Documentation Assignment 3

# Introduction

This application is a simple version of StackOverflow. It is implemented with java spring and I use the mariaDB which is a mySQL fork.

Key features: It allows users to ask questions, post answers, and vote on questions and answers.

- Only registered users can perform actions on the system, and passwords are encrypted for security.

Technology

My application is built using Java Spring and uses MariaDB as the database management system. I am using IntelliJ IDEA as my Integrated Development Environment (IDE).

# Use case diagrams

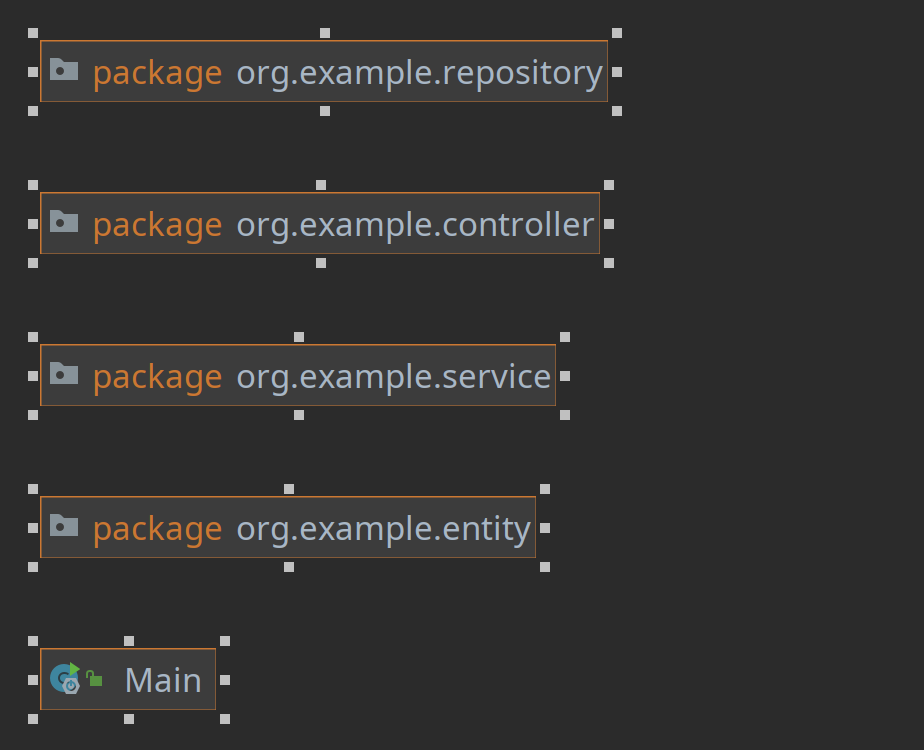
A use case diagram specifies the expected behavior of the application features and is usually used while developing the application while in its begging stages and design.

In this application so far there are the following use cases:

* Register: A user can create an account on the system.
* Login: A user can authenticate the system.
* Ask Question: A user can create a new question.
* Edit Question: A user can modify a question they have previously created.
* Delete Question: A user can delete a question they have previously created.
* Answer Question: A user can post an answer to a question.
* Edit Answer: A user can modify an answer they have previously posted.
* Delete Answer: A user can delete an answer they have previously posted.
* Vote: A user can vote on a question or answer.

# Package diagram

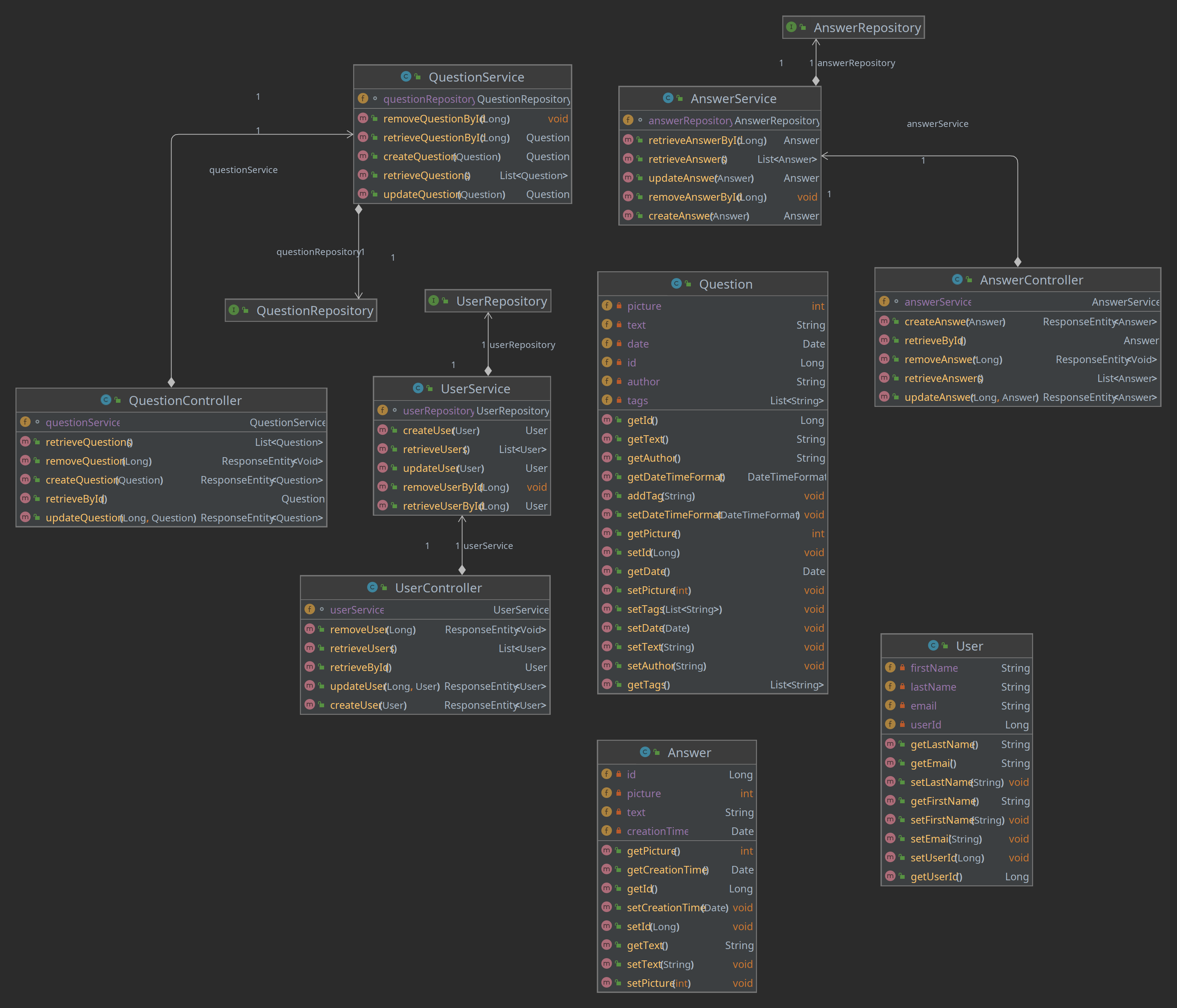
Here are all the packages present in the layered architecture.



