**A3**

**STACK BETTERFLOW**

**Cimpan Stefan**

**A1**

**Backend**

**Cimpan Stefan**

1. Introduction – talk about the project, key features etc...

2. Technology – present the technologies that you will use, IDE etc...

3. Use case diagrams – describe what is a use case diagram, and present the use case diagrams for your application

4. Architecture – present the architecture of your application, talk about each layer

5. Package diagram – describe what is a package diagram and show the package diagram of you application, talk about the entities

6. Class diagram – describe what is a class diagram and show the class diagram of you application, talk about the entities

7. Database Diagram – present the database diagram and talk about it and the relations

8. Endpoints requests – describe the endpoints of your application. For 5 of them present the input & output of them

**Introduction**

Welcome to my MERN-based Stack Overflow-like application! This project aims to provide a platform for developers and programming enthusiasts to connect, collaborate, and seek help within a vibrant community. Inspired by the renowned Stack Overflow platform (BETTER STACK), I have developed a robust application that leverages the power of the MERN stack (MongoDB, Express.js, React.js, and Node.js) to deliver a seamless and feature-rich experience.

Key Features:

1. User Authentication and Profiles: Our application allows users to create accounts, log in securely, and manage their profiles. Users can customize their profiles, add a profile picture, and provide information about their skills and interests.
2. Question and Answer System: The heart of our application lies in the question and answer system. Users can post questions related to various programming languages, frameworks, or technologies. They can provide detailed descriptions and tag their questions for better categorization. Other users can then provide answers, suggestions, and comments.
3. Voting and Reputation System: Our application incorporates a voting system where users can upvote or downvote questions and answers based on their quality and helpfulness. This feature helps in highlighting the most valuable and relevant content, ensuring that the best answers rise to the top. Users also gain reputation points based on their contributions, fostering a sense of recognition and expertise within the community.
4. Tagging and Search Functionality: To facilitate easy discovery of relevant content, our application includes tagging functionality. Users can assign tags to their questions and explore questions by specific tags. Additionally, a powerful search feature enables users to find questions based on keywords, tags, or specific criteria.
5. Notifications and Activity Feeds: Our application keeps users informed about activities relevant to them through notifications and activity feeds. Users receive notifications when someone comments on their questions or answers, upvotes their content, or mentions them. The activity feed provides a real-time stream of updates from the users and topics they follow.
6. Reputation-based Privileges: As users accumulate reputation points through their contributions, they unlock certain privileges within the application. These privileges may include the ability to edit questions and answers, moderate content, or access advanced features. This system encourages active participation and rewards valuable contributions.
7. Responsive and Intuitive UI: We have built a user-friendly interface using React.js that adapts seamlessly to different devices and screen sizes. The application provides a smooth and intuitive experience, allowing users to navigate through questions, answers, and discussions effortlessly.

**Technology Stack:**

My MERN-based Stack Overflow-like application utilizes a powerful combination of technologies to deliver a robust and scalable solution. I used:

1. MongoDB: MongoDB is a popular NoSQL database that we use as the persistent storage for our application. It provides flexibility in handling unstructured data and allows for efficient querying and retrieval of information.
2. Express.js: Express.js is a fast and minimalist web application framework for Node.js. We leverage Express.js to build the server-side of our application, handling HTTP requests, routing, and middleware functionality.
3. React.js: React.js is a widely adopted JavaScript library for building user interfaces. We utilize React.js to create a dynamic and responsive front-end interface, providing users with an interactive experience and seamless navigation.
4. Node.js: Node.js is a powerful JavaScript runtime environment that enables server-side development. With Node.js, we build the back-end of our application, handle business logic, interact with the database, and communicate with the front-end.
5. HTML5, CSS3, and JavaScript: These core web technologies form the foundation of our application's front-end. HTML5 provides the structure of the webpages, CSS3 styles them, and JavaScript adds interactivity and dynamic behavior.
6. RESTful API: Our application follows the principles of Representational State Transfer (REST) architecture. We design and implement a RESTful API that enables communication between the front-end and back-end, facilitating data exchange and manipulation.
7. IDEs and Tools: For development, we recommend using popular integrated development environments (IDEs) such as Visual Studio Code, WebStorm, or Atom, which provide excellent support for JavaScript, React.js, and Node.js development. These IDEs offer features like code highlighting, auto-completion, debugging, and version control integration, streamlining the development process.
8. Additional Libraries and Packages: To enhance the functionality and efficiency of our application, we utilize various libraries and packages available in the JavaScript ecosystem. These may include libraries for authentication and authorization, form validation, state management (such as Redux), routing (such as React Router), and UI components (such as Material-UI or Bootstrap).

