A Project Report

on

GROUP CHAT APPLICATION USING MERN STACK

Submitted in partial fulfillment of the

requirement for the award of the degree

of

B.Tech- Computer Science & Engineering

Under The Supervision
of Mr. V Arul
Assistant Professor
Department of Computer Scienece and Engineering

Submitted By

Muskan (18021011632)

Samarth Nanda (18021011704)

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
GALGOTIAS UNIVERSITY, GREATER NOIDA
INDIA
DECEMBER-2021

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING, GALGOTIAS UNIVERSITY, GREATER NOIDA

CANDIDATE'S DECLARATION

We hereby certify that the work which is being presented in the thesis/project/dissertation, entitled **"GROUP CHAT APPLICATION USING MERN STACK"** in partial fulfillment of the requirements for the award of the Btech submitted in the School of Computing Science and Engineering of Galgotias University, Greater Noida, is an original work carried out during the period of **JULY-2021 to DECEMBER-2021**, under the supervision of **Mr. V. ARUL, Assistant Professor, Department of Computer Science and Engineering** of School of Computing Science and Engineering , Galgotias University, Greater Noida

The matter presented in the project has not been submitted by us for the award of any other degree of this or any other places.

Muskan 18SCSE1010401

Samarth Nanda 18SCSE1010476

This is to certify that the above statement made by the candidates is correct to the best of my knowledge.

Supervisor Name
Designation

CERTIFICATE

The Final Project examination of **Muskan 18SCSE1010401, Samarth Nanda 18SCSE1010476**. has been held on December-2021 and the work is recommended for the award of **BACHELOR OF TECHNOLOGY IN COMPUTER SCIENCE AND ENGINEERING.**

Signature of Examiner(s) **Signature of Supervisor(s)**

Signature of Project Coordinator **Signature of Dean**

Date: 17 December, 2021
Place: Greater Noida

ACKNOWLEDGEMENT

We would like to acknowledge the great efforts and precious time given by our project guide **Mr. V Arul** Assistant Professor in Department of Computer Science and Engineering**,** School of Computing Science and Engineering , Galgotias University, Greater Noida .We were able to complete our job thanks to great advice and criticism. We'd also want to thank our parents for their constant support and care. Last but not least, without our team, we would not have been able to build this project and learn so much in the process.

ABSTRACT

This dissertation describes the process of the development of a chat application for developers, from a mere idea to a working cloud service. We have built a real time platform that makes it easy to have a group conversation between a projects' members, share code and stay up to date with their latest repository updates. It has given him a better understanding of ExpressJS and WebSocket, React.JS, NoSQL databases (MongoDB), REST JSON APIs with Node, and more. Web chat is a system that allows users to communicate in real-time using easily accessible web interfaces. It is a type of Internet online chat distinguished by its simplicity and accessibility to users who do not wish to take the time to install and learn to use specialized chat software which help to meet with unknown people and explore them and know the different different culture at your own comfort this also help to reduce stress and keep people mind clam and this will help to improve work efficiency of any person now a days in fast moving and modern culture no one have free time to talk with their relative. This application will help reduce the distance in the world by getting closer through this platform we have built

ACRONYMS

B.Tech.	Bachelor of Technology
RTC	Real – Time Communication
WebRTC	Web Browser Real Time Communication
HTML	Hype Text Markup Language
HTTPS	Hyper Text Transfer Protocol
CSS	Cascading Style Sheet
TURN	Traversal Using Relays around NAT
P2P	Peer-to-Peer
CPU	Central Processing Unit
GUI	Graphical User Interface

CHAPTER- 1

Introduction

1.1 Introduction About the Project

This project is to create a chat application with a server and users to enable the users to chat with each other's To develop an instant messaging solution to enable users to seamlessly communicate with each other. The project should be very easy to use enabling even an o vice person to use it. This project can play an important role in organizational field where employees can connect through LAN. The main purpose of this project is to provide group chatting functionality through network.