# Class diagram

* User: Represents a user of the system.
* Question: Represents a question posted by a user.
* Answer: Represents an answer posted by a user.

Each class has attributes such as name, email, password, text, creation date, picture, tags, vote count, and relationships with other classes such as one-to-many between User and Question, one-to-many between Question and Answer, and many-to-one between Answer



Architecture

This Java Spring application follows a layered architecture pattern, which is a widely used design pattern in software development. The layered architecture pattern divides the application into distinct layers, with each layer performing specific tasks and communicating with the adjacent layers.

My layers are:

1. Entity Layer
2. Repository Layer
3. Service Layer
4. Controller Layer

These are the layers and their responsibilities:

**Entity Layer:** The entity layer is responsible for defining the domain models of the application. In this case, the User, Question, and Answer entities are defined in this layer. These entities represent the core data structures of my application and encapsulate the business logic that operates on the data.

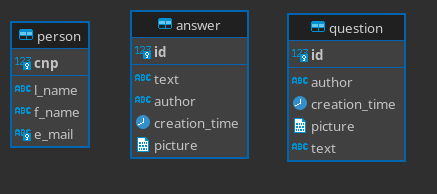
**Repository Layer:** The repository layer provides an abstraction over the data persistence layer, enabling my application to interact with the database. The repositories encapsulate the database operations such as Create, Read, Update, and Delete (CRUD) operations for the entities defined in the entity layer.

**Service Layer:** The service layer is responsible for implementing the business logic of my application. This layer serves as an intermediary between the repository layer and the controller layer. The service layer coordinates the data access and manipulations across multiple entities, implements the business rules, and ensures data integrity. The services in this layer provide high-level operations that can be called by the controller layer.

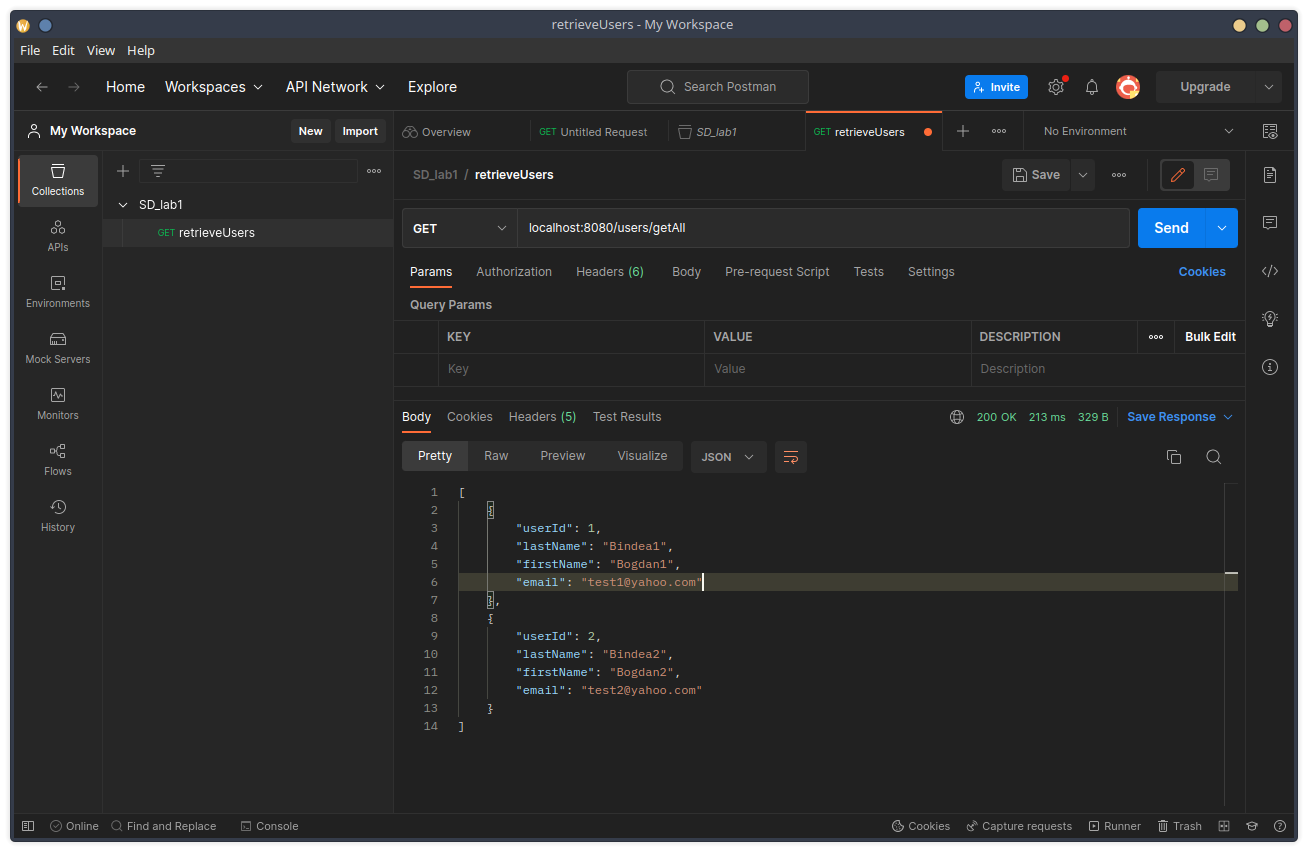
**Controller Layer:** The controller layer is the entry point for my application's API endpoints. It receives HTTP requests from the client and returns HTTP responses. The controller layer maps the incoming request to the appropriate service layer method and returns the results back to the client in the form of a response.

