ASSIGNMENT 3 – DOCUMENTATION

1. Functional requirements

No action is possible if the user is not logged in *(cannot access the application neither via url)*.
The user's passwords are stored in the database encrypted.
The users can see the other users and can update their data.

Feature 1:

Users are able to ask questions. Each question has an author, title, text, creation date & time, picture and one or more tags (which can be added/removed). If an appropriate tag does not exist, the user is able to create one and then add it to the question.
The list of questions is displayed sorted by creation date. The most recent question is displayed first.
Questions may be edited or deleted by their author.
The user is able to filter questions by tag, via a text search, via users or for their own questions. The text search should check the question title.

Feature 2:

Each question may be answered one or more times by any user (including the original author).
Each answer must have an author, text, picture and creation date & time.
Answers may be edited or deleted by their author.
When displaying a question individually, the list of answers of it is also displayed.

Feature 3:

Users may vote questions and answers (*upvote/downvote*).
Each user may only vote once on each question or answer. Users cannot vote on their own answers or questions.
On each voted question or answer, the vote count is displayed, which can be negative. (*vote count = upvote count – downvote count*)
The answers for a question are sorted by their vote count. Answers with the highest vote count are displayed first.

Feature 4:

Based on upvotes and downvotes, the system computes a user score with the following rules:

Each user starts with 0 points.

Users gain points when:
• Their question is voted up (+2.5 points per vote),
• Their answer is voted up (+5 points per vote).

Users lose points when:
• Their question is voted down (-1.5 points per vote),
• Their answer is voted down (-2.5 points per vote),
• They down vote an answer of another user (-1.5 point).
• The user score is displayed next to the author's name on each question / answer.
• The score can be negative.

Feature 5:

Moderators are users with special privileges. They are able to:
• Remove questions or answers if inappropriate
• Edit any question or answer on the site
• Ban users from the site indefinitely in case of bad behaviour
• The users can be unbanned by the moderator.

Banned users see a message indicating that they were banned when trying to login and are unable to perform any other actions. (*cannot access the application neither via url*)
The banned users receive an e-mail & sms when they are banned.

2. Design and Implementation

I. Database Design

The database design plays a crucial role in the development of any software solution, serving as the foundation upon which the entire system is built. It encompasses the organization, structure, and relationships of the data that will be stored and managed by the application. For these reasons, in order to fulfill the requirements provided, I started first by modelling my database with the following tables:

1. *User*:

Attributes:
user_id INT (Primary Key)
username VARCHAR(255) (Unique)
email VARCHAR(255) (Unique)
password VARCHAR(255) - *hashed*
first_name VARCHAR(255)

last_name VARCHAR(255)
position INT (0 – regular user, 1 – moderator)
banned INT (0 – false, 1 – true)
phone VARCHAR(255)

All the attributes must be *not-null*.

2. *Question*:

Attributes:
q_id INT (Primary Key)
user_id INT (Foreign Key referencing User)
title VARCHAR(255) (Not-null)
text VARCHAR(255) (Not-null)
creation_date DATE (Not-null)
picture VARCHAR(255) (the location of the image)

Relationships:
One-to-Many relationship with *User* (one user can ask multiple questions)

3. *Tag*:

Attributes:
tag_id INT (Primary Key)
name VARCHAR(255)

4. *Questions_Tags*:

Attributes:
q_id INT (Foreign Key referencing Question)
tag_id INT (Foreign Key referencing Tag)

Relationships:
Many-to-Many relationship between *Question* and *Tag* (A question can have multiple tags, and a tag can be associated with multiple questions) -> This relationship was divided into two *One-To-Many relationships* through this table to respect the *First Normalization Form*

5. *Answer*:

Attributes:
a_id INT (Primary Key)

q_id INT (Foreign Key referencing Question)
user_id INT (Foreign Key referencing User)
text VARCHAR(255) (Not-null)
creation_date DATE (Not-null)
picture VARCHAR(255)

Relationships:
One-to-Many relationship with *User* (one user can provide multiple answers)
One-to-Many relationship with *Question* (one question can have multiple answers)

6. *Vote_Answer*:

Attributes:
va_id INT (Primary Key)
a_id INT (Foreign Key referencing Answer)
user_id INT (Foreign Key referencing User)
up_or_down INT (1 – upvote, -1 – downvote)

Relationships:
One-to-Many relationship with *User* (one user can vote multiple answers)
One-to-Many relationship with *Answer* (one answer can have multiple votes)

7. *Vote_Question*:

Attributes:
vq_id INT (Primary Key)
q_id INT (Foreign Key referencing Question)
user_id INT (Foreign Key referencing User)
up_or_down INT (1 – upvote, -1 – downvote)

Relationships:
One-to-Many relationship with *User* (one user can vote multiple questions)
One-to-Many relationship with *Question* (one question can have multiple votes)

8. *Salt* – used in hashing the user's passwords:

Attributes:
s_id INT (Primary Key)
user_id INT (Foreign Key referencing User)
salt V ARCHAR(255) (Not-null)

Relationships:

One-to-One relationship with *User* (one user can have their password encoded with only 1 salt)

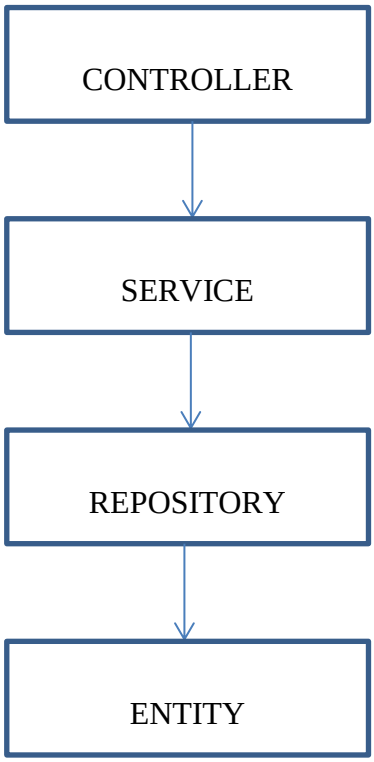
II. Back-end

The next part of the developing process consists of building the backend which serves as the backbone of the software solution. It is responsible for processing data, implementing business logic, and facilitating communication between the front end and the database.

In this section, I will delve into the architecture, functionality, and implementation details of the backend components that power the application:

This component is built using *Spring Boot*, a powerful framework for rapidly developing Java-based applications and the correctness of the backend was tested using *Postman*. Leveraging the flexibility of Spring Boot, I have designed the **backend architecture** following a *layered approach*, ensuring modularity, maintainability, and scalability.

In addition, for sending emails, I used the *core Java mail library* and, for sending SMS, I used *Twilio API*.



1. *Controller Package*

- serves as the entry point for incoming requests from clients (received from front-end). It handles request mapping, input validation, and response generation, acting as the interface between the *client* and the *services*
- maps incoming *HTTP* requests to specific handler methods based on their URLs and HTTP methods (GET (retrieve data), POST(create new data), PUT(update existing data), DELETE(delete data))
- in order to apply CRUD operations on *User*, *Question*, *Answer*, *Tag* etc. there exists a controller for each of them: UserController, QuestionController, AnswerController, TagController etc.

-> they treat: *getAll*, *getById*, *deleteById*, *update*, *add*, *filter* etc. requests from those who use the website

2. *Service Package*

- encapsulates the business logic of our application. It orchestrates interactions between different components, performs data processing, and enforces business rules
- in this application, it usually processes the data retrieved from the database, using the *Repository* package or from services of the other entities
- there exists a service class for each main entity: *AnswerSerivce*, *TagService*, *QuestionService*, *UserService*, *VoteAnswerService*, *VoteQuestionService*

3. *Repository Package*

- provides an abstraction for data access operations. It interacts with the underlying database, executing CRUD (Create, Read, Update, Delete) operations and translating data between the application and the database entities
- JPA allows entities to be managed as Java objects while transparently persisting them to the database and that's why it generates database queries based on method names defined in repository interfaces, reducing the need for manual query creation and enhancing developer productivity

4. *Entity Package*

- defines the structure and relationships of the data entities stored in the database. Each entity corresponds to a database table, encapsulating attributes and behaviors relevant to that entity
- there exists an entity class for each table of the database: *User*, *Question*, *Tag*, *Answer*, *VoteAnswer*, *VoteQuestion*, *Salt*, except for the *questions_tags* -> this table is represented in the *Question* entity class with a many-to-many relationship annotation through the Spring Boot framework

- the other relationships are represented in the same way, with an annotation in one of the involved entities
- *Question*: @PrePersist was used to automatically put the creation time as the current local time
- in order to be able to return a user with its associated score, a new entity class *UserScoreDTO* was created

III. Front-end

Lastly, the users interact directly with the application through the front-end. *Angular* is used as the primary framework for building this part. It's using TypeScript, which provides type safety and improved developer experience.

Angular Router is employed for handling navigation within the Stack Overflow application. It allows for defining routes and mapping them to components. *Routing Guards* are also used for ensuring that someone who isn't logged in can't access the application through urls (a user is *logged in* if the currentUser in *session storage* is set): the user can access only the *Login* and *Register* pages; when a user logs in, they are redirected to the page with the *Users*.

The styling is defined using *SCSS (Sass CSS)*. SCSS provides powerful features such as variables, nesting, and mixins, which help maintain a consistent visual style across the project and make styling more modular and reusable. Additionally, *Angular Material* library is utilized for implementing and designing: it provides a set of pre-designed UI components such as buttons, forms, cards, and navigation elements, which offer a consistent and visually appealing user experience.

The project follows a **component-based architecture** where each UI element is encapsulated within its own component. Each component follows a consistent structure comprising three main elements: *HTML*, *SCSS*, and *TypeScript*.

