SE Project 3

Task 1: Requirements and Subsystems

Functional Requirements:

- Authentication:
  - Users should be able to sign up, log in, and log out securely.
  - Users should be able to update their profile information.
- Real-Time Chat:
  - Users should be able to send and receive messages in real-time to other users.
  - Messages should be delivered instantly with minimal latency.
  - Messages should support text and media attachments.
- Search Users:
  - Users should be able to search for other users by their usernames or other criteria.
  - Search results should be displayed in a user-friendly manner.
- Group Chats:
  - Users should be able to create group chats with multiple participants.
  - Group chats should support sending messages, adding/removing participants, and changing group settings.

Non-Functional Requirements:

- Performance:
  - The system should handle high loads and concurrent users without significant performance degradation.
  - Real-time communication should have minimal latency to provide a smooth user experience.
- Scalability:
  - The architecture should support horizontal scaling to accommodate increasing numbers of users and messages.

    - The database should be scalable to handle growing data volumes.
- Security:
    - User authentication and authorization should be secure to prevent unauthorized access.
- Reliability:
    - The system should be available and reliable, with minimal downtime.
    - Data integrity should be maintained, and messages should not be lost.
- Usability:
    - The user interface should be intuitive and easy to navigate.
    - Error messages should be informative and user-friendly.

Architecturally Significant Requirements:

- Real-Time Communication:
    - The real-time one-to-one chat and group chat functionalities require a WebSocket-based communication architecture to enable instant message delivery.
    - Node.js, as part of the MERN stack, is well-suited for real-time communication using WebSockets. It allows for bidirectional communication, crucial for implementing real-time chat features.
- Scalable Backend:
    - The backend should be designed using scalable technologies such as Node.js and MongoDB, allowing it to handle a large number of users and messages.
- Authentication Mechanism:
    - Email and password authentication with hashed passwords stored in the database ensures secure access to the application's features and data.
- Database Design:

    - The database schema should be designed efficiently to support complex queries, indexing, and data relationships, especially for group chats and user management.
    - MongoDB's flexible document-based data model is well-suited for storing chat messages, user profiles, and group chat information, allowing for easy scalability and adaptability to changing data requirements.

Subsystem Overview:

User Management Subsystem:

- User Registration: Allows new users to create accounts by providing necessary information such as username, email address, and password. It may include validation mechanisms to ensure data accuracy.
- User Authentication: Verifies the identity of users during the login process using credentials provided during registration (e.g., username/email and password). It ensures secure access to the system by authenticating user identities.
- User Profile Management: Enables users to manage their profiles by updating personal information, profile pictures, and preferences. Users may also have the ability to customize their account settings.

Message Handling Subsystem:

- Message Composition: Enables users to compose and send messages to other users or groups within the system. Users can typically compose messages using text, multimedia content, or attachments.
- Message Delivery: Handles the delivery of messages from senders to recipients, ensuring that messages are transmitted securely and reliably. This may involve protocols such as SMTP (Simple Mail Transfer Protocol), HTTP (Hypertext Transfer Protocol), or custom messaging protocols.
- Message Storage: Stores messages temporarily or permanently, depending on system requirements and user preferences. Message storage ensures that messages can be accessed and retrieved by users at a later time.

- Message Notifications: Notifies users about new messages, replies, or other relevant events through notifications sent via email, push notifications, or in-app notifications. Notifications

Group and Chat Management Subsystem:

- Group Creation: Allows users to create new groups for specific topics, projects, or interests. Group creators typically define group names, descriptions, privacy settings, and membership criteria.
- Group Invitation and Membership Management: Enables users to invite others to join groups and manage group memberships. Group administrators may have the authority to approve or reject membership requests and manage member roles and permissions.
- Real-time Chatting: Provides a real-time messaging platform for group members to engage in text-based conversations, discussions, and collaboration. It may support features such as text formatting, emojis, file sharing, and multimedia content.
- Group Notifications: Notifies group members about new messages, mentions, or other relevant events within the group. Notifications ensure that users stay updated on group activities and conversations, even when they are not actively participating.

Task 2: Architecture Framework

Stakeholder Identification:

1. Identify Stakeholders: Potential stakeholders include:
    - Users (individual users, group participants)
    - Developers
    - System administrators
    - Project managers
    - Business owners or clients
2. Determine Stakeholder Concerns: Some examples of concerns for the app could be:
    - Users: Usability, performance, privacy and security

    - Developers: Maintainability, extensibility, and testability of the codebase
    - System administrators: Deployment, monitoring, and scalability
    - Project managers: Budget, timeline, and resource allocation
    - Business owners/clients: Cost-effectiveness, business value, and compliance with regulations
3. Define Viewpoints: Potential viewpoints could include:
    - Logical Viewpoint: Describes the functional requirements, features, and interactions within the system.
    - Development Viewpoint: Focuses on the software architecture, design patterns, and implementation details.
    - Deployment Viewpoint: Addresses the physical deployment, infrastructure, and operational aspects.
    - Security Viewpoint: Covers the security requirements, threats, and mitigation strategies.
4. Create Views: Some views include:
    - Use Case Diagrams (Logical Viewpoint): Illustrating the functional requirements and user interactions.
    - Component Diagrams (Development Viewpoint): Depicting the architectural components and their relationships.
    - Deployment Diagrams (Deployment Viewpoint): Showing the physical deployment of components across servers or environments.
    - Data Flow Diagrams (Security Viewpoint): Representing the flow of data and potential security risks.