* Overall, the layered architecture pattern provides a clear separation of concerns in my application, making it easier to maintain, test and extend. The clear separation between the different layers of the application allows for greater modularity, making it easier to swap out or modify individual components without affecting the rest of the system.

Database diagram



Endpoints

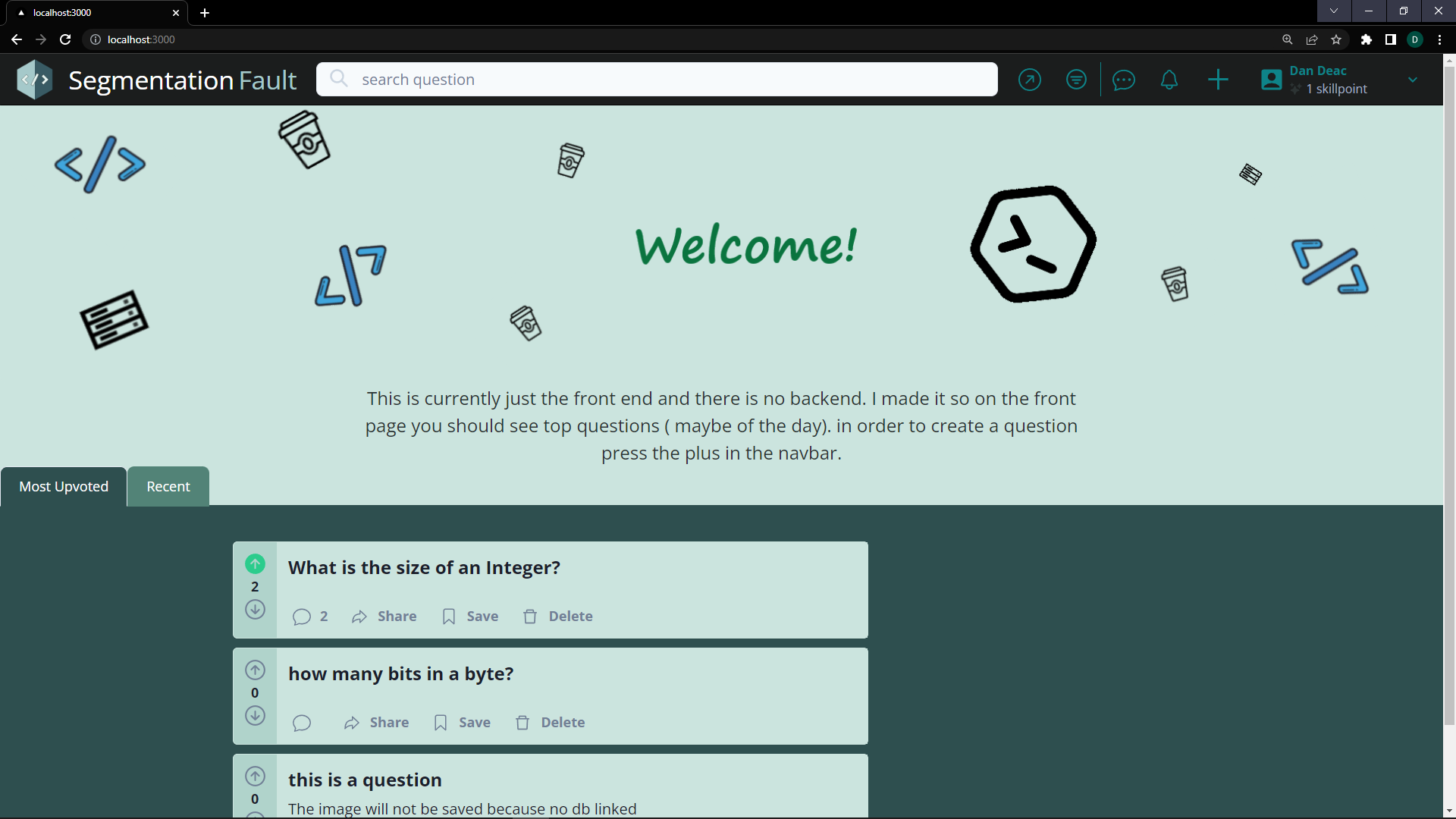


Introduction



Front End

<<Segmentation Fault – Dan Cristian Deac>>

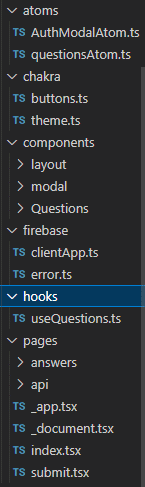


The documentation covers the front-end part of the Software Design project named “Segmentation Fault”, which is a Stack Overflow inspired website for asking and answering questions. This is my first time using React to develop a web front, and first-time doing front end for the matter. I want to make something that not only looks good but is intuitive to use and easy to understand. I have not finished yet implementing everything because I need to tie everything to the backend to test some features and create them.

