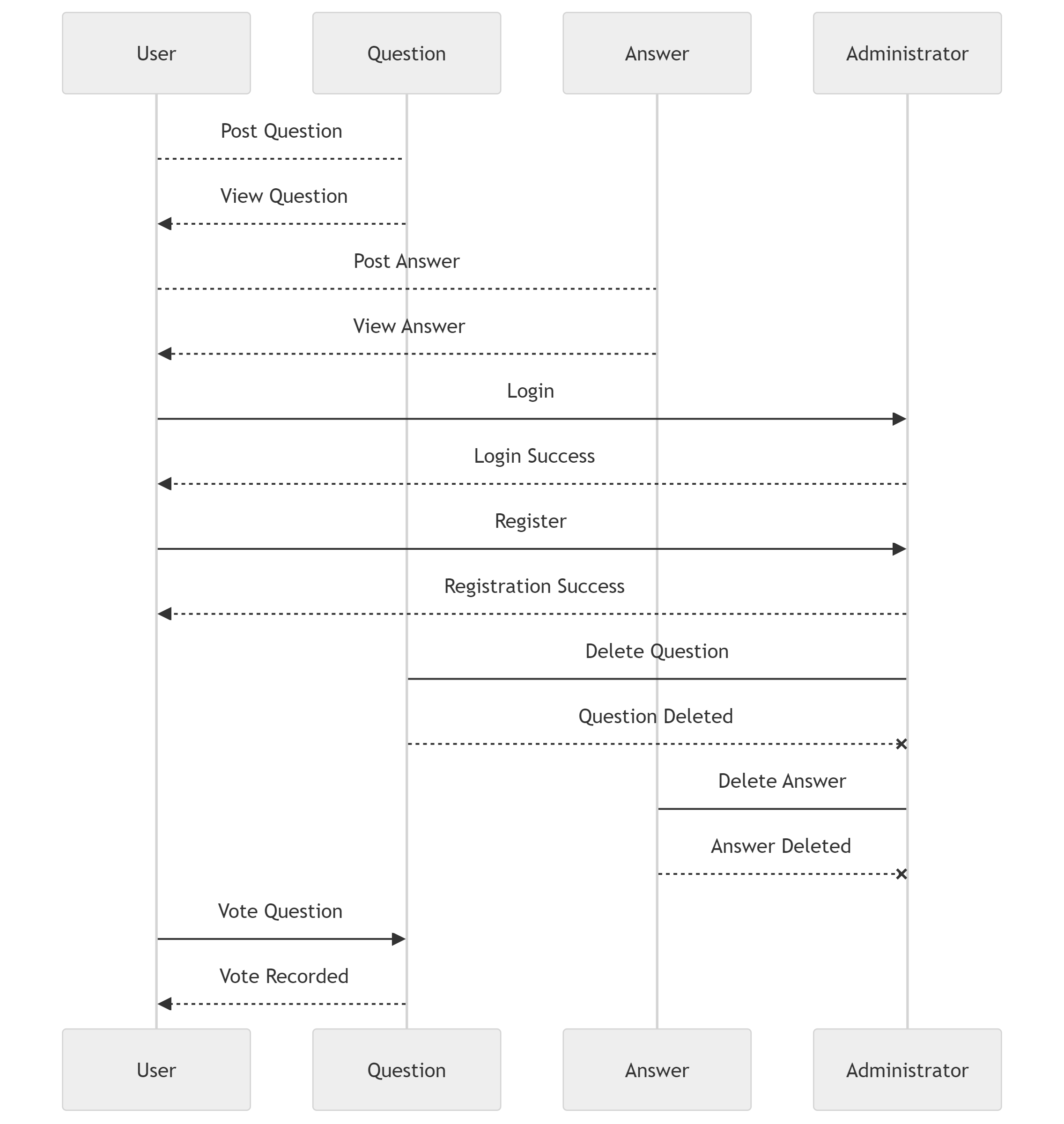


* Voting content



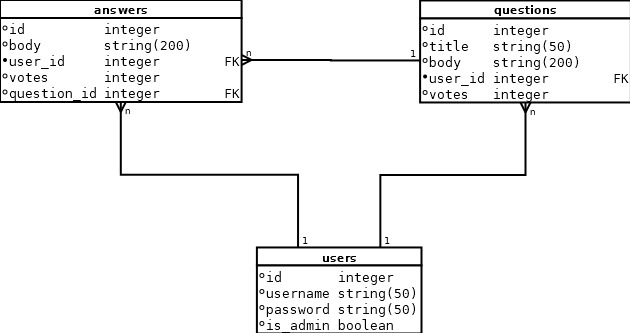
Each process with its activity diagram is derivated from the necessities described in the user stories