**Use case diagrams**

A use case diagram is a visual representation of the interactions between actors (users or external systems) and the system under consideration. It illustrates the various use cases (functionalities) of the system and shows how actors interact with those use cases.

A screen shot of a diagram

Description automatically generated with low confidence

* The user initiates the authentication process.
* The system validates the user's credentials.
* Upon successful authentication, the user gains access to the application.

A diagram of a system

Description automatically generated with low confidence

* The user creates a new question by providing the necessary details.
* The system saves the question to the database.
* Other users can view and respond to the question.
* A diagram of a system

  Description automatically generated with low confidenceThe user selects a question to answer.
* The system presents the question details.
* The user provides an answer and submits it.
* The system saves the answer and associates it with the respective question.

**Architecture**

The architecture of our Stack Overflow-like application follows a multi-layered approach, commonly known as the three-tier architecture. This architecture divides the application into three main layers: the presentation layer, the application logic layer, and the data storage layer.

1. Presentation Layer: The presentation layer is responsible for handling the user interface and user interactions. In our application, this layer is implemented using React.js, HTML, CSS, and JavaScript. It provides an intuitive and responsive interface for users to interact with the application. The presentation layer communicates with the application logic layer to retrieve and display data, handle user input, and present information to the users.
2. Application Logic Layer: The application logic layer, also known as the business logic layer or the backend layer, handles the core functionalities and business rules of the application. In our MERN-based application, this layer is built using Node.js and Express.js. It receives requests from the presentation layer, processes the data, performs necessary computations, and interacts with the data storage layer. It ensures the proper functioning and behavior of the application by implementing various modules, services, and controllers.
3. Data Storage Layer: The data storage layer is responsible for managing and persisting data. In our application, we use MongoDB as our NoSQL database. This layer handles the storage and retrieval of user profiles, questions, answers, comments, and other relevant data. MongoDB's flexible schema allows us to store and query unstructured data efficiently. The application logic layer interacts with the data storage layer to fetch and manipulate data based on user requests.

**Package Diagram**

A package diagram is a structural diagram in UML (Unified Modeling Language) that represents the organization and dependencies of packages within a system. It provides a high-level view of the system's structure, showcasing how the system is organized into different packages and the relationships between them.

A picture containing text, screenshot, rectangle, font

Description automatically generated

Entities in the Package Diagram:

1. Application: The top-level package represents the entire application. It encapsulates the frontend and backend packages and serves as the entry point of the system.
2. Frontend: The frontend package contains all the components responsible for the user interface and user interactions. It includes packages such as Components, Views, and Services. This package handles the presentation layer of the application.
3. Components: The Components package consists of reusable UI components that are utilized across the application. These components contribute to the modular and scalable nature of the frontend.
4. Views: The Views package contains different views or pages of the application. Each view represents a specific page or screen that the user interacts with, such as the home page, question page, user profile page, etc.
5. Services: The Services package includes modules responsible for interacting with the backend and fetching data from the API endpoints. These services facilitate communication between the frontend and backend layers.
6. Backend: The backend package encompasses all the server-side logic of the application. It includes packages such as Controllers, Services, Models, and Middleware.
7. Controllers: The Controllers package contains modules that handle incoming requests, perform necessary validations and transformations, and coordinate the flow of data between the frontend and backend layers.
8. Services: The Services package implements the core business logic of the application. It handles the processing of data, performs operations such as authentication, validation, and interacts with the data storage layer.
9. Models: The Models package represents the data structures and entities used within the application. It defines the schema and structure of data objects such as users, questions, answers, etc., and provides the interface to interact with the data storage layer.
10. Middleware: The Middleware package includes modules that provide additional functionalities, such as authentication middleware, logging middleware, error handling middleware, etc. These modules intercept and process incoming requests before they reach the controllers.
11. External Packages: The diagram also showcases the existence of external packages that our application may rely on, such as libraries, frameworks, or third-party services. These packages are represented in the diagram to indicate their integration with our application.