Tech Stack

This design was realized using **React** with typescript. In addition, I used a framework called **ChakraUI** for more components and help with design. It is a mobile first tool so It ensures the app can be perfectly fit for mobile screens. For routing I use next/router and Recoil module and for quick testing I used the **Firebase database**. IDE: VSCODE

Architecture



**src/atoms**: This folder contains atomic or reusable components that are not specific to any particular feature or page of the application. The AuthModalAtom.ts file contains the implementation for an authentication modal that can be used throughout the application, and the questionAtom.ts file contains the implementation for a question component that can be reused in different parts of the application.

**src/chakra**: This folder contains files related to the Chakra UI library, which is a popular UI library for React. The buttons.ts file contains custom button styles and configurations for the application, while the theme.ts file contains custom theme variables for the Chakra UI library.

**src/components**: This folder contains all the reusable components that are specific to a particular feature or page of the application. The components are further organized into subfolders, such as layout, modal, and Questions. **moda**l folder contains components that display modal dialogs, such as an authentication or confirmation modal.

**src/firebase**: This folder contains files related to the Firebase backend service. The clientApp.ts file contains the configuration for the Firebase client app, while the error.ts file contains custom error handling logic for Firebase errors.

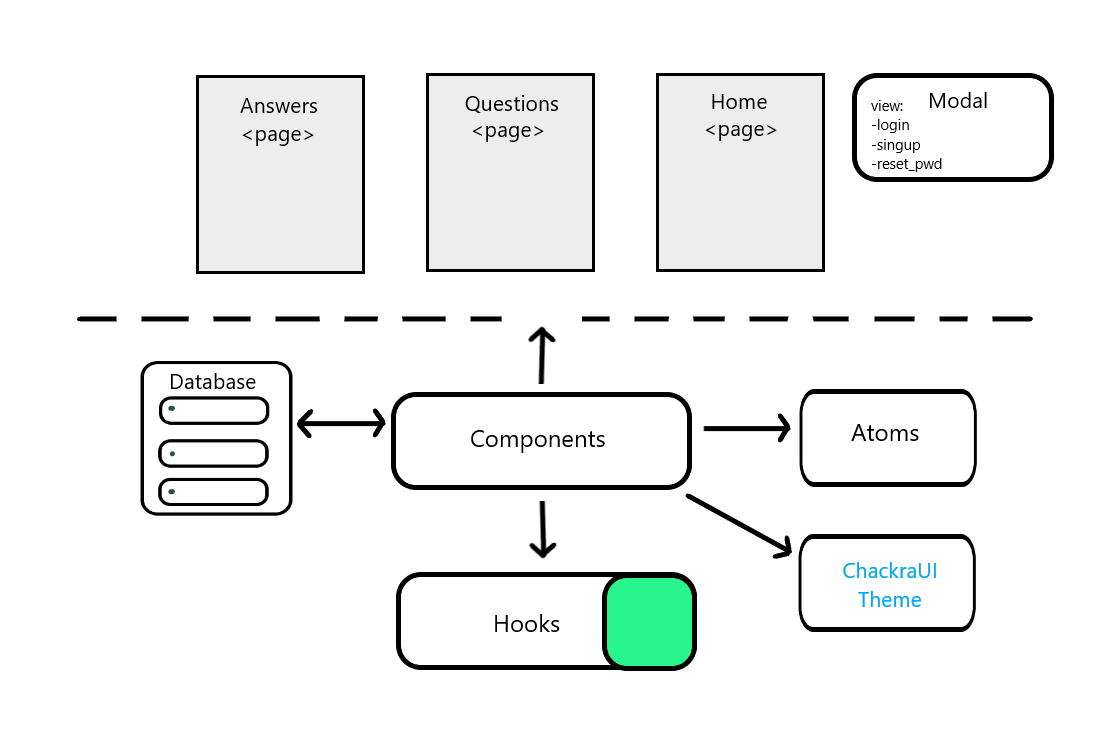
**src/hooks**: This folder contains custom hooks that can be used throughout the application. The useQuestions.ts file contains a custom hook for fetching and managing questions from the Firebase backend service.

**src/pages**: This folder contains all the top-level pages of the application. The answers folder contains the implementation for a page that displays answers to questions, while the index.tsx and app.tsx files contain the implementation for the application's main entry point and top-level component, respectively.

