INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR

MESSAGING SERVICE PROTOTYPE: SYSTEM DESIGN AND SETUP

A REPORT ON THE

ASSIGNMENT

Submitted By:

John Twipraham Debbarma
(Roll No.: 22110108)

The code is available at the GitHub repository link for your kind perusal:
https://github.com/JohnTwiprahamDebbarma/messaging-service-prototype

Introduction:

This document provides an overview of the messaging service prototype web app, which integrates real-time communication and various functionalities such as user registration, following users, and engaging in personal or group chats. The web app has been built using modern technologies for both frontend and backend, ensuring scalability, security, and a smooth user experience.

The system leverages **Next.js** as a React-based frontend framework, enabling server-side rendering for faster load times and better SEO. The frontend is styled using **Tailwind CSS**, a utility-first CSS framework that provides flexibility and a mobile-responsive layout without the need for writing custom styles. Real-time messaging capabilities are made possible using **Pusher**, which integrates WebSocket-based communication for instant notifications and updates.

On the backend, **Node.js** is used with **Express.js** to handle API routing and server-side logic. **Prisma**, a robust ORM (Object-Relational Mapping) tool, is used for interacting with **MongoDB**, a NoSQL database that efficiently manages user data, chat history, and message storage. **MongoDB** is highly scalable and suitable for handling large amounts of unstructured data, such as messages and user interactions.

The app ensures real-time interaction, where user messages are broadcasted to recipients through WebSockets via **Pusher**, guaranteeing instant delivery of messages in both personal and group chats. This allows for real-time user engagement, ensuring that both parties receive updates immediately without delays.

The following sections will dive deeper into the architecture, communication flow, and dependencies used in building this application, as well as instructions on setting up and running the prototype.

This document provides an overview of the messaging service prototype web app, detailing the architectural choices, system components, and the libraries used.

System Architecture:

- Frontend (Client)
    - Framework: Next.js (React-based)
    - CSS Framework: Tailwind CSS
    - WebSocket Integration: Pusher

Atomic Design in the Frontend:

Following **Atomic Design** principles, the frontend is divided into:

- **Atoms**: Smallest building blocks like buttons, form fields, and icons.
- **Molecules**: Combinations of atoms such as input fields combined with labels or buttons.
- **Organisms**: Groups of molecules forming distinct sections of the interface, like chat windows.
- **Templates**: Page-level layouts using organisms to define structure.
- **Pages**: Final output combining templates and data to render the user interface.

- Backend (Server)
    - Framework: Node.js with Express.js for handling API routes.
    - ORM: Prisma for database management.
    - Database: MongoDB, a NoSQL database for managing user data and chat history.
    - Real-Time Messaging: Pusher for real-time notifications and updates.

Atomic Design in the Backend

Atomic Design principles in the backend aim to modularize functionalities:

- **Atoms**: Individual utility functions (e.g., token generation).
- **Molecules**: Grouped functionality (e.g., authentication handlers).
- **Organisms**: Full features such as user management, chat handling, etc.

Example:

- **Atom**: A function to generate a JWT token.
- **Molecule**: A middleware function that checks authentication using the token.

- **Organism**: User authentication logic (signup, login, JWT validation).

- Communication Flow
    - Client to Server: Users interact with the frontend through Next.js. Data is fetched and sent to the server using RESTful APIs or WebSockets via Pusher for real-time features.
    - Server to Database: The server uses Prisma to interface with MongoDB, managing user accounts, chats, and messages.
    - Real-Time Messaging: Messages are sent via WebSockets using Pusher, and both parties (or groups) receive real-time updates.

System Components:

Frontend (Client)

- Next.js: Allows server-side rendering and improves SEO and performance.
- React: The core library for building interactive UI components.
- Tailwind CSS: Utility-first CSS framework for creating responsive and customizable UI designs.
- Pusher: Enables WebSocket-based real-time communication for chats.

Why These Technologies?

- Next.js: Chosen for server-side rendering, performance benefits, and easy deployment.
- Tailwind CSS: Allows easy customization of styles without writing custom CSS, making the app scalable and maintainable.
- Pusher: Provides robust WebSocket functionality for real-time messaging, essential for this app.

Backend (Server)

- Node.js with Express.js: Handles API routing and server-side logic.
- Prisma: ORM for simplifying database interactions with MongoDB, including type-safe queries and migrations.
- MongoDB: NoSQL database ideal for unstructured and growing data (e.g., chat histories).
- Pusher: Manages real-time updates and notification functionality.

Why These Technologies?

- Node.js with Express.js: Simple and fast for API development.
- Prisma: Simplifies database management, making it easier to query and modify MongoDB.
- MongoDB: Flexible and scalable, perfect for handling large amounts of unstructured chat data.
- Pusher: Chosen for its seamless real-time messaging and notification features.

Scalability Considerations:

- Horizontal Scaling: Additional frontend and backend instances can be deployed to handle growing user traffic.
- Database Sharding: MongoDB supports sharding, allowing for the division of large datasets across multiple databases for improved performance.
- Caching: For performance optimization, caching mechanisms can be added to reduce the load on the database.

Setup Documentation:

1. Prerequisites
    - Node.js (version 16+)
    - MongoDB (local or cloud instance)
    - Pusher account (for real-time features)
    - Prisma (ORM)

2. Installation Steps:
    - Backend setup:
        - Clone the repository:
            - git clone <repository-url>
            - cd backend
            - npm install

        - Set environment variables by creating a `.env` file:
            - DATABASE_URL=mongodb://<user>:<password>@cluster.mongodb.net/<dbname>?retryWrites=true&w=majority
            - PUSHER_APP_ID=<your-pusher-app-id>
            - PUSHER_KEY=<your-pusher-key>
            - PUSHER_SECRET=<your-pusher-secret>
            - JWT_SECRET=<your-jwt-secret>
        - Run Prisma migrations:
            - npx prisma migrate dev

    - Frontend Setup:
        - Navigate to the frontend directory:
            - cd frontend
            - npm install
        - Set environment variables by creating a `.env.local` file:
            - NEXT_PUBLIC_PUSHER_KEY=<your-pusher-key>
            - NEXT_PUBLIC_API_URL=http://localhost:3000
        - Start the frontend server:
            - npm run dev
    - Run the Application:
        - Open `http://localhost:3000` in your browser to access the web app.
        - Register and log in to test the messaging service functionality.

Conclusion:

This document outlines the design and setup of the messaging service prototype using **Atomic Design** principles. By breaking down the system into modular and scalable components, the application is well-suited for future expansions and maintenance.