Major Design Decisions:

ADR 1: - Using Monolithic architecture:

We have chosen to adopt a monolithic architecture for our application because:

- **Development Speed:**
    - We aim to deliver our application within a reasonable timeframe, and a monolithic architecture can help development by allowing us to focus on building features rather than managing microservices.
- **Team Familiarity:**
    - Our team is experienced with monolithic architecture and does not have enough expertise for other architecture patterns like microservices. So using monolithic architecture enables us to leverage existing knowledge and expertise effectively.
- **Simplicity:**
    - Given the size and scope of our application, a monolithic architecture provides simplicity in deployment, testing, and maintenance, making it easier to manage.

ADR 2: Choice of MERN Stack

We have decided to use the MERN (MongoDB, Express.js, React, Node.js) stack for building our chat application because:

- **Full-stack JavaScript Development:**
    - Using the MERN stack allows us to develop both the frontend and backend of our application using JavaScript. This facilitates code reuse, reduces context switching, and makes it easier to develop.
- **Scalability and Performance:**
    - Node.js, as part of the stack, provides non-blocking, event-driven architecture, making it suitable for handling real-time communication efficiently. This enables our chat application to handle multiple concurrent connections and scale as the user base grows.
- **React for Dynamic UI:**
    - React's component-based architecture enables us to create a dynamic and interactive user interface.

ADR 3 - Use of MongoDB for Data Storage:
We have opted to use MongoDB as our database for storing chat messages, user profiles, and group chat information.

- **Flexible Data Model:**
    - MongoDB's document-based data model provides flexibility in storing unstructured data, which is suitable for storing chat messages and user profiles.
- **Scalability:**
    - MongoDB is horizontally scalable, allowing us to scale our database easily as the volume of data increases or as the user base grows.
- **Real-Time Data Access**:
    - MongoDB's query language and indexing capabilities enable fast and efficient access to real-time data, crucial for implementing features like message retrieval and user search.

Task 3: Architectural Tactics and Patterns
Architectural Tactics:

1. Authentication and Authorization Middleware
    - Explanation: Implement middleware for authentication and authorization to validate user credentials and enforce access control policies. This is achieved by using custom middleware for authorization.
    - Addressing Non-Functional Requirement: Enhances security by ensuring that only authenticated users have access to sensitive data and functionalities. Proper authentication and authorization mechanisms also contribute to compliance with data protection regulations and protect against unauthorized access.
2. Asynchronous Processing

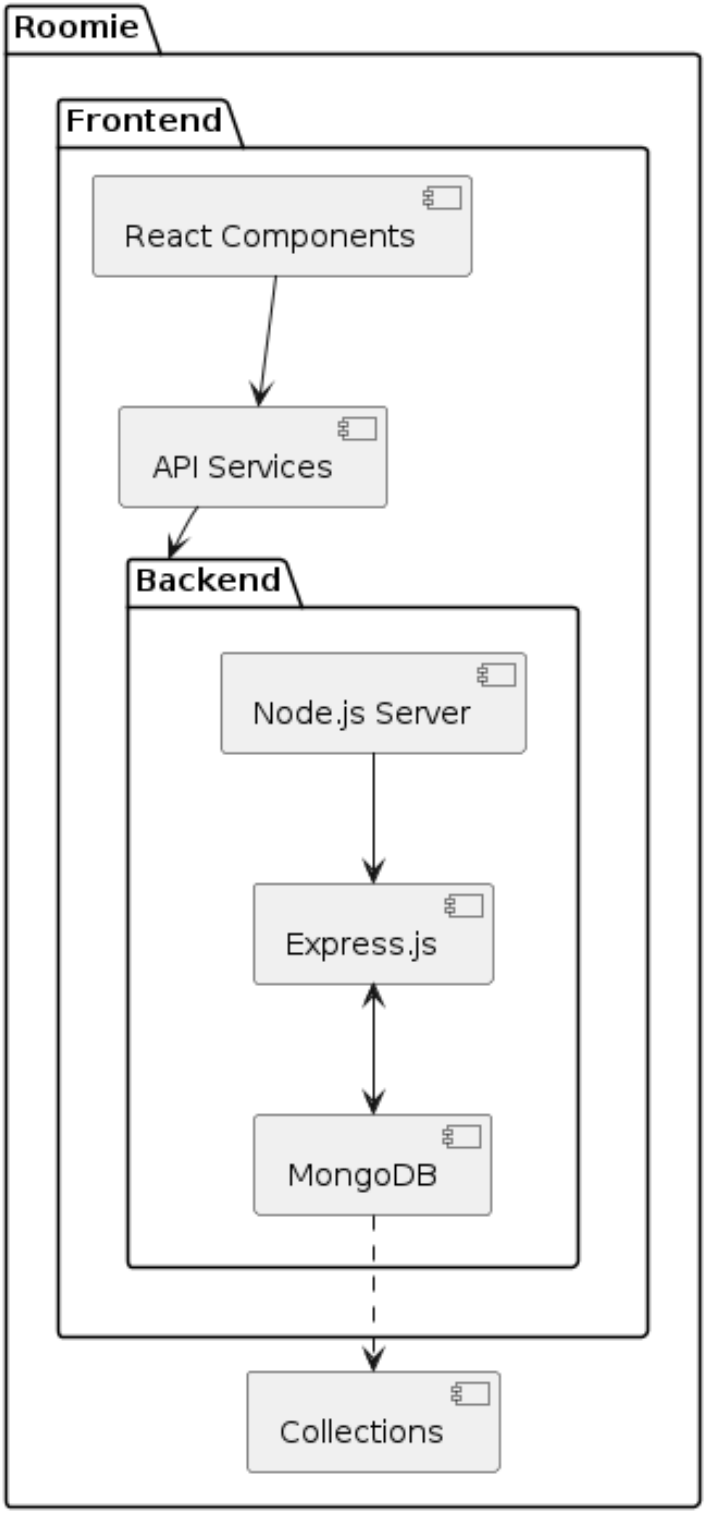
- Explanation: Utilize asynchronous processing for tasks such as sending notifications, updating user statuses, and processing chat messages.
- Addressing Non-Functional Requirement: Improves performance and scalability by offloading time-consuming tasks from the main application thread, allowing it to handle more requests concurrently. Asynchronous processing also enhances fault tolerance by decoupling components, ensuring that failures in one part of the system do not impact others.