A class diagram is a type of static structure diagram in UML (Unified Modeling Language) that depicts the structure and relationships between classes in a system. It illustrates the attributes, methods, and associations of classes, providing a detailed view of the system's object-oriented design.

A picture containing text, screenshot, font, rectangle

Description automatically generated

Here is a simplified class diagram for STACKBETTERFLOW:

Entities in the Class Diagram:

1. User: The User class represents a user of our application. It encapsulates properties such as id, username, email, password, and reputation. These attributes store information about the user's identity, credentials, and reputation within the community.
2. Question: The Question class represents a question posted by a user. It contains attributes like id, title, body, tags, and userId. These attributes store information about the question's unique identifier, title, description, associated tags, and the user who posted the question.
3. Answer: The Answer class represents an answer provided by a user in response to a question. It has attributes such as id, content, questionId, userId, and upvotes. These attributes store information about the answer's identifier, content, the question it belongs to, the user who posted the answer, and the number of upvotes it has received.

The class diagram showcases the key entities and their attributes in our Stack Overflow-like application. The User class represents users and their associated information, while the Question and Answer classes represent the questions and answers posted by users, respectively. These classes capture the core data structures and relationships within the application, forming the foundation for the application's functionality.

**Database**

Entities:

1. User:
   * id: The unique identifier for a user.
   * username: The username chosen by the user.
   * email: The email address of the user.
   * password: The hashed password of the user.
   * reputation: The reputation score of the user.
2. Question:
   * id: The unique identifier for a question.
   * title: The title of the question.
   * body: The body or content of the question.
   * tags: An array of tags associated with the question.
   * User Id: The identifier of the user who posted the question.
3. Answer:
   * id: The unique identifier for an answer.
   * content: The content or body of the answer.
   * Question Id: The identifier of the question to which the answer is posted.
   * User Id: The identifier of the user who posted the answer.
   * upvotes: The number of upvotes received for the answer.

Relations:

* One-to-Many (User to Question):
  + A user can post multiple questions.
  + A question is posted by a single user.
  + The relationship is established by the user Id attribute in the Question entity.
* One-to-Many (User to Answer):
  + A user can post multiple answers.
  + An answer is posted by a single user.
  + The relationship is established by the user Id attribute in the Answer entity.
* One-to-Many (Question to Answer):
  + A question can have multiple answers.
  + An answer belongs to a single question.
  + The relationship is established by the question Id attribute in the Answer entity.

These relations enable the establishment of connections between users, questions, and answers in the database. By utilizing these relationships, we can retrieve the questions and answers associated with a specific user, fetch all answers for a particular question, or retrieve the user who posted a specific question or answer.

**A2**

**Frontend**

**Cimpan Stefan**

**User Pages**

As part of my project, I developed a clone of Stack Overflow, focusing on creating user-specific pages. Specifically, I implemented pages for asking questions, authentication, and the home page.

For the "Ask Questions" page, users can submit their queries and receive answers from the community. This feature enables users to seek assistance and share their knowledge within the platform.

Regarding authentication, I implemented a dedicated page where users can register and log in to access personalized features. By securely storing user information and managing authentication, I ensure a smooth and secure user experience.

The home page serves as the central hub, providing a comprehensive view of the platform's content. Users can browse through a curated feed of popular questions, recent updates, and relevant discussions. This intuitive interface helps users navigate and engage with the platform effectively.

Throughout the development process, I focused on ensuring a user-friendly and visually appealing design. By leveraging modern web development technologies, I aimed to deliver an intuitive and responsive user interface that caters to the needs of Stack Overflow's target audience.

By creating these user-focused pages, I strived to replicate the core functionalities of Stack Overflow while adding my own unique touches to enhance the user experience.

Usage of Modules & Components/ Similar

I have successfully implemented components for essential elements such as avatars, buttons, the home main bar, the left sidebar, the right sidebar, and the navigation. These components play a vital role in establishing a polished and professional user interface.