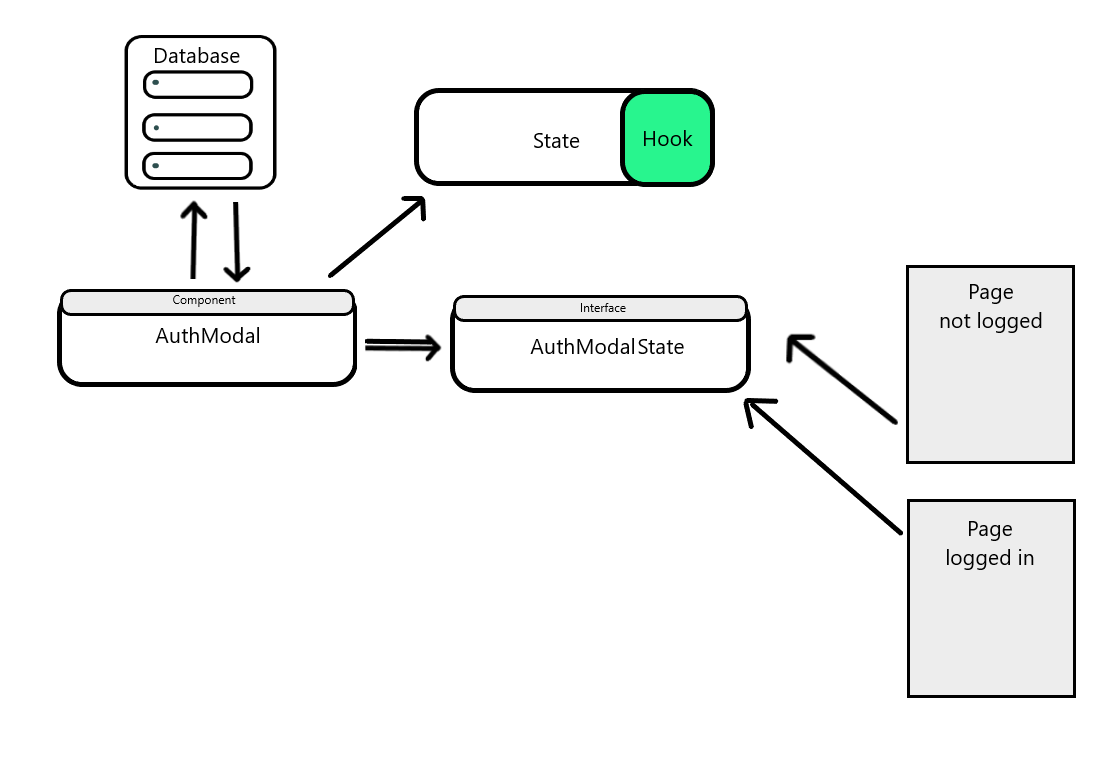
Overall, this folder structure follows a clear separation of concerns and a **modular approach** to organizing the code. The components are organized by feature or page, making them easier to locate and reuse, and the reusable components are further organized into subfolders for clarity. The custom hooks and Firebase-related files are also organized into their own folders, making them easier to manage.

Logic diagram

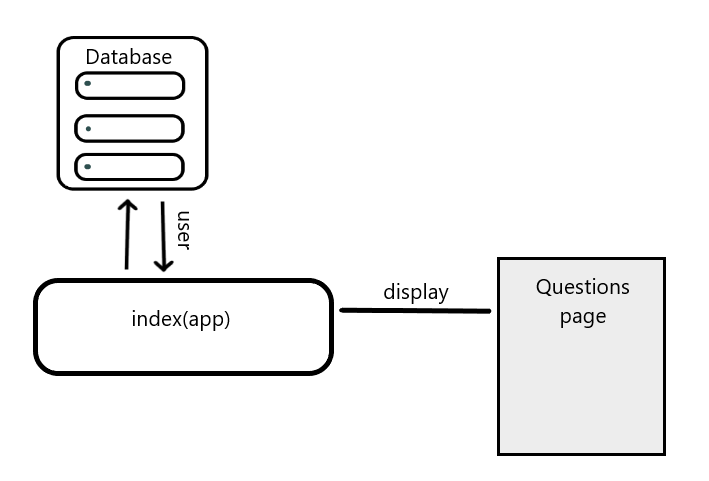
1.General diagram



2.AuthModal



3.Home Questions



Routing

Routing is an important aspect of any front-end application that determines how different pages or components of the application are accessed and navigated by the user. In this project, we are using the **next/router** package in combination with **React** to handle routing.

In this code snippet, we are using the push() method of the router object to navigate to the /submit page. This method takes the desired URL as its argument and navigates to that page.

In addition to navigating to a particular page, we can also use routing to handle user authentication and access control. For example, in the following code snippet, we are checking whether the user is authenticated before allowing them to access the /submit page:

const setAuthModalState = useSetRecoilState(authModalState);

  const router = useRouter();

  const createQuestion = () => {

    if (!user) {

      setAuthModalState({ open: true, view: "login" });

      return;

    }

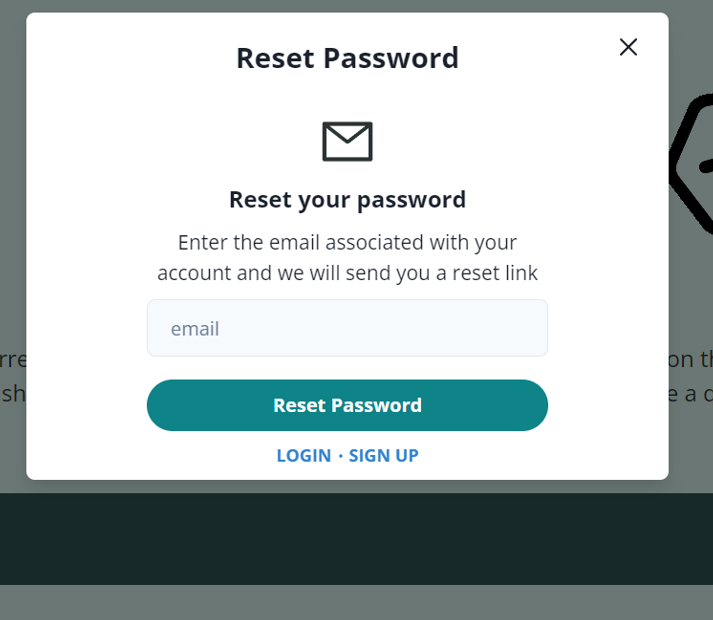
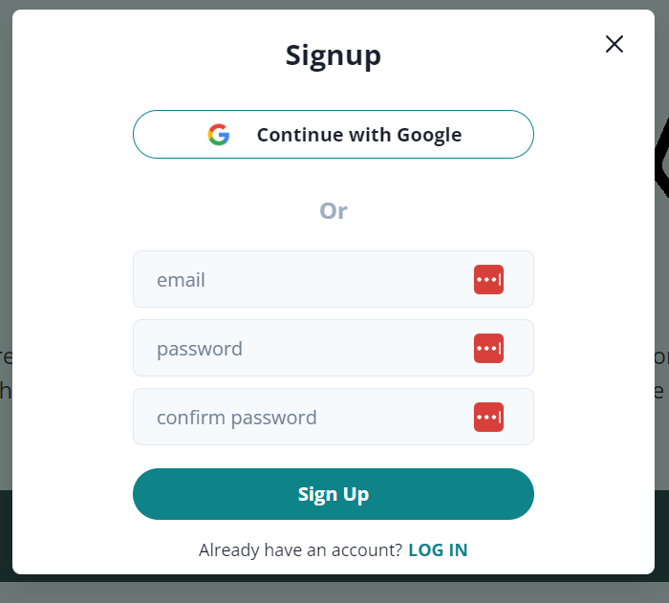
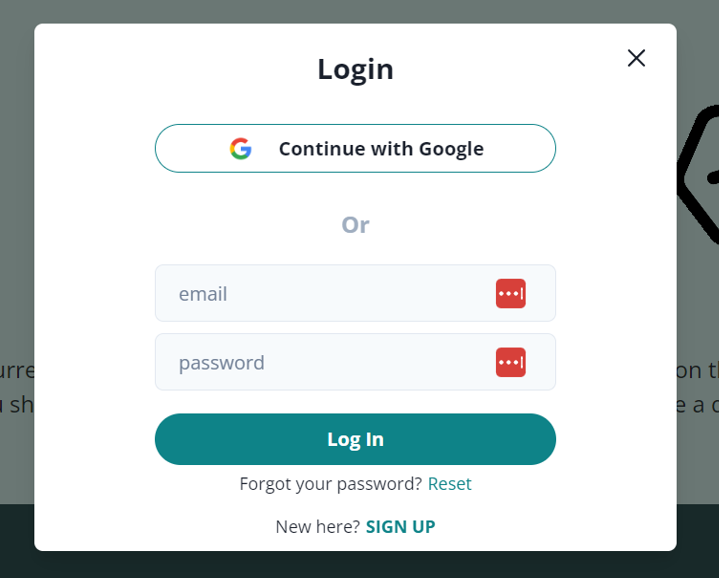
    router.push(`/submit`);