Also related to the processes we visualize the interaction of them with the users in the next sequence diagram:



Database desing:

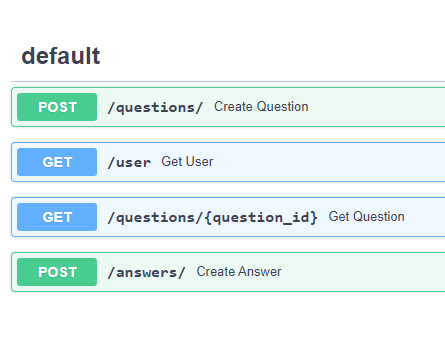
Only 3 tables based on the relation between user answer and question, one user can have many questions but a question only have one user, also a user can have many answers, but an answer only belongs to one user, and a question can have many answers but an answer only belong to one question



Code smells and code quality:

Regular checks on pylint while the development of the code made us to, not have to correct all de code quality at the end of the project but to have the code already in quality before finishing it

Web desing:



Post and get function in the fastapi docs from localhost:8080/docs in wich we see the functions to get and post question answers and also get an user

Decision making:

In the development of our backend for the Q&A forum, several critical decisions were made regarding design patterns and their implementations. Here are the specific decisions made:

* Singleton for Database Management:

Decision: We used the Singleton pattern for the subsystem that creates and works with the database.

Rationale: The Singleton pattern ensures a single instance of the database connection throughout the application, maintaining consistency and preventing multiple connections that could lead to conflicts.

* Decorator for User Creation:

Decision: The user creation subsystem utilizes the Decorator pattern.

Rationale: This pattern allows us to dynamically add responsibilities to user objects, such as granting administrative capabilities, without modifying the existing user class.

* Factory for Post Creation:

Decision: A Factory pattern is used to differentiate between posts with multimedia and those with only text and hyperlinks.

Rationale: The Factory pattern provides a way to create objects based on specified criteria, which is useful for managing different types of posts uniformly.

* Composite for Responses:

Decision: The Composite pattern is considered for managing responses.

Rationale: This pattern allows us to treat individual objects and compositions of objects uniformly. It is particularly useful for managing nested responses.

* Observer for Real-Time Updates:

Decision: The Observer pattern is considered for allowing responses to be added and updated as they are published.

Rationale: This pattern helps in implementing real-time update functionality, where changes in one part of the system can trigger updates in another.

* Composite for Forums and Subforums:

Decision: The Composite pattern is used for handling forums and subforums, which contain posts.

Rationale: This pattern is appropriate as it allows for a hierarchical tree structure where a forum can contain subforums and posts, simplifying the management and navigation of nested forums.

* Facade as Interface for Subsystems:

Decision: A Facade pattern is used to provide a unified interface for each of the subsystems.