At component-lever, a **layered architecture** can be detected: when accessing the input fields (the correspondents to *Controllers* in back-end), data is acquired with the help of *Angular services* (similar to *Services* in back-end). Services in Angular are used for encapsulating reusable business logic, data manipulation, and communication with external resources such as APIs (in this application: *http://localhost:8080*). They help keep components lean by moving common functionality out of components and promoting code reusability and maintainability. In order to be able to transmit the required data in back-end (through *JSON*), these services work with certain data structures -> *entities*.

Structure:

The application project is organized into several parts, each serving a specific purpose in the application's architecture:

1. Components

- *LoginComponent*: Login Page-> It presents a form interface with input fields for entering user details such as username and password; after login, the user is redirect to the Users List

- *RegisterComponent*: Register Page-> It presents a form interface with input fields for entering user details such as username, password, e-mail, first name, last name and phone number; after registering, the user (with role: regular user) is redirected to the login page

- *UserListComponent*: Users Page-> is responsible for displaying a list of users in a tabular format. It provides an interface for users to view and interact with user data, such as viewing users' details, editing current user information, and deleting users; it also allows logging out and going to the questions page; moderators are allowed to ban/unban users

- *UpdateUserComponent* : Update User Page -> It presents a form interface with input fields for changing current user's details such as username and password; when the user's details are updated, the user is redirected to the user list page

- *QuestionListComponent*: Questions Page -> is responsible for displaying all existing questions in a tabular format. It provides an interface for users to view and interact with question data, such as viewing questions' details, editing/deleting their own questions (or any question if moderator), viewing the answers of a question, adding an answer to a question or liking/disliking a question; it also allows logging out, going to the users page and filtering the questions to be displayed based

- *QuestionComponent*: It presents a form interface with input fields for entering question details such as title, text, picture or tags; if a user can't find the desired tag, they can add another one; after submitting the (updated) question, the user is redirected to the question page; there are two pages that use this component:

-> Add Question Page (*AddQuestionComponent*)

-> Update Question Page (*UpdateQuestionComponent*)

- *QuestionWithAnswersComponent*: Question with Answers Page -> is responsible for displaying the selected question along with all its existing answers in a tabular format. It provides an interface for users to view the current question's details, to view the answers of a question, add an answer to the question, delete/update own answer (or any if moderator) or liking/disliking an answer; it also allows logging out, going to the users page and going to the questions page

- *AnswerComponent*: It presents a form interface with input fields for entering answer details such as text or picture; after submitting the (updated) answer, the user is redirected to the question with answers page; there are two pages that use this component:

-> Add Answer Page (*AddAnswerComponent*)

-> Update Answer Page (*UpdateAnswerComponent*)

2. Services

- *Entity usage*: Entities represent the core data structures used within the application. They encapsulate the properties and behaviors of domain-specific objects such as questions, users, answers, tags, votes for an answer/a question or are responsible for *Routing Guards* (*auth.guard.ts*).

User Entity: The User entity represents individual users within the project. It defines properties such as userId, username, password and any other relevant attributes associated with users. User entities are utilized in *UserService* for CRUD (Create, Read, Update, Delete) operations and data manipulation, and also for login, logout and keeping track of the current user.

Question Entity: The Question entity represents the individual questions. It defines properties such as questionId, title, text and any other relevant attributes associated with questions (including user of the question). Question entities are utilized in *QuestionService* for CRUD (Create, Read, Update, Delete) operations and data manipulation, and also for keeping track of the currently selected question.

Answer Entity: The Answer entity represents the individual answers. It defines properties such as answerId, text and any other relevant attributes associated with answers (including user of answer and the question itself). Answer entities are utilized in *AnswerService* for CRUD (Create, Read, Update, Delete) operations and data manipulation, and also for keeping track of the currently selected answer.

Tag Entity: The Tag entity represents the individual tags. It defines properties such as tagId and name of the tag. Tag entities are utilized in *TagService* for Create and Read operations and data manipulation.

Vote Answer Entity: The Vote Answer entity represents the upvotes/downvotes of answers. It defines properties such as voteAnswerId, user of the vote and the voted answer. They are utilized in *VoteAnswerService* for Create and Read operations (upvote, downvote, getting scores and votes).

Vote Question Entity: The Vote Question entity represents the upvotes/downvotes of questions. It defines properties such as voteQuestionId, user of the vote and the voted question. They are utilized in *VoteQuestionService* for Create and Read operations (upvote, downvote, getting scores and votes).

3. App Module

The root module of the Angular application. It imports and configures all *modules* (ex: *MatCardModule*, *MatInputModule*, *MatSelectModule* modules from Angular Material), *components*, *services*, and *dependencies* needed to bootstrap and run the application.

4. App Routing

Defines the application's routing configuration using Angular Router. It maps URL paths to Angular components, enabling navigation between different views and features of the application.

Routes:

Home (‘/’) – the Login Page

'users/addUser' – the Register Page

‘users/getAll’ – the Users Page

‘users/updateUser’ – the Update User Page

'questions/getAll' – the Questions Page

'questions/addQuestion' – the Add Question Page

'questions/updateQuestion' – the Update Question Page

‘answers/getAll’ – the Question with Answers Page

'answers/addAnswer' – the Add Answer Page

'answers/updateAnswer' – the Update Answer Page