  };

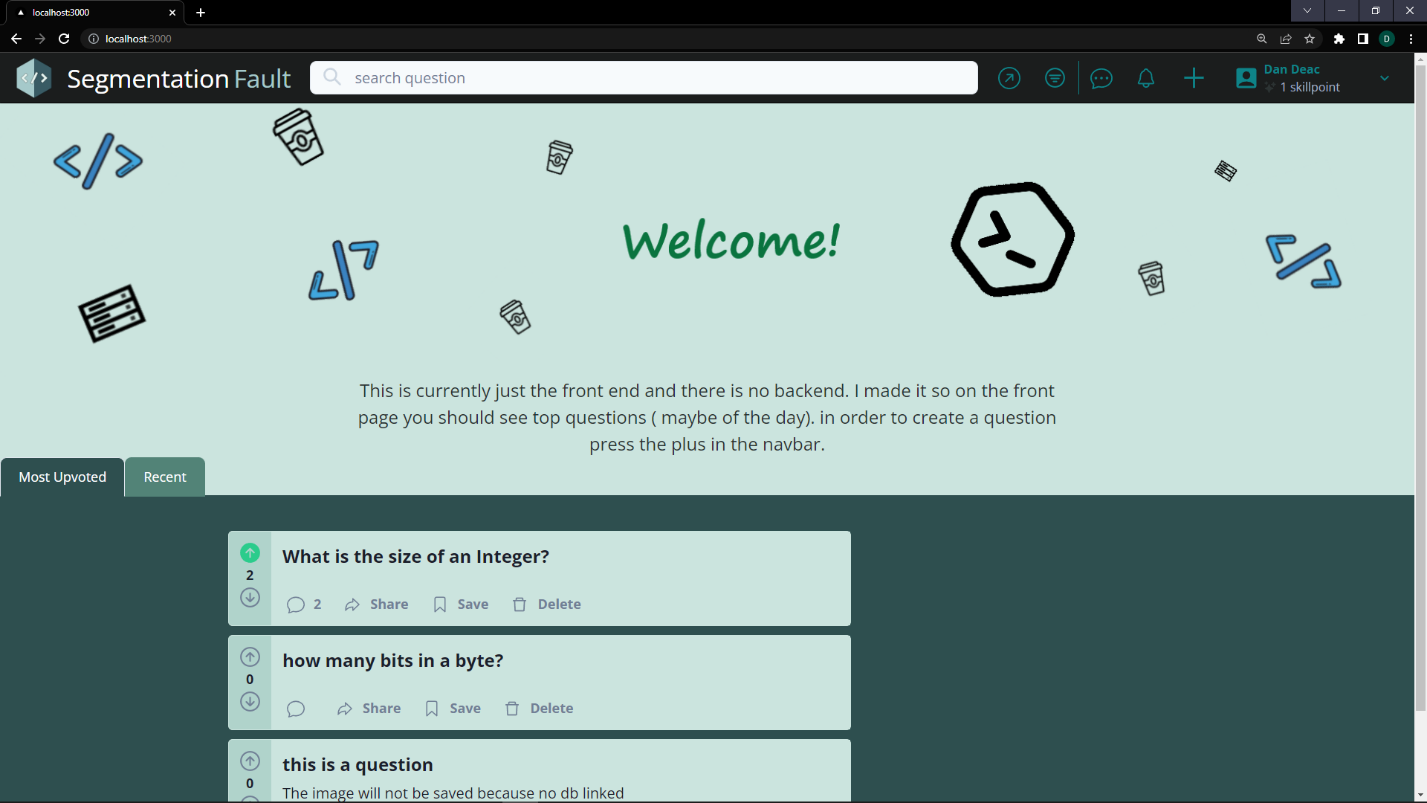
In this code snippet, we are using the **useSetRecoilState** hook to set the state of the **authModalState** when the user is not authenticated. This opens a modal prompting the user to login or register before they can access the /submit page.