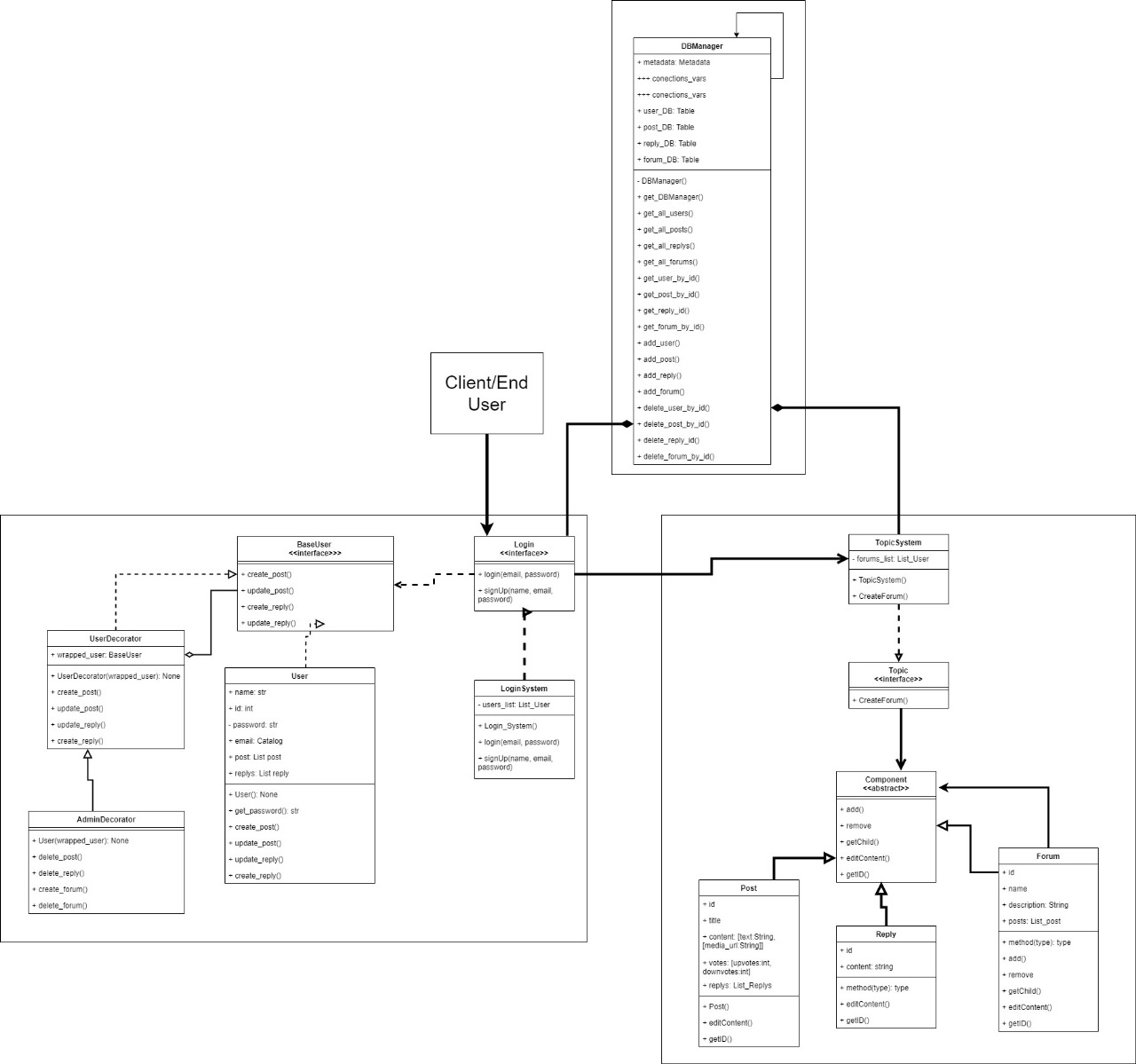
Rationale: The Facade pattern simplifies interaction with the subsystems, providing a high-level interface that makes the subsystems easier to use and integrate

* Command Pattern for User and Forum Interaction:

Decision: The interaction between users and forums (and their contents) is managed by either using a separate subsystem with the Command pattern or by encapsulating the forum subsystem to handle all interactions.

Rationale: The Command pattern encapsulates requests as objects, allowing for parameterization and queuing of requests. This pattern provides flexibility and decouples the sender and receiver of the commands. Encapsulating interactions within the forum subsystem could simplify the architecture but might reduce flexibility

By carefully selecting these design patterns, we ensured that our backend is flexible, maintainable, and scalable. These patterns provided clear structure and responsibility segregation, contributing to the robustness and efficiency of our system. Alternative patterns were considered but ultimately not chosen due to complexity, lack of alignment with our requirements, or potential for increased maintenance overhead.

Class diagram