

Assignment 3 - Documentation

Simple StackOverflow

Software Design

Faculty of Automation and Computer Science

2022-2023



Submitted by:

Noémi Veres

group – 30433

Contents

Introduction.....	4
Technologies	5
Problem Analysis and Use Cases.....	6
Architecture	10
Packages.....	11
Classes	12
Database.....	14
Endpoint Requests	16
FRONT-END.....	19
Architecture	20
Routing.....	22

Table of Figures

Figure 1: Use Case Diagram	7
Figure 2: Package Diagram.....	11
Figure 3: Class Diagram	12
Figure 4: Database Diagram	14
Figure 5: The Home Page of my web application.....	19
Figure 6: Dependency Injection.....	20
Figure 7: The way that questions are displayed.	22
Figure 8: Registration Form for a New User	23
Figure 9: Login Form.....	24
Figure 10: New Question Form	24
Figure 11: New Answer Form	24

Introduction

A simple version of StackOverflow will be designed and implemented. StackOverflow is a question-and-answer website where users can post questions related to programming and receive answers from the community. Users can vote on the quality of answers, and the most popular or helpful answers are typically displayed at the top. StackOverflow has become a popular resource for developers seeking help or guidance on specific programming issues. Our objective is to implement a similar system.

The sub-objectives are the following:

- Analyze the problem and identify requirements
- Design the system for asking and answering questions
- Implement the system for asking and answering questions
- Test the system for asking and answering questions

Our system will have two types of users: regular and administrator. No actions should be possible if the user is not logged in. The user's passwords should be stored in the database encrypted.

Our system for asking and answering questions will have the following main features:

- Users can ask questions, each question having an author, title, text, picture, creation date & time, and one or more tags. The list of questions is displayed sorted by creation date, and users can filter by tag, text search, user, or their own questions. Questions can be edited or deleted by their author.
- Each question can be answered one or more times by any user, each answer having an author, text, picture, and creation date & time. Answers can be edited or deleted by their author. When displaying a question, the list of answers is also displayed.
- Users can vote on questions and answers with upvote and downvote (like and dislike). Each user can only vote once on each question or answer and cannot vote on their own posts. The vote count is displayed for each question and answer, with the answers for a question sorted by their vote count.

Technologies

The core programming language used to write the application was Java, used together with the Spring Framework, providing features such as inversion of control and support for web application development. The IDE I used for this assignment was IntelliJ IDEA, which allowed me to write code, debug and test the application. A database management system was also needed, in order to store application data. My choice was MySQL. As a build tool for this Java Spring application, I used Maven. The web server used to deploy and run the application was Apache Tomcat in this case. For testing my code, I used the JUnit testing framework, which enabled me to write and execute unit tests.

REST (Representational State Transfer) is a software architectural style used to create web services that are lightweight, scalable, and easy to maintain. RESTful web services are in their turn a type of web service that adheres to the REST architectural principles. They use HTTP protocols to communicate between the client and server, and the server exposes a set of resources or endpoints that the client can interact with using standard HTTP methods such as GET, POST, PUT, and DELETE, corresponding to the CRUD operations: Create, Read, Update, Delete.

Overall, the combination of these technologies and tools is aimed to allow the development of robust and scalable Java Spring applications that are both efficient and easy to maintain.

Problem Analysis and Use Cases

Functional requirements:

- Users should be able ask questions, each question having an author, title, text, picture, creation date & time, and one or more tags.
- The list of questions should be displayed sorted by creation date.
- Users should be able to filter questions by tag, text search, user, or their own questions.
- Questions should be edited or deleted by their author.
- Each question could be answered one or more times by any user, each answer having an author, text, picture, and creation date & time.
- Answers should be edited or deleted by their author.
- When displaying a question, the list of answers should be also displayed.
- Users should be able to vote on questions and answers with upvote and downvote (like and dislike).
- Each user should only vote once on each question or answer.
- Users cannot vote on their own posts.
- The vote count should be displayed for each question and answer.
- The answers for a question should be sorted by their vote count.

Non-functional requirements:

- Users must be logged in to perform any actions.
- User passwords must be stored in the database encrypted.
- The system should be fast and responsive.
- The system should be reliable and available at all times.
- The system should be secure, with appropriate measures in place to prevent unauthorized access and other security issues.
- The system should be easy to use and intuitive for both regular and administrator users.
- The system should be scalable, with the ability to handle a large number of users and data growth.

A use-case diagram (see Figure 1) is a type of UML diagram used to visualize the functional requirements of a system or application. It provides a high-level view of the interactions between actors (users, systems, or other external entities) and the system, depicting the various use cases and the relationships between them. This diagram provides a structure for our application as it helps to identify components in the design phase. It also helps to capture the requirements which were presented in detail.