By breaking down the user interface into modular components, the codebase benefits from improved organization, reusability, and maintainability. Each component serves a distinct purpose, encompassing specific functionalities and visually appealing elements. This modular approach allows for seamless integration and facilitates reuse across different sections of the application.

Furthermore, consider the compositional aspect of these components. Thoughtful consideration should be given to how they come together to form larger sections or layouts within the application. For instance, combining the home main bar, left sidebar, and right sidebar creates a cohesive and visually pleasing page layout. The components should be designed with flexibility in mind, enabling them to adapt and fit harmoniously within diverse contexts.

The inclusion of these meticulously crafted components ensures a consistent and professional user experience throughout the application. The usage of node\_modules and the development of these components contribute significantly to the overall quality and sophistication of the project.

**Clean Code**

Here are some characteristics that define my Clean Code:

**Readability**: Your code is easy to understand, with meaningful and descriptive names for variables, functions, and classes. It follows a consistent formatting style, utilizes proper indentation, and employs clear and concise comments to explain complex sections or algorithms.

**Simplicity**: Your code avoids unnecessary complexity and embraces the concept of "less is more." It follows the principle of keeping functions and classes focused on a single responsibility, avoiding excessive dependencies, and minimizing the number of lines required to achieve a particular task.

**Modularity**: Your code is modular and promotes the separation of concerns. It organizes functionality into logical modules or components, allowing for easier testing, debugging, and reusability. Dependencies between modules are minimized, leading to a more flexible and maintainable codebase.

**Consistency**: Your code follows consistent coding conventions and style guidelines. This consistency makes it easier for other developers to navigate and understand the code. It also ensures that code changes or additions integrate smoothly with the existing codebase.

**Usage of Services**

Modularity: You have effectively modularized your application's functionality by encapsulating related operations and logic within service components. Services provide a clear separation of concerns, allowing for better organization, code reuse, and maintainability.

Single Responsibility: Each service focuses on a specific task or responsibility within your application. By adhering to the Single Responsibility Principle, you ensure that services have well-defined purposes and do not become bloated with unrelated functionalities. This enhances the readability and maintainability of your code.

Abstraction of Implementation: Services provide an abstraction layer that hides the internal implementation details from other parts of your application. This allows other components, such as controllers or user interfaces, to interact with the services without needing to understand the underlying implementation complexities. This abstraction improves code clarity and facilitates future changes or updates.

Reusability: By separating your application's functionality into services, you enable code reuse across different parts of your application. Services can be accessed and utilized by multiple components, promoting a DRY (Don't Repeat Yourself) coding principle. This reduces duplication and ensures consistent behavior and data management throughout your application.

**Routing Documentation:**

**- "/" (exact path)**

**- Components rendered:**

**- NavMenuDesktop**

**- QuesListPage**

**- RightSidePanel**

**- "/ask" (exact path)**

**- Components rendered:**

**- If user exists:**

**- NavMenuDesktop**

**- AskQuestionPage**

**- RightSidePanel**

**- If user doesn't exist:**

**- Redirected to "/"**

**- "/tags" (exact path)**

**- Components rendered:**

**- NavMenuDesktop**

**- AllTagsPage**

**- "/users" (exact path)**

**- Components rendered:**

**- NavMenuDesktop**

**- AllUsersPage**

**- "/user/:username" (exact path)**

**- Components rendered:**

**- NavMenuDesktop**

**- UserPage**

**- "/questions/:quesId" (exact path)**

**- Components rendered:**

**- NavMenuDesktop**

**- QuestionPage**

**- "/tags/:tagName" (exact path)**

**- Components rendered:**

**- NavMenuDesktop**

**- QuesListPage with tagFilterActive set to true**

**- RightSidePanel**

**- "/search/:query" (exact path)**

**- Components rendered:**

**- NavMenuDesktop**

**- QuesListPage with searchFilterActive set to true**

**- RightSidePanel**

**- All other routes (default)**

**- Components rendered:**

**- NavMenuDesktop**

**- NotFoundPage**

**- RightSidePanel**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated