Stack Overflow Assignment Documentation

Table of Contents

[1. Introduction 3](#_Toc129516782)

[2. Technology 4](#_Toc129516783)

[3. Use Cases 4](#_Toc129516784)

[4. Architecture 5](#_Toc129516785)

[4.1 The Layered Architecture 5](#_Toc129516786)

[4.2 Spring Boot Architecture 5](#_Toc129516787)

[4.3 Spring Boot Flow Architecture 6](#_Toc129516788)

[5. Diagrams 7](#_Toc129516789)

[6. Database 7](#_Toc129516790)

[7. Endpoints 8](#_Toc129516791)

[References 8](#_Toc129516792)

# Introduction

Stack Overflow is a very renowned website among developers all around the world. Whenever you have a bug in your code which you cannot solve, or you don’t know what causes it, the first place you search for a solution is Stack Overflow. This project resembles the famous web application with a very reduced functionality, implementing the bare minimum operations that a forum should be able to perform. This forum encourages correct and complete answers by rewarding users that try to provide a helpful answer with a score; each question and answer can get upvotes and downvotes from the community based on the impact it has on other developers.

The application should have some key features that resemble the basic functionality of a forum in which people can communicate effectively and safely:

***Feature 1***

* Users shall be able to ask questions. Each question must have an author, title, text, creation date and time, picture and one or more tags. If an appropriate tag does not exist, the user must be able to create one.
* The list of questions shall be displayed, sorted by creation date. The most recent question should be displayed first.
* Questions may be edited or deleted by their author.
* The user must be able to filter questions by tag, via a text search, via users or for his 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 and time.
* Answers may be edited or deleted by their author.
* When displaying a question individually, the list of answers must also be displayed.

***Feature 3***

* Users may vote questions and answers (upvote and downvote, like and dislike).
* 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 must be displayed (vote count = upvote/like count -downvote/dislike count). The vote count can be negative.
* The answers for a question must be sorted by their vote count. Answers with the highest vote count must be displayed first.

***Feature 4***

* Based on upvotes and downvotes or likes and dislikes, the system must compute a user score with the following rules:
  + Each user starts with 0 points.
  + Users lose points when:
    - Their question is voted down (-1.5points per vote),
    - Their answer is voted down (-2.5points per vote),
    - They down vote an answer of another user (-1.5point).
  + Users gain points when:
    - Their question is voted up (+2.5points per vote),
    - Their answer is voted up (+5points per vote).
* The user score shall be 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 should be 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 behavior,
  + Unban users.
* Banned users must see a message indicating that they were banned when trying to login and should be unable to perform any other actions (cannot access the application via URL).
* The banned users must receive an e-mail and SMS when they are banned.

# Technology

For the frontend, the technology used is *React* framework and for the backend, *Java* with *Spring Boot* framework and *MySQL* relational database management system. I used *IntelliJ IDE* because of its full support for all the languages and tools that I used for developing the application. IntelliJ makes it very easy to structure the project, atomate repetitive tasks, it has a very good intellisense for *Java* and it also supports *SQL* language for managing databases directly from the IDE.

For frontend I used ReactJS with Redux for managing the state and Axios for performing the requests to the backend.

# Use Cases

A [*UML*](https://en.wikipedia.org/wiki/Unified_Modeling_Language)*use case diagram* is the primary form of system/software requirements for a new software program under developed. Use cases specify the expected *behavior (what)*, and not the exact method of making it happen (how). Use cases once specified can be denoted both textual and visual representation (i.e., use case diagram). A key concept of use case modeling is that it helps us design a system from the end user's perspective. It is an effective technique for communicating system behavior in the user's terms by specifying all externally visible system behavior. (Visual Paradigm, n.d.)

For all the actions a user can perform, he must be logged in for privilege separation between a regular user and a moderator.

Diagram

Description automatically generated

Fig. 1 Use Case Diagram

# Backend Architecture

## 4.1 The Layered Architecture

This application uses the *Layered Architecture* pattern. Layered Architecture is a solid general-purpose pattern, making it a good starting point for most applications. Components within the layered architecture pattern are organized into *horizontal layers*, each layer performing a specific role within the application (e.g., presentation logic or business logic). Each layer of the layered architecture pattern has a *specific role and responsibility* within the application. Each layer in the architecture forms an *abstraction* around the work that needs to be done to satisfy a particular business request. One of the powerful features of the layered architecture pattern is the *separation of concerns* among components. Components within a specific layer deal only with logic that pertains to that layer. (Richards, 2017)

## 4.2 Spring Boot Architecture

Spring Boot is a module of the Spring Framework. It is used to create stand-alone, production-grade Spring Based Applications with minimum effort. Spring Boot follows a layered architecture in which each layer communicates directly with the layer situated directly above or below itself. *Spring Boot Architecture* has *four* layers: ***Presentation Layer***, ***Business Layer, Persistence Layer*** and ***Data Access Layer***.

**Presentation Layer**: handles the HTTP requests, translates JSON parameters to objects, and authenticates and delegates requests to the business layer. This is the *frontend* of the application.

**Business Layer**: also called the Service Layer, it handles the ***business logic.*** It consists of service classes and uses services provided by the data access layer. It also performs **authorization** and **validation.**

**Persistence Layer:** contains all the ***storage logic*** and translates business objects to database rows and vice versa.

**Database Layer:** here are all the ***CRUD (create, retrieve, update, delete)*** operations performed.

(JavaTptoint, 2021)

Diagram

Description automatically generated

Fig. 2 Spring Boot Architecture (JavaTptoint, 2021)

## 4.3 Spring Boot Flow Architecture

The *Spring Boot Flow Architecture* describes the data flow through the application layers and the processing it goes through. In a typical scenario, a client sends an HTTP request to the controller layer, then the controller maps the request and handles it and calls the appropriate service if needed. The service layer then performs the required business logic on the data mapped to the JPA with model(entity) classes. Finally, a JSP page is returned to the client if no error occurred.

Diagram

Description automatically generated

Fig. 3 Spring Boot Flow Architecture (JavaTptoint, 2021)

# Diagrams

# Database

MySQL database management system was used for storing the data used by the application. The database schema is called *stackoverflow* and it contains 6 tables: *User*, *Post*, *Question, Answer, Tag* and *QuestionTag*.

There is a *one-to-many* relationship between **User** and **Post** entities. One user can post multiple questions/answers, but there is no question/answer without an author, which is a user.

The *Question* and *Answer* tables are the children of the Post table, but because *inheritance* cannot be represented in a relational database explicitly, I used the *post\_id* attribute as a ***primary key***, as well as a ***foreign key*** too, to represent the ”*IS-A*” relationship between the *Post* and *Question* and *Answer* entities. I chose to store the common attributes of the questions and answers in a separate table to avoid duplication. An answer is just an alias for a post, whereas a question also has title and one or more tags.

A question has one or more tags, so there is a *many-to-many* relationship between *Question* and *Tag* tables. This is modeled through an auxiliary table *QuestionTag* which only stores the *post\_id* and *tag\_id* attributes as ***foreign keys*** and they represent the ***primary key***.

Graphical user interface

Description automatically generated

Fig. 4 Entity Relation Diagram

# Endpoints

The endpoints of the application are implemented for User, Question and Answer entities. The endpoints provide all CRUD operations on the entities.

**User:**

* Create: <http://localhost:8080/users>
* Read: <http://localhost:8080/users/>all, <http://localhost:8080/users?user_id=>
* Update: <http://localhost:8080/users>?user\_id=
* Delete: [http://localhost:8080/users?user\_id](http://localhost:8080/users?user_id%7d)=

**Question:**

* Create: <http://localhost:8080/questions>
* Read: <http://localhost:8080/questions/all>
* Update: <http://localhost:8080/questions?question_id=>
* Delete: <http://localhost:8080/questions?question_id=>

**Answer:**

* Create: <http://localhost:8080/answers>
* Read: [http://localhost:8080/answers](http://localhost:8080/answers/)?question\_id=
* Update: <http://localhost:8080/answers?>question\_id=qid&&answer\_id=aid
* Delete: <http://localhost:8080/answers?>question\_id=qid&&answer\_id=aid

# Frontend Architecture

A React application starts with a single root component which is built using one or more components. Each component can be nested with other components. The React application can include third party components for specific purposes like routing (React Router DOM), animation, state management (Redux).

A screenshot of a cellphone

Description automatically generated with low confidence

Fig. 4 ReactJS Application Architecture (tutorialspoint, n.d.)

The application has the following tree structure:

* public
* src:
  + actions
  + api
  + assets
  + components
  + reducers

The ***src*** folder holds all the code of the application. In the ***actions*** folder are all the actions that can be performed on the model data from the frontend application (CRUD operations, authentication, etc). The ***api*** folder contains the gateway which communicates directly with the API of the backend application, here are defined all the requests that are made to the backend. The ***assets*** folder contains all the assets for the frontend like icons, logos and other pictures used in components. The ***reducers*** folder contains all the reducers that take the data received from the backend using Axios and transforms and stores the data in the Redux store to make it available to all the components in the application for later use. Here is processed the data that comes in JSON format, and it is stored in the *local storage* of the application. Finally, the ***components*** folder holds all the functional components that define the UI, like pages or smaller components that are reused throughout the application.

# Routing

The is done using the { BrouseRouter } component from the ‘react-router-dom’ package. When the application starts, the client displays to the user the homepage with the URL: <http://localhost:3000> . If the user tries to do any action, like asking a question, voting or any other action that requires being authenticated, the user is redirected to the login page. Here he has the option of signing in <http://localhost:3000/login> or creating a new account <http://localhost:3000/signup> .

All the questions are displayed on the questions page and on the home page too. When the user clicks on the title of a question, he is redirected to the <http://localhost:3000/questions/qid> URL, where *qid* represents the question ID. If the user clicks on the ‘Ask Question’ button, then he is redirected to the <http://localhost:3000/questions/ask-question> URL. Here he can enter the details of the question and then he must click the ‘Review your question’ button to post the question and then he is redirected to the <http://localhost:3000/questions> URL.

The ‘Tags’ page <http://localhost:3000/tags> displays all the tags from the asked questions.

The ‘Users’ <http://localhost:3000/users> page displays all the users that are registered on the website. If the currently logged in user is a moderator, then the appropriate options are shown too. A user can also click on the avatar of a user, and he will be redirected to the user profile page of that user <http://localhost:3000/users/uid> , where the *uid* is the user ID.

# References

JavaTptoint. (2021). *JavaTpoint*. Retrieved from JavaTpoint: https://www.javatpoint.com/spring-boot-tutorial

Richards, M. (2017). *Software Architecture Patterns.* Sebastopol: O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA.

tutorialspoint. (n.d.). *tutorialspoint*. Retrieved from tutorialspoint: https://www.tutorialspoint.com/reactjs/reactjs\_architecture.htm

Visual Paradigm. (n.d.). *Visual Paradigm*. Retrieved from Visual Paradigm: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/