Interface:

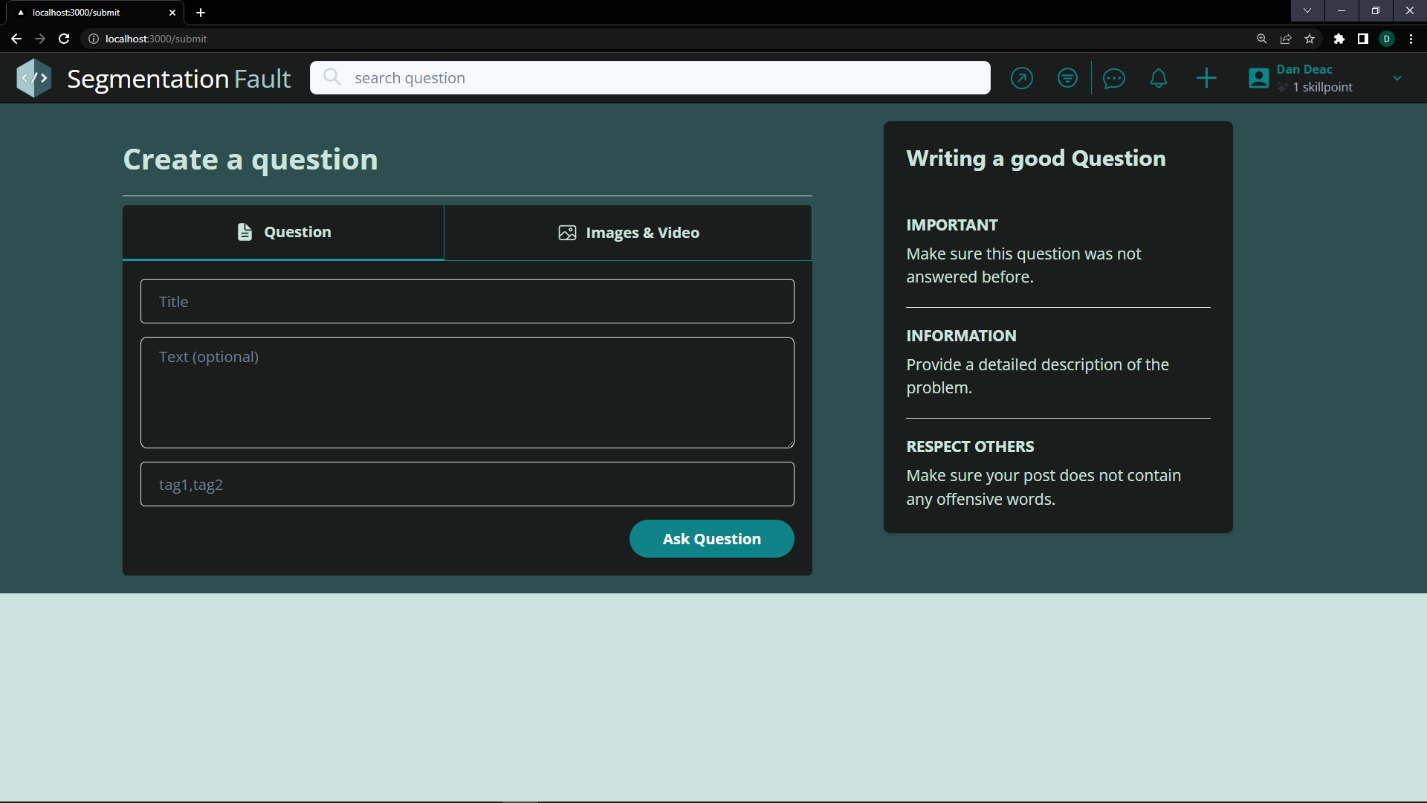
1.Modals



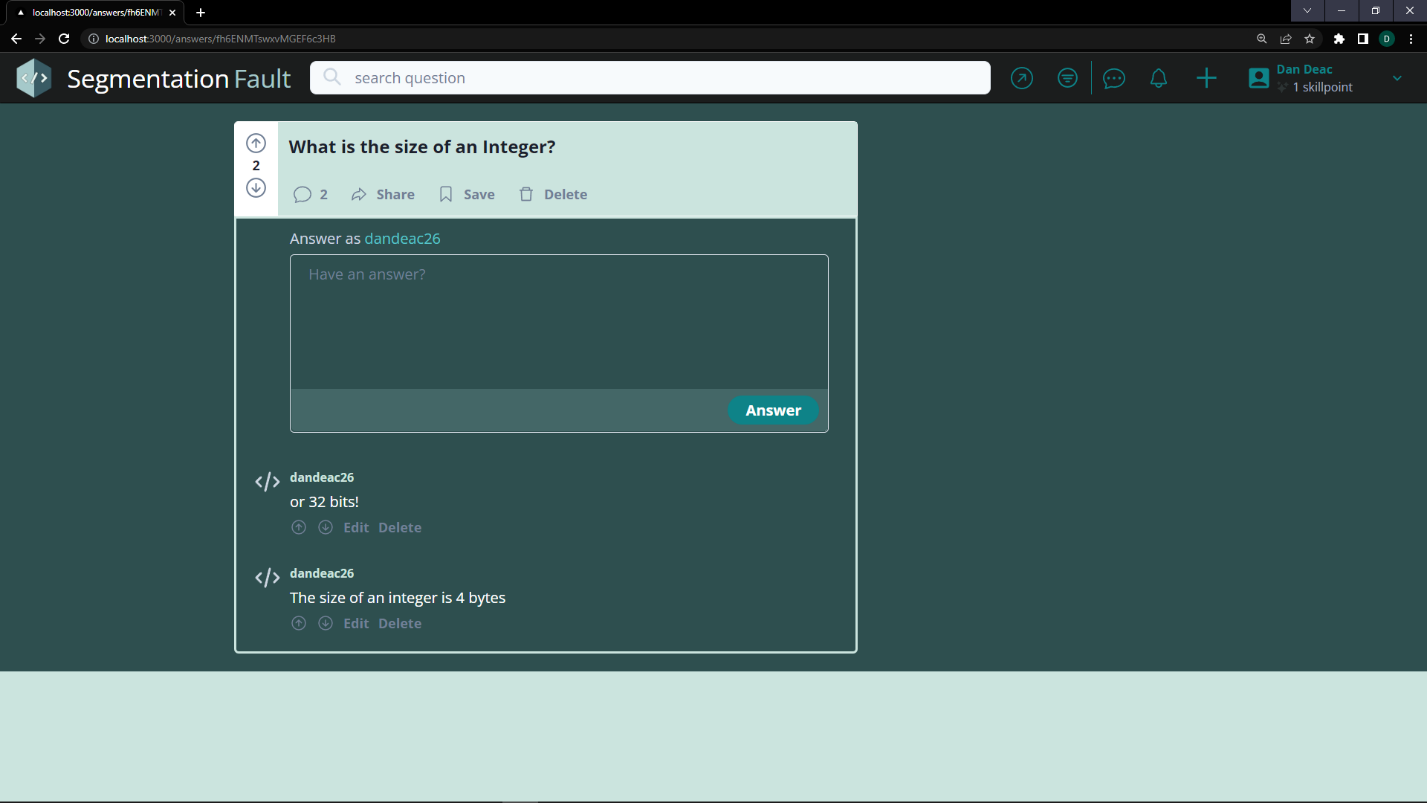
2.Home



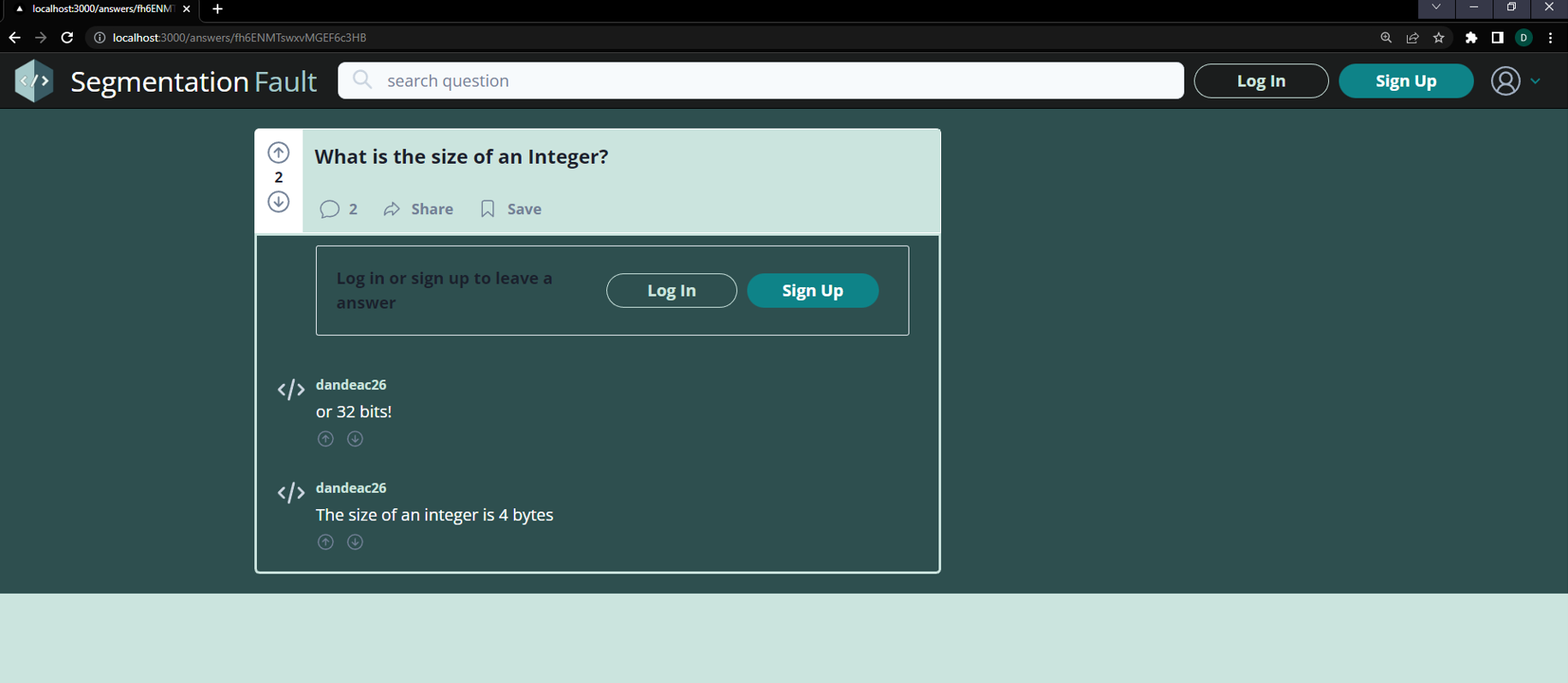
3.Ask Question



4.Answer



5.Before Login



Conclusion

This is not really a conclusion because this is going to get tied to a database but I had to end on a note. The current state of the interface is quite mediocre. I need to tie things to the database and to the backend in order to perform significant operations and make sure things work fine. I will switch from firebase the mariadb I have on my workstation and hope things work out fine.

REPO: <https://github.com/dandeac26/SegmentationFaultWeb.git>

!!! Use Branch: **BranchFromWindows**. I have not yet merged / pushed to the main branch for now

Resources

Here are the resources I used for learning and developing this application:

- StackOverflow.com – for inspiration ofc.

- <https://www.youtube.com/@freecodecamp>

- https://www.youtube.com/watch?v=rCm5RVYKWVg&t=11744s