3. Websockets for Real-time Communication:
    - Explanation: Utilize Websockets to enable real-time communication between clients and the server for instant message delivery and updates.
    - Addressed Non-functional Requirements: Real-time responsiveness and user engagement. Websockets facilitate instant message delivery without the need for constant polling, providing a seamless messaging experience for users.

Implementation Pattern:

1. MVC (Model-View-Controller):
    - Role: Divide the application into three interconnected components: Model (data storage and manipulation), View (user interface), and Controller (logic and flow control).
    - Explanation: The MVC pattern separates concerns, making the application more organized and easier to maintain. For example, in the messaging app, the Model handles data storage in MongoDB, the View is represented by React components rendering UI elements, and the Controller manages user interactions and business logic in Express.js routes.

High Level diagram of MVC architecture implemented



Prototype Development:
Link to the github repo - https://github.com/ram-ek/roomie

Architecture Analysis:
We have implemented the MVC architecture in order to build the application, which can be considered as a monolithic architecture.
We have also build a subsystem of user authentication as a microservice,
We have done the comparison of both these architectures.
In order to compare the performance based on the 2 non-functional requirements - response time and throughput, we have done the following -

We have created a test script to test the number of requests our system was able to service in one second as well as what is the average response time for the requests (testing is done by sending 10000 requests).

1. For the monolithic architecture of user login sub-component:
    a. Response time: 0.69 ms
    b. Throughput: 1451.37 requests/second
2. For the microservice architecture of user management:
    a. Response time: 0.6682 ms
    b. Throughput: 1496.55 requests/second

These results indicate that both architectures have similar response times, with the microservice architecture slightly faster by 0.0218 ms. However, the microservice architecture also has slightly higher throughput, indicating better capacity to handle concurrent requests.

Contributions:

1. Karun Choudhary:

- Leveraged Socket.IO to establish WebSocket connections between clients and the server.
- Implemented event-based messaging where clients can send and receive messages in real-time.

- Used Socket.IO's rooms or namespaces to facilitate private or group messaging functionalities.
- Created endpoints (routes) in Express to handle file uploads using Multer middleware for multipart/form-data.
- Implemented secure file storage mechanisms for storing files on a local server.
- Developed endpoints for downloading files.

2. Vivek Ram:

- Designed and implemented RESTful API endpoints using Node.js and Express to handle chat-related operations such as sending messages, retrieving message history, and managing user conversations.
- Spearheaded the design and development of microservices using Node.js and related frameworks like Express and Nest.js.
- Decomposed monolithic features into independent microservices, each responsible for specific functionalities such as user management.

3. Ronak Redkar::

- Developed mongodb database models to store user data, messages and group chats.
- Implemented user registration and authentication mechanism using JSON Web Token.
- Implemented RESTful API endpoints in controllers to search, register and authorize the users.
- Designed frontend UI for Login/SignUp page.

4. Naitik Kariwal:

- Designed frontend UI for chat-app main page:
    - My chats panel
    - Single chat panel (right-side)
    - Group creation
    - Search feature on nav-bar
    - Profile viewer and logout on nav-bar

5. Jayank Mahaur:

- Performed the work of integrating the complete app (i.,connecting frontend and backend)
- Created a test script to test the response time and throughput of both the monolithic and microservice architecture for the user login sub-component
- Performed the work of testing the entire application at several stages to report the bugs to all the team members for fixing.