Figure 1: Use Case Diagram

Simple StackOverflow

Now the use cases will be presented:

Use Case: register

Primary Actor: user

Main Success Scenario:

1. The user clicks on the “Register now” button
2. The user enters the details and the chosen password 2 times
3. The user presses the “Register” button
4. The user will be registered

Alternative Sequence: The 2 passwords do not match

The client inserts 2 passwords which do not match

The application displays an error message and requests the user to insert the same password

The scenario returns to step 1

Use Case: login

Primary Actor: user

Main Success Scenario:

1. The user enters their email and password
2. The user presses the “Log in” button
3. The user gets logged-in

Alternative Sequence: Incorrect password

The user inserts an incorrect password

The application displays an error message and requests the user to make sure that the password is correct

The scenario returns to step 1

Use Case: ask a question

Primary Actor: user

Main Success Scenario:

1. The user logs in to the application
2. The user navigates to the “Ask a Question” section
3. The user enters the question in the text box provided
4. The user clicks on the “Post” button to submit the question
5. The question is posted and displayed to other users

Use Case: answer a question

Primary Actor: user

Main Success Scenario:

1. The user logs in to the application
2. The user navigates to the “Answer Questions” section
3. The user selects the question they want to answer from the list of questions
4. The user enters their answer in the text box provided
5. The user clicks on the “Post” button to submit their answer
6. The answer is posted and displayed to other users

Use Case: filter questions

Primary Actor: user

Main Success Scenario:

1. The user logs in to the application
2. The user navigates to the “Filter Questions” section
3. The user selects the criteria they want to filter by, such as tag, user, or text search
4. The application filters the questions based on the selected criteria and displays them to the user

Simple StackOverflow

Use Case: edit own question

Primary Actor: user

Main Success Scenario:

1. The user logs in to the application
2. The user navigates to the “My Questions” section
3. The user selects the question they want to edit from the list of their own questions
4. The user clicks on the “Edit” button next to the selected question
5. The user modifies the question text in the text box provided
6. The user clicks on the “Save” button to save the changes
7. The question is updated and displayed to other users

Use Case: delete own question

Primary Actor: user

Main Success Scenario:

1. The user logs in to the application
2. The user navigates to the “My Questions” section
3. The user selects the question they want to delete from the list of their own questions
4. The user clicks on the “Delete” button next to the selected question
5. A confirmation message is displayed to the user, asking if they are sure they want to delete the question
6. The user clicks on the “Yes” button to confirm the deletion
7. The question is deleted and no longer displayed to other users

Use Case: vote for a question

Primary Actor: user

Main Success Scenario:

1. The user navigates to the question they wish to vote for.
2. User selects the upvote or downvote button for the question.
3. The system updates the question's vote count accordingly.

Alternative Sequence: Invalid vote

If the user has already voted on the question, the system displays an error message informing the user that they have already voted.

For editing own questions and answers, and for voting as well, a similar approach is used.

Use Case: ban user

Primary Actor: Moderator

Main Success Scenario:

1. The moderator logs into the system.
2. The moderator navigates to the user profile they wish to ban.
3. The moderator selects the "Ban User" option.
4. The system presents a pop-up message asking the moderator to confirm their decision.
5. Moderator confirms the decision to ban the user.
6. The system deactivates the user's account, preventing them from accessing the platform.
7. The system sends an email and SMS notification to the banned user, informing them that their account has been deactivated due to violating the platform's terms of use.

For unbanning a user, similar steps are performed.

Architecture

Layered Architecture is a popular architectural pattern that is often used in web application development. It separates the application into different layers based on the responsibilities. So far I have 4 different layers: Controller, Service, Model and Repository.

Controller Layer: This layer is responsible for receiving user input from the UI, processing it, and sending it to the appropriate Service Layer. It also handles HTTP requests and responses, routing, and authentication. The Controller Layer is the entry point for the application and is responsible for controlling the flow of data between the UI and the Service Layer.

Service Layer: This layer contains the business logic of the application. It is responsible for processing and manipulating data. It also communicates with the Repository Layer to persist data. The Service Layer is where the main application logic resides, and it acts as a mediator between the Controller Layer and the Model Layer.

Model Layer: This layer represents the application's data and is responsible for the creation and manipulation of business objects. The Model Layer contains classes and data structures that represent entities in the application, such as users, questions, and answers in our case.

Repository Layer: This layer is responsible for persisting data to the database. It provides an abstraction layer between the Model Layer and the database, and it is responsible for querying, inserting, updating, and deleting data. It communicates with the database to perform CRUD operations (Create, Read, Update, Delete).

In a typical Layered Architecture, each layer is independent and can be developed and tested separately. It also enables developers to change one layer without affecting the other layers.

Overall, Layered Architecture is a widely used pattern that is particularly useful for building large and complex web applications. It provides a clear separation of concerns, which makes the application more modular, flexible, and easier to maintain.

Packages

Packages are used to organize the code and to provide a general view of the system's architecture. In the packages, I grouped together the classes which are correlated.

A package diagram is a UML diagram. It is representative for the system, as it shows the organization and the structure of the application. In such a diagram, the dependencies between packages should be indicated.

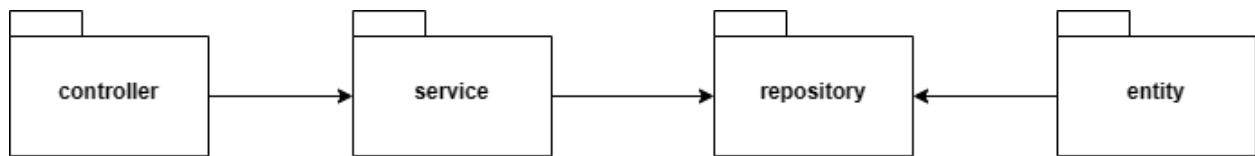


Figure 2: Package Diagram

In the context of my application, the package diagram (see Figure 2) depicts four main packages: controller, service, entity, and repository. Each of these packages contains related classes, and they are organized based on their functionalities and responsibilities. Here is a brief description of each package and its entities:

controller Package: This package contains different controllers for the three main entities of our application: Answer, Question, and User. Each controller handles the logic and routing for the respective entity and communicates with the corresponding service layer. This layer is responsible for handling incoming requests from clients and invoking the appropriate Service methods to process the requests.

service Package: This package contains the services for the three main entities of our application. Each service contains the business logic and rules for the respective entity and communicates with the corresponding repository layer to perform operations on the application's data.

entity Package: This package contains the entity classes for the five entities of our application: Answer, Question, which extend the Content class, User, and Tag. These classes represent the data that our application manipulates, and define their properties, relationships, and behaviours.

repository Package: This package contains the repositories for the three main entities of our application. Each repository handles the data access for the respective entity and communicates with the database.

Classes

A class diagram is a type of UML diagram, used to visualize the structure of a system. The package diagram, as its name says, focuses on the relationship between packages, but the class diagram provides an overview of the classes and interfaces and their relationships.

In a class diagram, each class is represented by a rectangular box, to which we can add its attributes and methods. Attributes describe the properties of an object, while methods describe the behaviours or actions that can be performed by the object.

Classes are connected to each other by lines and arrows that represent relationships between them. Different kinds of relationships exist, like inheritance, association, aggregation and composition.

In Figure 3 the main classes are presented.

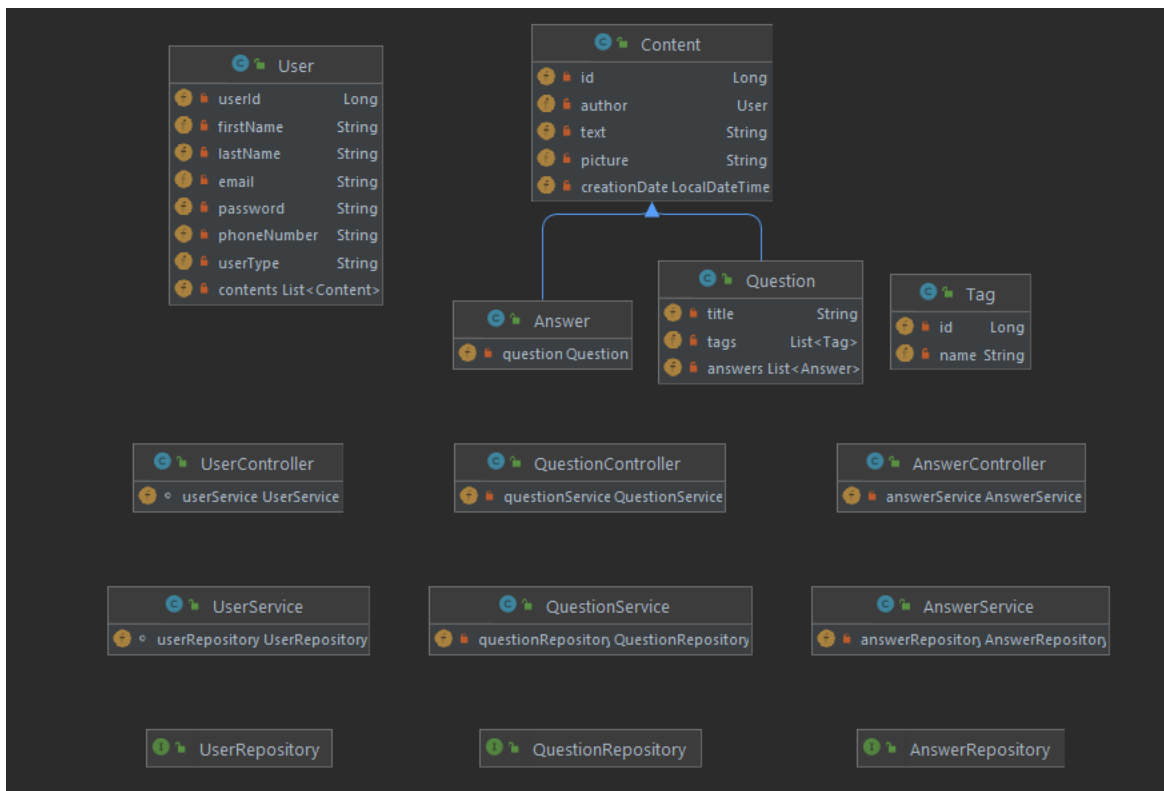


Figure 3: Class Diagram

The **User class** contains information about the user, such as id, name, email address, phone number and user type (which can be either regular or moderator). To each user corresponds a list of contents, each user's questions and answers.

The **Content class** can be viewed as a general one, it is a parent class. Each content has an author, some text, a picture and a creation date. These are the attributes which are in common for both questions and answers.

The **Question class** extends the Content class, we can say that it is its child, but has its own title, a list of tags and answers. The parent-child relationship is used to avoid duplicated code in the entities which share some attributes.

The **Answer class** also extends the Content class. In addition to the attributes present in the Content class, it also has reference to the question to which it responds.

The **Tag class** is aimed to represent the tags which can be added to each question, based on which a user can make some filtering. A question may have several tags, and several tags may belong to different questions, yet these questions might share a context.

As the project was evolving, I introduced a new class, the **Vote class**, which represents a user's vote on some content. This class has as attributes its own id, then a content with a corresponding user who voted for that content. It also has a type which can be either an upvote or a downvote.

All of the entities contain constructors, a no-argument version and a parameterized one. They also contain getters and setter for their private attributes.

The different **Repository interfaces** are similar in their structure, they extend the CrudRepository class. CrudRepository is an interface provided by the Spring Framework that allows the implementation of basic CRUD (Create, Read, Update, Delete) operations on entities. It provides several methods such save(), used to both create and update an object, findById(), findAll(), delete(), and deleteById(), which can be used to perform these common database operations.

The **Service classes** use the @Autowired annotation for repository injection. In this way, the service class can call methods on the repository object, without having to create a new instance of that repository. Service classes contain methods that implement business logic for the application and interact with repositories or other to perform database operations. The **EmailService** classes are somewhat special, as they are implementing the method for sending an email. Emails are sent via SMTP, using the JavaMail library. The **AuthenticationService** class handles authentication, it helps with registration and extracting the JWT from the authorization header. The JWT's payload contains information about the user who wants to login. When a user is registering, here happens the password encryption, for which the BCryptPasswordEncoder class was imported.

The **Controller classes** are responsible for handling incoming requests from clients. The controller typically acts as an intermediary between the user interface and the service layer, receiving input from the user and passing it to the appropriate service class to perform the necessary business logic. In our case, a corresponding service is injected into each controller class. Once the service has completed its processing, the controller then returns the appropriate response to the client.

Database

A database diagram is a visual representation of the structure of a database. It shows the tables in the database, the relationships between them, and the columns and data types for each table, also indicating primary keys. Database diagrams can be used to help developers and database administrators understand the structure of a database and even identify potential problems. The database diagram of our system can be seen in Figure 4).

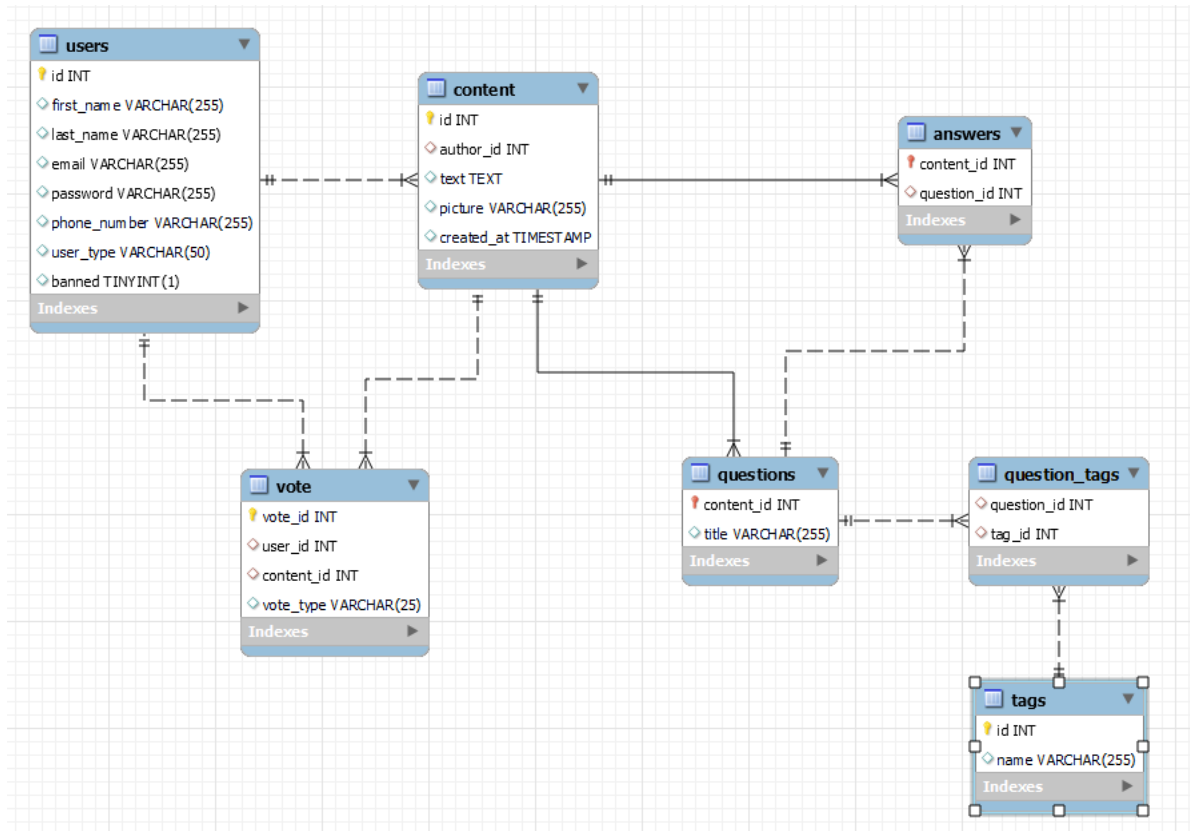


Figure 4: Database Diagram

In the beginning, the schema contained 6 tables:

The **users** table stores information about the users of the system. It contains columns for the user's first name, last name, email, password, phone number, and user type (regular or moderator). The `id` column serves as the primary key for the table, so it is unique for each user.

The **content** table stores content created by the users. It contains columns for the content's author, text, picture, and creation timestamp. The `id` column serves as the primary key for the table, and the `author_id` column is a foreign key that references the `id` column in the `users` table. This is a many-to-one relationship, as several contents may belong to the same user.

Simple StackOverflow

The **questions table** stores information about questions asked in the system. It contains columns for the question's title and the corresponding content_id that refers to the id column in the content table, being a one-to-one relationship, the question table having one additional information, a title.

The **tags table** stores the name of the tags, each one having its own id. Then, the **question_tags** represents the many-to-many relationship between questions and tags. It contains columns for the question_id and tag_id, so it allows multiple tags to be associated with a single question and vice versa.

The **answers table** stores information about answers to questions. The content_id column serves as the primary key for the table, by which we can also access the corresponding content information of an answer. The question_id column is a foreign key that references the question to which the answer was given.

Later, a new table was introduced: **vote table**. The vote_id column is defined as the primary key with auto-incrementing values. The user_id and content_id columns are used as foreign keys referencing the "users" and "content" tables, respectively.

Endpoint Requests

An endpoint request refers to a specific URL on a server that an application can use to access a particular resource or service provided by that server. In other words, an endpoint is a point of entry to a web service or API that enables clients to send requests to the server to access its functionalities or retrieve its data. An endpoint request typically specifies the type of request (GET, POST, PUT, DELETE, etc.) and includes any necessary parameters or data required to perform the requested action.

Here are the endpoints for our simple StackOverflow application, along with their descriptions and some examples:

- POST /users/register would typically expect a JSON payload in the request body containing the user's registration information, such as their name, email, password, and other required fields.

POST /questions is used for creating a new question. The user provides the question's author, title, text, picture, and one or more tags in the request body.

POST /answers is used in a similar manner as for questions.

Input: POST <http://localhost:8080/users/register>

```
{
  "firstName": "Rozsi",
  "lastName": "Veres",
  "email": "vrs@example.com",
  "password": "pwdvrs",
  "phoneNumber": "0747511312",
  "userType": "regular"
}
```

Output: 200 OK

- GET /users is used to retrieve all the users from our database, so the server will respond with an array of user objects.

GET /questions and GET /answers are used accordingly, to obtain all the objects.

Input: GET <http://localhost:8080/users>

Output:

```
[
  {
    "userId": 1,
    "firstName": "John",
    "lastName": "Doe",
    "email": "johndoe@example.com",
    "password": "password123",
```


Simple StackOverflow

```
        "phoneNumber": "1234567890",
        "userType": "regular"
    },
    {
        "userId": 2,
        "firstName": "Jane",
        "lastName": "Doe",
        "email": "janedoe@example.com",
        "password": "password456",
        "phoneNumber": "0987654321",
        "userType": "moderator"
    },
    {
        "userId": 3,
        "firstName": "Rozsi",
        "lastName": "Veres",
        "email": "vrs@example.com",
        "password": "pwdvrs",
        "phoneNumber": "0747511312",
        "userType": "regular"
    }
]
```

- GET /users/{id} is used to retrieve a certain user, given their id.
GET /questions/{id} and GET /answers/{id} work in a similar way.

Input: GET <http://localhost:8080/users/2>

Output:

```
{
    "userId": 2,
    "firstName": "Jane",
    "lastName": "Doe",
    "email": "janedoe@example.com",
    "password": "password456",
    "phoneNumber": "0987654321",
    "userType": "moderator"
}
```

- GET /answers/toQuestion/{id} is used to get all the answers for a specific question, the question's id being specified in the URL.
- PUT /users/{id} is used to modify information about the user. It corresponds to an update operation for which the client has to provide the user with the modified attributes in the request body. The server will update the question successfully and responds with an OK message.

PUT /questions/{id} and PUT /answers/{id} are used in a similar manner.

Input: PUT <http://localhost:8080/users/3>

```
{
  "userId": 3,
  "firstName": "Rozsi",
  "lastName": "Veres",
  "email": "vrs@example.com",
  "password": "newpwdvrs",
  "phoneNumber": "0747511312",
  "userType": "regular"
}
```

Output: 200 OK

- DELETE /questions/{id} is used for deleting a user from the database. The user's id should be provided in the URL.

DELETE /questions/{id} is used for deleting a question, and DELETE /answers/{id} is used for deleting an answer, similar to deleting a user.

Input: DELETE <http://localhost:8080/users/3>

Output: 200 OK

FRONT-END

The front-end of an application is the part of the software that users interact with directly. It typically consists of the user interface (the UI), as well as any other elements that users interact with, such as forms, buttons, and menus.

Front-end development involves creating and implementing the UI and other interactive elements using technologies such as HTML, CSS, and JavaScript. It also involves ensuring that the UI is responsive and accessible across a wide range of devices and screen sizes. The front-end should provide an intuitive user experience and it should also be visually appealing and consistent. Additionally, it should be responsive and accessible, ensuring that users can access and use the application across a variety of devices.

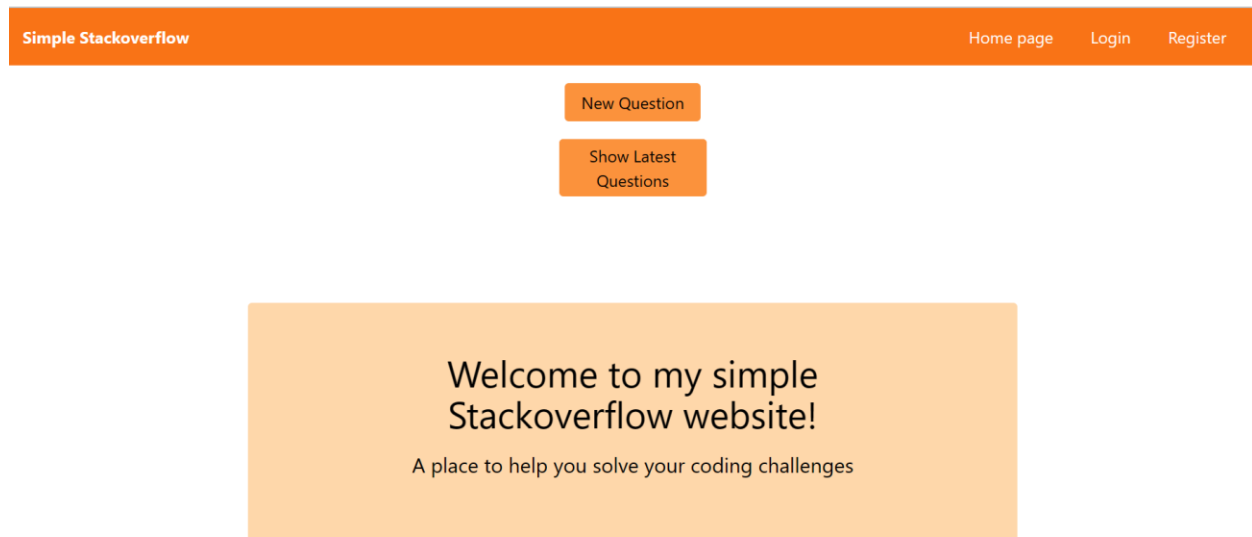


Figure 5: The Home Page of my web application

Architecture

In front-end development, **module directory structure** is a way of organizing code files that are related to a specific feature or component of an application. The goal of module directory structure is to create a clear and modular architecture that makes it easy to maintain and modify the codebase over time.

Within each module directory, the files are typically organized into subdirectories that reflect the different components of the feature or module. In our case, for authentication, we have a module directory called “auth” which has subdirectories for login and registering, with each subdirectory containing the relevant code.

Also, our system has a **presentation layer**. The only responsibilities of this layer are to present and delegate. In other words, it presents the UI and delegates the user’s actions to the **service layer**. It knows what to display and what to do, but it does not know how the user’s interactions should be handled. In order to be able to use the services, dependency injection is used.

In Angular, the constructor is used to inject dependencies into a class. By including a parameter of a specific type in the constructor, Angular's dependency injection system can automatically provide an instance of that dependency to the class when it is instantiated.

As seen in Figure 6, the `questionsService` parameter is used to inject an instance of the `QuestionsService` class into the current class, which is the `QuestionsComponent`. This allows the current class to access the methods and properties of the `QuestionsService` class, which is needed to perform certain actions, such as retrieving all questions from the database through the back-end server or adding a new question to the database.

```
export class QuestionsComponent implements OnInit {  
  questions: Question[] = [];  
  
  constructor(private questionsService: QuestionsService) {  
  
  }  
}
```

Figure 6: Dependency Injection

Angular follows the **component-based architecture**. This architecture provides developers with a clear separation of concerns and allows for greater reusability and maintainability of code.

In Angular, the component plays a similar role to the controller in the MVC pattern, as it manages the view and interacts with the model. Each component is a self-contained unit of functionality that can be composed together to form complex applications.

We have a few components representing users, others for representing questions and answers. I used some components such as login and register which is needed for the authentication part.

We can also say that our application has a three tiered architecture, which is divided into services, container components and presentation components. **Services** contain all business logic in our application. They process data for components and then components are the part of the infrastructure that display information. In my application, I have a questions and answers services, and an authentication service as well.

Container components are responsible for managing the data and state of the application, and for communicating with the services that provide the data. They typically have more complex logic and are responsible for handling user interactions and managing the overall flow of the application. Container components are also sometimes referred to as smart components, as they contain the intelligence of the application.

On the other hand, **presentation components** are responsible for rendering the user interface and displaying the data that is provided by the container components. They are typically simpler than container components and focus on the visual aspects of the application. Presentation components are also sometimes referred to as dumb components, as they are not responsible for any complex logic or data management.

HTML files are an essential part of presentation components, as they define the structure and layout of the user interface. They contain the HTML markup and Angular directives that bind data to the template and control the rendering of the UI.

The user's profile page that manages the user's data and communicates with the back-end to fetch and update the data is a container component. This component is responsible for managing the user's data, and handling user interactions such as updating the profile information. The component that manages the questions, communicates with the back-end as well to add retrieve items. This component is responsible for managing the state of the questions list and handling user interactions such as adding a new question.

The component for registration displays a form where the user can input their name, email address, phone number and other basic information. This is rather a presentation component which would receive the user data and display it in a visually attractive way. A question component that displays the title user, text and image of a product is a presentation component as well.

Routing

In Angular, app routing is the process of defining the navigation paths and URLs that we can use to move between different pages within an application. This is accomplished using the Angular Router module, which provides a flexible way to manage application state and handle user navigation.

To set up app routing in an Angular application, we must define first a set of routes that correspond to different pages in our application. This is usually done in a separate routing module, which is responsible for defining the routes and configuring the router.

Each route is defined as a JavaScript object that contains information about the URL path and the component that should be displayed when the route is activated. Once the routes have been defined, we need to configure the router to use them. This is done by importing the routing module into the app module and adding it to the app's list of imports together with the declaration of the components which are going to be used.

With app routing set up, users can navigate between different views in our application by clicking on links. The router will automatically handle the navigation and display the appropriate views and components based on the current route and authentication privileges.

Now the paths will be presented.

```
path: '', component: QuestionsComponent
```

The line above corresponds to our main page, seen in Figure 5. Here, the user can see the most recent questions posted, scroll among them being able to see the answers given so far or answer a question if certain links are followed. The user can also have a new question for which we introduced another link.

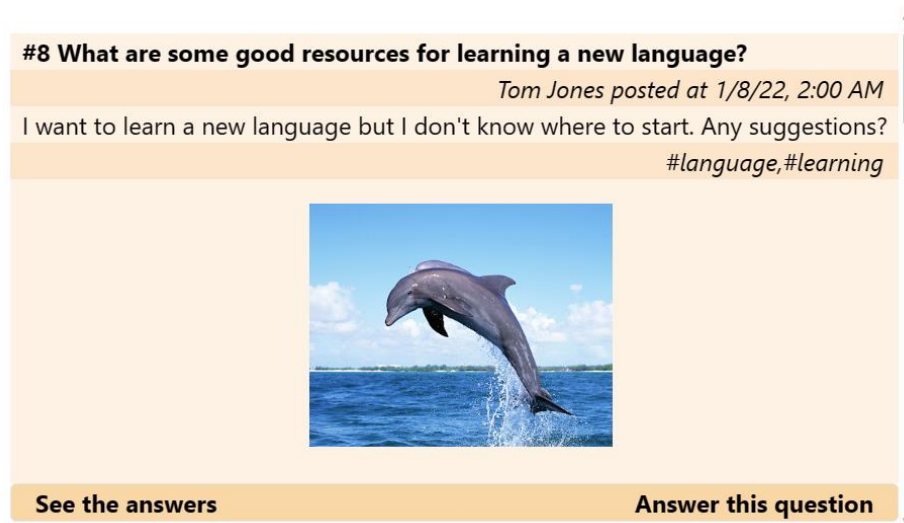


Figure 7: The way that questions are displayed.

```
path: 'register', component: RegisterComponent
path: 'login', component: LoginComponent
```

These lines contain a registration and login form for the user, whose info is connected to a Firebase project, which provides a good authentication service for our application. For these forms, see Figure 8 and Figure 9.

```
path: 'questions/new', component: NewQuestionComponent
path: 'answers/new', component: NewAnswerComponent
```

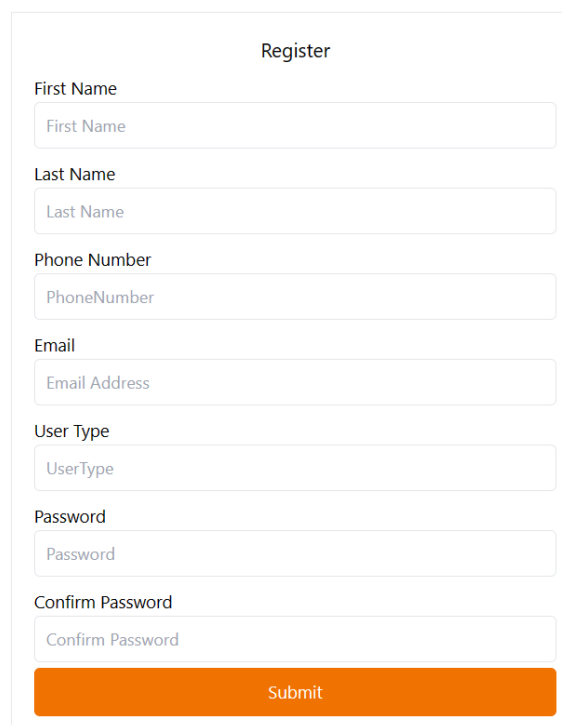
The above lines are necessary for providing the path and components for new questions and answers. The user can fill in a form for that and then submit it as seen in **Error! Reference source not found.** and **Error! Reference source not found.** If the user is logged in and all the textboxes were filled out correctly, the new question or answer will be saved to the database.

```
path: 'user/profile', component: MyProfileComponent}
path: 'user/answers', component: MyAnswersComponent}
path: 'user/questions', component: MyQuestionsComponent}
```

These paths can be accessed by a logged in user. A logged in user can see their own profile information, can edit their data and view their own questions and answers. The questions and answers are displayed in a similar manner as in Figure 7.

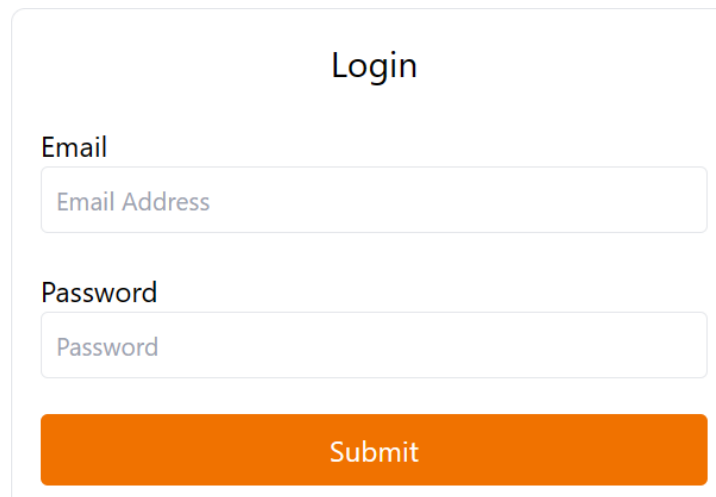
```
path: 'questions/:id/answers', component: AnswersComponent
```

Finally, but not least, this path corresponds to a page where all the answers of a certain question can be read. For this, we are going to use the id of the question, which can be passed to the new page in this way.



The image shows a registration form titled "Register". It contains several input fields: "First Name", "Last Name", "Phone Number", "Email", "User Type", "Password", and "Confirm Password". Each field has a placeholder text that matches the field's label. At the bottom of the form is a large orange "Submit" button.

Figure 8: Registration Form for a New User



The login form is centered on the page. It has a title 'Login' at the top. Below it are two input fields: 'Email' with a placeholder 'Email Address' and 'Password' with a placeholder 'Password'. At the bottom is an orange 'Submit' button.

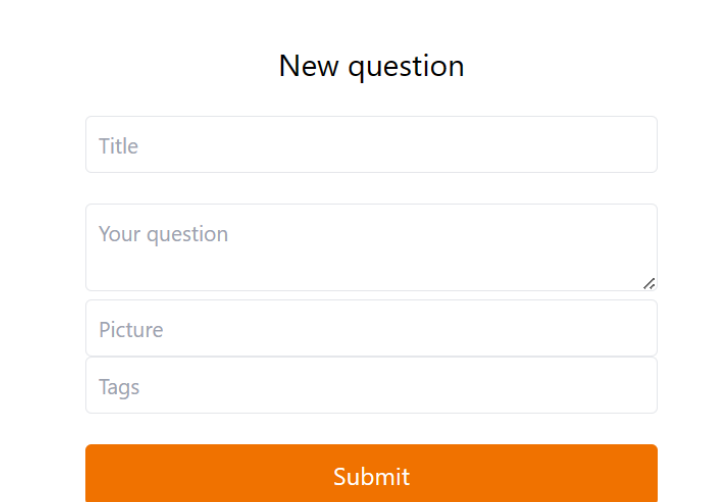
Login

Email

Password

Submit

Figure 9: Login Form



The 'New question' form is centered. It has a title 'New question'. Below it are four input fields: 'Title', 'Your question' (with a text area icon), 'Picture', and 'Tags'. At the bottom is an orange 'Submit' button.

New question

Title

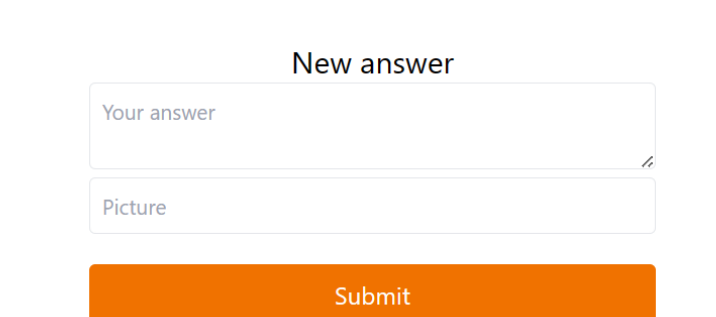
Your question

Picture

Tags

Submit

Figure 10: New Question Form



The 'New answer' form is centered. It has a title 'New answer'. Below it are two input fields: 'Your answer' (with a text area icon) and 'Picture'. At the bottom is an orange 'Submit' button.

New answer

Your answer

Picture

Submit

Figure 11: New Answer Form