Chatting is a method of using technology to bring people and ideas together despite of the geographical barriers. The technology has been available for years but the acceptance was quite recent. Our project is an example of a chat server. It is made up of two applications-the client application, which runs on the user's web browser and server application, runs on any hosting servers on the network. To start chatting client should get connected to server where they can do private and group chat. Security measures were taken during the last one The MERN stack which consists of Mongo DB, Express.js, Node.js, and React.js is a popular stack for building full-stack web-based applications because of its simplicity and ease of use. In recent years, with the explosivepopularity and the growing maturity of the JavaScript ecosystem, the MERN stack has been the go to stack for a large number of web applications. This stack is also highly popular among newcomers to the JS field because of how easy it is to get started with this stack. This repository consists of a Chat Application built with the MERN stack .I built this sometime back when I was trying to learn the stack and I have left therefor anyone new to the stack so that they can use this repo as a guide. This is a full-stack chat application that can be up and running with just a few steps. Its frontend is built with Material UI running on top of React. The backend is built with Express.js and Node.js. Real-time message broadcasting is developed using Socket.IO.

1.2 Aim

The aim of this project is to build a functional real-time messaging application for developers by using modern web technologies. Unlike most chat applications available in the market, this one will focus on developers and will attempt to boost their productivity. Although we are not expecting it to have a plethora of utilities due to the limited time frame, sharing code and watching a repository will be our core features. It will be fully open-source. Everyone will be able to dig into the code to read what is going on behind the scenes, or even contribute to the source code. So it was within our intentions to write clean, scalable code following the most popular patterns and conventions for each of the languages and relevant libraries.

1.3 Ideation

This dissertation describes the process of the development of a chat application for developers, from a mere idea to a working cloud service. We have built a real time platform that makes it easy to have a group conversation between a projects' members, share code and stay up to date with their latest repository updates.

Chatting is a method of using technology to bring people and ideas together despite of the geographical barriers. The technology has been available for years but the acceptance was quite recent. Our project is an example of a chat server. It is made up of two applications-the client application, which runs on the user's web browser and server application, runs on any hosting servers on the network. To start chatting client should get connected to server where they can do private and group chat. Security measures were taken during the last one The MERN stack which consists of Mongo DB, Express.js, Node.js, and React.js is a popular stack for building full-stack web-based applications because of its simplicity and ease of use[1]. In recent years, with the explosivepopularity and the growing maturity of the JavaScript ecosystem, the MERN stack has been the go to stack for a large number of web applications. This stack is also highly popular among newcomers to the JS field because of how easy it is to get started with this stack. This repository consists of a Chat Application built with the MERN stack .I built this sometime

back when I was trying to learn the stack and I have left therefor anyone new to the stack so that they can use this repo as a guide. This is a full-stack chat application that can be up and running with just a few steps. Its frontend is built with Material UI running on top of React. The backend is built with Express.js and Node.js. Real-time message broadcasting is developed using Socket.IO[2].

This application provides users with the following features of Authentication using JWT Tokens. A Global Chat which can be used by anyone using the application to broadcast messages to everyone else. A Private Chat functionality where users can chat with other users privately. Real-time updates to the user list, conversation list, and conversation messages Chatting is a method of using technology to bring people and ideas together despite of the geographical barriers. The technology has been available for years but the acceptance was quite recent.

Our project is an example of a chat server. It is made up of two applications- the client application, which runs on the user's web browser and server application, runs on any hosting servers on the network. To start chatting client should get connected to server where they can do private and group chat. Security measures weretaken during the last one.

1.4 Features

Before getting into any specific chat features that our application should/could have, we will list the basic ones that most chat services offer us nowadays, regardless of their type:

- Instant messaging
- Notifications
- Message sender (username)
- Group chats
- Join Multiple rooms
- Connect to different groups
- Keep your chat Backup
- Chats Encrypted End to End
- List of online members
- Video calls
- Group meeting
- Emojis & animated emoticons

1.5 System Requirements

Now, this method is intended in such the way that it takes fewer resources to figure out work correctly. That is the minimum needs that we'd like to require care of:-

- The system wants a minimum of two GB of ram to run all the options.
- It wants a minimum 1.3 GHz processor to run smoothly.
- Rest is all up to the user's usage can take care of hardware.
- For security opposing anti-virus is suggested.

- **RAM**: At least 256 MB of RAM. The amount of RAM needed depends on the number of concurrent client connections, and whether the server and multiplexor are deployed on thesame host.
- **Disk Space**: Approximately 300 MB required for Instant Messaging Server software.
- **Processor**: Minimum 1.3 gigahertz (GHz) x86- or x64-bit dual core processor with SSE2instruction set and recommended 3.3 gigahertz (GHz) or faster 64-bit dual core processorwith SSE2 instruction set.
- **Memory**: Minimum 2-GB RAM and recommended 4-GB RAM or more

The system is made correctly, and all the testing is done as per the requirements. So, the rest of the things depend on the user, and no one can harm the data or the software if the proper care is done.

CHAPTER-2

Literature Survey

2.1 Competitive Analysis

Competitive Analysis Prior to getting started with the application development, we did some research on the current messaging platforms out there. We were looking forward to building a unique experience, rather than an exact clone of an existing chat platform. We already knew of the existence of several messaging applications, and a few chat applications that suited developers. However, never before had we done an in-depth analysis of their tools to find out whether they were good enough for developers. Soon, we realized that none of the sites were heading in our direction. Some of them were missing features which we considered crucial and others had opportunities for further enhancements. Contrary to what many people think, having a few platforms around is not a necessarily a bad thing. We were able to get ideas of what to build and how and determine which technologies and strategies to use based on their experience. Often, this was as simple as checking their blogs[3]. Companies like Slack regularly post development updates (such as performance reviews, technology comparisons, and scalability posts). Other times, we had to dig into the web to find out the different options we had and pick out the one which we considered to be the most appropriate.

2.2 Communication Protocols

For most web applications, communication protocols are not a subject of discussion. AJAX through HTTP is the way to go since it is reliable and widely supported. However, that is not our case. We need, albeit not in every single situation, an extremely fast communication method to send/receive messages in real time. For messaging, there are a few communication protocols available for the web. The most popular ones are AJAX, WebSockets, and WebRTC. AJAX is a slow approach. Not only because of the headers that have to be sent in every request, but also, and more important, because there is no way to get notified of new messages in a chat room.

By using AJAX, we would have to request/pull new messages from the server every few seconds, which would result in new messages to take up to a few seconds to appear on the screen, not to say the numerous redundant requests that this would generate. WebSockets are a better approach. WebSockets connections can take up to few seconds to establish, but thanks to the full-duplex communication channel, messages can be exchanged swiftly (averaging few milliseconds delay per message). Also, both client and server can get notified of new requests through the same communication channel, which means that unlike AJAX, the client does not have to send the server a petition to retrieve new messages but rather wait for the server to send them.

Lets discuss all communication protocols for web:-

2.2.1 AJAX

Asynchronous JavaScript and XML, while not a technology in itself, is a term coined in 2005 by Jesse James Garrett, that describes a "new" approach to using a number of existing technologies together, including HTML or XHTML, CSS, JavaScript, DOM, XML, XSLT, and most importantly the XML Http Request object. When these technologies are combined in the Ajax model, web applications are able to make quick, incremental updates to the user interface without reloading the entire browser page. This makes the application faster and more responsive to user actions.

Although X in Ajax stands for XML, JSON is preferred over XML nowadays because of its many advantages such as being a part of JavaScript, thus being lighter in size. Both JSON and XML are used for packaging information in the Ajax model. The Fetch API provides an interface for fetching resources. It will seem familiar to anyone who has used XMLHTTPRequest, but this API provides a more powerful and flexible feature set.

Server-sent events Traditionally, a web page has to send a request to the server to receive new data; that is, the page requests data from the server. With server-sent events, it's possible for a server to send new data to a web page at any time, by pushing messages to the web page. These incoming messages can be treated as *Events + data* inside the web page. See also: Using server-sent events.

2.2.2 WebSocket

The **WebSocket API** is an advanced technology that makes it possible to open a two-way interactive communication session between the user's browser and a server. With this API, you can send messages to a server and receive event-driven responses without having to poll the server for a reply.

The primary interface for connecting to a WebSocket server and then sending and receiving data on the connection. The event sent by the WebSocket object when a message is received from the server. Either a single protocol string or an array of protocol strings. These strings are used to indicate sub-protocols, so that a single server can implement multiple WebSocket sub-protocols (for example, you might want one server to be able to handle different types of interactions depending on the specified protocol). If you don't specify a protocol string, an empty string is assumed.

The constructor will throw a Security Error if the destination doesn't allow access. This may happen if you attempt to use an insecure connection (most user agents now require a secure link for all WebSocket connections unless they're on the same device or possibly on the same network). WebSockets is an event-driven API; when messages are received, a message event is sent to the WebSocket object[5]. To handle it, add an event listener for the message event, or use the on message event handler. To begin listening for incoming data, you can do something like this.

2.2.3 WebRTC

WebRTC or Web Real Time Communications allows the establishment of real time audio and video communications between users' browsers. The creation of the communication link is mediated by a web server with WebRTC capabilities. The link itself can be peer-to-peer between the browsers. WebRTC does not depend on a standardised signalling infrastructure, but rather leverages the web JavaScript environment and standardised browser APIs. This allows for implementations that range from a simple audio communication between two people

up to a videoconference with multiple participants, out of the box as part of a normal browser. WebRTC thus has at least three actors involved: a web server and two browsers. The mediation of the usable communication channels is negotiated between these based on a complex set of specifications that are detailed in the WebRTC overview below. Not only is there a complex combination of specifications, but the work is also distributed between the IETF doing the protocol stack and W3C creating the browser APIs[6]. Still WebRTC remains a part of the Web, thus all the vulnerabilities described in the Web Security Guide still apply. But the complexity and the fact that the communication is real time also bring new aspects and vulnerabilities. This case study dives deeper into the vulnerabilities to which user and server assets are exposed by their use of WebRTC. This is not limited to the three actors mentioned, but these may also behave in unexpected ways. Chapter 2 describes the relevant assets, and based on the list of assets, the surface of exposure and the assets targeted and the list of possible malicious actors becomes much clearer. This exposes some of the underlying issues otherwise hidden by the complexity of the combination of WebRTC specifications. The fact that real time communication is involved immediately raises the issue around permissions and timing of access. For communication between two users, the browser needs access to speaker and microphone. This means the browser can potentially be remotely turned into a device to capture all the sounds in a location. It is known that native applications on mobile phones have issues around permissions and how fine grained they ought to be, whereas how coarse they actually are. For WebRTC the issue is the same. This case study assesses the implications and gives hints on potential security problems in current implementations. A central issue in WebRTC that was discussed widely in the Working Groups but has not yielded satisfactory results is identity management[7]. Am I really sure I am talking to the right person? On the Web, the trusted telephone operator is not always there. The case study looks into user authentication and naming and raises issues around credentials and keying. The case study then goes on with an analysis of a classic cross site scripting attack in a WebRTC scenario. What happens when an attacker can inject arbitrary JavaScript code into the running WebRTC application or even manages to upload a malicious WebRTC application to a server that delivers it to the browser? The evaluation of exposure is used to describe a landscape of possible attacks.

CHAPTER – 3

Requirements & Tools of Project

3.1 Hardware Requirements

RAM: At least 256 MB of RAM. The amount of RAM needed depends on the number of concurrent client connections, and whether the server and multiplexor are deployed on the same host.

Disk Space: Approximately 300 MB required for Instant Messaging Server software.

Processor: Minimun 1.9 gigahertz (GHz) x86- or x64-bit dual core processor with SSE2 instruction set and recommended 3.3 gigahertz (GHz) or faster 64-bit dual core processor with SSE2 instruction set.

Memory: Minimum 2-GB RAM and recommended 4-GB RAM or more

3.2 Languages and Libraries

Html

HTML (Hypertext Markup Language) is the code that is used **to structure a web page and its content**. For example, content could be structured within a set of paragraphs,a list of bulleted points, or using images and data tables

CSS :

CSS (Cascading Style Sheets) is used **to style and layout web pages** — for example,to alter the font, color, size, and spacing of your content, split it into multiple columns, or add

animations and other decorative features.

JavaScript:

JavaScript is a text-based programming language used both on the client- side **and server-side that allows you to make web pages interactive**. Where HTML andCSS are languages that give structure and style to web pages, JavaScript gives web pages interactive elements that engage a user.

ReactJS:

ReactJS is one of the most popular JavaScript front-end libraries which has a strong foundation and a large community .It is a **declarative**, **efficient**, and flexible **JavaScript library** for building reusable UI components. The main objective ofReactJS is to develop User Interfaces (UI) that improves the speed of the apps. It uses virtual DOM (JavaScript object), which improves the performance of the app.

NodeJS:

Node.js is an open source server environment. These allows you to run JavaScript on the server. It is primarily used for **non-blocking, event-driven servers**, due to its single-threaded nature. It's used for traditional web sites and back-end API services, but was designed with real-time, push-based architectures in mind.

Mongo DB:

MongoDB is an open source NoSQL database management program. NoSQL is used as an alternative to traditional relational databases. NoSQL databases are quite useful for working with large sets of distributed data. MongoDB is **a tool that can manage document-oriented information, store or retrieve information**.

3.3 Software Requirement

VS Code:

Visual Studio Code is a **streamlined code editor with support for development operations like debugging, task running, and version control**. It aims to provide just the tools a developer needs for a quick code-build-debug cycle and leaves more complex workflows to fuller featured IDEs, such as Visual Studio IDE.

Postman:

Postman is an application used for **API testing**. It is an HTTP client that tests HTTP requests, utilizing a graphical user interface, through which we obtain different types of responses that need to be subsequently validated.

Robo3t:

Robo3T (formerly Robo-3T mongo) is a popular **desktop graphical user interface (GUI) for your MongoDB hosting deployments** that allows you to interact with your data through visual indicators instead of a text-based interface.

CHAPTER – 4

Project Overview

4.1 Project Methodology

Agile is a set of techniques to manage software development projects. It consists in:

- Being able to respond to changes and new requirements quickly.
- Teamwork, even with the client.
- Building operating software over extensive documentation.
- Individuals and their interaction over tools.

We believed it was a perfect fit for our project since we did not know most requirements beforehand. By using the Agile, we were able to focus only on the features which had the most priority at the time.

4.1.1 Use Cases and Scenarios

User stories are one of the primary development artifacts when working with Agile methodology. A user story is a very high-level definition of a requirement, containing just enough information so that the developers can produce a reasonable estimate of the effort to implement it. Gathered from stakeholders (people, groups or organizations who are interested in the project), they show us what we have to work in. Since we were working with Agile, this list did not have to be complete before we started working on the project, but it was desirable to have at least a few items to start with so that we could establish proper feature priorities. At the commencement of every sprint, we analyzed all user stories, estimated the value they added to the project and the amount of time they would take us doing each of them, and sorted them by descending order — placing the user stories which had the most added value and the least

time cost at the top. The value was quite subjective. We gave the highest priority to features which we believed they were essential to the platform (such as instant text messages) or were very related to the chat's topic — coding. We gave them a score from 1-10. Time cost was an estimation of how much we thought an individual story was going to take to implement. The measurement was done in days, considering each working day to be as long as 4 hours. We then translated this value as follows:

• 1-2 days: 1

• 3-4 days: 2

• 5-6 days: 3

• 7-9 days: 4

• 10+ days: 5

To sort both the product backlog and sprint backlog lists, we relied on a third number, the priority, which was simply the result of the value minus the time cost. Nonetheless, in some cases, we had to make exceptions due to user stories dependencies. For example, sign in and sign up features had to be implemented the first, since we needed user information to properly identify the room owner or the message sender.

4.2 Technology

The architecture of the application consists of the back end and the front end, both of them having their own set dependencies (libraries and frameworks). The front end is the presentation layer that the end user sees when they enter the site. The back end provides all the data and part of the logic and it is running behind the scenes.

4.2.1 Back End

The "back end" refers to the logic and data layers running on the server side. In our case, the back end makes sure that the data introduced through the client application (the front end), is valid. Since the front end can be avoided or easily manipulated (the source code is available to

the end user) we have to make sure that all the requests we receive are first verified by the server: the requested URI is supported, the user has the appropriate permissions, the parameters are valid, etc.

If the request data is valid, we do often proceed to execute some logic accompanied by one or more database accesses.

API

Our application is all about I/O. We were looking forward a programming environment which was able to handle lots of requests per second, rather than one which was proficient at handling CPU-intensive tasks. At the moment it seemed like the choice was between PHP, Python, Java, Go or Node.js. These languages have plenty of web development documentation available, and they have been widely tested by many already. The trendiest choice in 2016 was Node.js, which was exceptional for handling I/O requests through asynchronous processing in a single thread. So we went for Node.js not only because of the performance but also because of how fast it was to implement stuff with it, contrary to other languages such as Java which are way more verbose. For web development, we would then use Express, which makes use of the powerfulness of Node.js to make web content even faster to implement. A feasible alternative to Node.js would be Go, which is becoming popular nowadays due to somewhat faster I/O than Node.js with its Go subroutines, and unquestionably better performance when doing intensive calculations[4] (though we were not particularly looking for the last one). Nonetheless, Go meant slower development speed. It lacked libraries as it was not as mature as Node.js and the cumbersome management of JSON made it not very ideal for our application (since the JavaScript client would use JSON all the time). We are writing Node.js with the latest ECMAScript ES6 and ES2017 standard supported features. The development was started with ES6, but we also used a few features originally from ES2017 as soon as Node.js turned to v7. ES6/ES2017 standards differ from the classic Vanilla JavaScript in that they have a few more language features and utilities out of the box which makes code easier to read, faster to write and reduce the need to make use of external libraries to do the most common operations. For example, Promises over callbacks or classes over functions, even though they are just syntactical sugar. A few remarkable frameworks/libraries we are using on the development of the application are:

Express

A Node.js framework which makes web development fast. It abstracts most of the complexity behind the web server and acts as an HTTP route handler. It can also render views (a sort of HTML templates with variables) but are using the front end application for this instead. By using Express, we are able to focus on the logic behind every request rather than on the request itself.

Mongoose

A MongoDB high-level library. By using objects as database models, which will later end up being the data inside our collections, it handles inserts, updates, and deletes, as well as the validation for each of its fields.

Passport

An authentication library build specifically for Node.js. By using the different login modules (one module per provider), it hides all the complexity behind OAuth, OAuth2, and OpenID. Passport commits to notifying the developer in the same way regardless of the authentication method they have chosen. We are using Passport to handle GitHub and Google authentication, as well as the local one (email + password).

Sinon

An extensive testing library that has a set of useful utilities: spies, stubs, and mocks. Throughout our tests, we often feel the need to know whether a certain function has been called, has been called with the right parameters, or even to fake external incoming data to ensure that we are testing solely what we want to test.

Socket.io

A JavaScript library which handles WebSocket connections. It abstracts most of the complexity behind WebSockets, and it also provides fallback methods which work without any special configuration. Socket.io takes care of the real time updates in our application, such as sending or receiving messages.

Data storage

We believe that NoSQL is the future. Hence, we did not hesitate to choose to use NoSQL storages only. Why choosing NoSQL databases over the traditional SQL ones?

- They are more flexible: you can access nested data without having to perform any join.
- They are faster[6]: nested data is stored in the same place and can be consulted without any additional query.
- They scale better[7] when distributing the data over different nodes.
- There are many types of NoSQL databases which fit for different kinds of work, such as Key-Value for sessions or Document-based for complex data.

At first, we were going to go with MongoDB only, but later we realized that it would be a good idea to have Redis as well to map session keys with user identifiers.

MongoDB is a schemeless document-oriented database. It gives us the possibility to store complex data effortlessly and retrieve it straight away, without any additional query. Although the data is always stored on disk, it is very fast and highly scalable.

We are using MongoDB to store any persistent data, such as user details and preferences, rooms information and chat messages. You can find more details about the MongoDB document modeling on the implementation chapter.

Redis is a key-value data structure, which uses memory storage to perform searches by a given key very quickly. Queries are performed faster than in MongoDB but there is also a higher risk of losing data, and it cannot process complex values (such as nested documents).

Although Redis performs better than MongoDB, we cannot rely on it for critical data. For this reason, we are only using it to store user sessions, which in the case of loss, would only mean that the user would have to re-login to keep using our platform. Nonetheless, we expect to performance gains to be noticeable when the site is at its peak capacity because the session data is something we are are looking up in every single request to the API.

4.2.2 Front End

Having separated the server-side from the client side, a SPA (Single-Page Application) was an outstanding choice. SPAs dynamically fetch data from the API as the user is browsing the site, avoiding to refresh the whole page whenever the user has filled in a form or navigated to another part of the site. The UX boost a SPA can get over a traditional website is very significant. It is true that it often takes longer to load for the first time, due to having to download a bigger JavaScript file chunk, but once loaded the delay between operations is minimal which leads to a more fluid User experience, and less bandwidth use in most cases.

Implementing a scalable Single-Page Application by using Vanilla JavaScript only would take an enormous amount of time, since it has none of the high-level utilities that make it simple to develop one of this kind, such as a high-level HTML renderer that allows you to build elements on the fly, storage or router. Hence, it made sense to choose an actively maintained and documented framework/library to start with.

At the time, the decision was between Angular, React and Vue.

Both Angular and React were being maintained by powerful corporations, Google and Facebook respectively, so we had a brief look at their documentation and developers' reviews before taking our final choice. Eventually, we chose React.

React is a very powerful library with an enormous ecosystem (you can find many utilities that were meant to be used with React). It is featured due to its fast performance and small memory consumption, which is especially useful when targeting mobile devices. Moreover, there is a plethora of documentation on its official site and around the Internet. The library main features are: Tree Structure A React page always starts with a single root component (tree node) rendered in a pre-existing HTML element on the page.

Babel

A few users coming to our site might be using old browser versions, which have little to no support to ES6/ES2017 features. To make sure all browsers can understand our code we make use of Babel, which transpires our modern JavaScript code into JavaScript code that most browsers can understand.

Redux

An in-memory storage for JavaScript. It saves application states, which in other terms are the different data that our application uses over the time.

A storage like Redux avoids having to transfer data up and down the React tree, since Redux stores it all in one place which can be accessed anytime.

It is also modular, which makes it ideal for our application since it helps towards scalability. That does not mean that it makes properties and functions we explained earlier become redundant. We should still use these for simple or very specific interactions with components. Nonetheless, Redux simplifies things when working especially with global variables, such as the currently authenticated user.

Redux was initially built for React, so it works hand to hand with it. The storage can be easily connected to React components, which will have access to any of the stored data and also be able to dispatch new actions to add/update the data in it.

4.2.3 Version Control

A version control system can be useful to developers, even when working alone. It enables us to go back in time to figure out what broke a certain utility, work on different features at the time and revert/merge them into the original source code with no difficulty, watch how the project evolved over the time, and so on.

We chose Git. Not only because it is the most popular and widely used version control system, but also because part of our project was the integration with GitHub, and GitHub works with Git.

For the same reason as above, we chose GitHub to be our remote source code repository. Currently, it is a public space where developers can come and have a look at the source code that is powering the chat application, report bugs they encounter or even contribute by submitting pull requests.

CHAPTER -5

Working and Functionality of project

This chapter details the most relevant parts of the application development, decisions taken and algorithms.

We have divided this chapter into three sections: databases (design), features (the most important ones) and a brief overview on how we tested our features.

5.1 Databases and Models

A key defining aspect of any database-dependent application is its database structure. The database design can vary depending on many different factors, such as the number of reads over writes or the values that the user is likely to request the most. That is because as full stack developers we want the database to have the best performance, which can often be achieved by focusing the optimizations on the most common actions.

We concentrated on the MongoDB database, which is the most complex data storage and the one which stores the most data.

Our MongoDB data structure limits to mapping sessions to user identifiers, both of type text. That is how a web request works: Node.js queries MongoDB by using the user session identifier to determine whether the user is signed and their account identifier. If an account identifier is found, Node.js queries MongoDB to find out the rest of the user information. The MongoDB database stores everything else: users' information, rooms, chats, and messages. 32 Implementation of a chat application for developers Our final database design ended up having four different collections: users, rooms, chats, and messages. Although MongoDB is schema-less, by using the Mongoose library on Node.js, we were also able to define a flexible schema for each of the collections. A schema constrains the contents of a collection to a known format, saving us from validating the structure of the data before or after it has been put in into the database.

5.1.1 Users

To start, we needed somewhere to store our users. Since we were expecting a significant number of entries, an individual collection for the users' themselves was the most appropriate. What we mean by that is that it was best for the users' collection to solely store the information that made reference to their authentication and personal data. Their rooms, chats, and messages should be stored somewhere else. Given that we were expecting a lot of rooms, chats, and messages per user, we refrained from even making references to them in this collection. We are querying these other collections directly. Schema fields:

- _id: identifier.
- username: friendly identifier.
- email: email address.
- password: encrypted password.
- passwordResetToken: token to reset their password.
- passwordResetExpires: expiration date of the password reset token.
- GitHub: GitHub's profile id.
- google: Google's profile id.
- tokens: list of linked services tokens.
  - kind: service name (i.e., GitHub)
  - accessToken: access token given by the service.
- profile: personal details

Users' collection is indexed by _id, username, email, github, and google fields. These cover most searches, which is what is being done the most often: users are being looked up many times whereas they barely change during their lifetime.

For example, we are searching the associated user through the _id field on every request, but we only set the _id on their creation. Moreover, we are referring to the email, github, and google identifiers every time a user logs in through each respective method, yet most times these identifiers are only set once during the user's lifespan.

Although we did not specify, some of the schema fields are required, whereas others can be left undefined. All these specifications, including each of the fields' validation, were given to Mongoose, either in the form of configuration or functions.

5.1.2 Rooms

Given that we were not going to store rooms as nested data inside the users' collection, mainly because we were looking forward to referring to them directly, we created an independent collection for them. In addition, we were expecting many rooms, probably even more than users. Hence it was not a not a good idea to nest them under any other document. Schema fields:

- _id: identifier.
- title.
- slug: room URL identifier.
- description.
- owner: _id of the owner user.
- isPrivate: whether the room is private or public.
- members: array of user _id who are members of that room.
- updatedAt: modification date.
- createdAt: creation date.

Notice that once again we are not storing any of the chats inside it, not even the reference. Although some would argue that it would not be a bad idea, in this case, we preferred storing them on an individual collection given that we were expecting many due to the ability to fork

chats.

Other chat applications which set a limit of 10-20 chats per room, should consider either embedding the whole chat object inside their room or at least store a reference to them.

On the other hand, we are storing a reference to the members of a room. That is because we are not expecting more than few hundred users per room and they are also not a clear entity by themselves and the disk space these references take does not look like to be a problem.

When designing MongoDB collections, we always have to keep in mind that the maximum size per document is 4MB (16MB in the latest versions).

The other field which we are also storing by reference is the owner of the room. The reason why we are not embedding the user, in this case, is not because of the size, but rather because the user profile data might frequently be updated which would mean having to update all rooms he owns, apart from the corresponding User.

We are indexing Rooms by _id, slug, owner, and members.

At first, we thought _id and slug would suffice since they cover the most common searches: users referring to a chat through its identifier or entering through a direct URL (in which case the lookup would be done by the URL slug).

However, later we realized that users might often want to look up chats which they either own or are members of, which is the reason why we created two additional indexes to cover the owner and members.

5.1.3 Chats

As we stated earlier, our chats were going to be in individual collections. There might be rooms in which their members have few chats, but others might have hundreds (even if that leads to having a few inactive ones).

Once again, we had to think whether it was worth embedding or referring messages inside the Chats collection or keeping them isolated in another one.

In this case, it was evident. We were expecting thousands of messages in any Chat, which would rapidly go over the 16MB that any MongoDB document can hold, even if only storing

references.

Thus, messages had to be saved in a different collection. Schema fields:

- _id: identifier.
- room: identifier of the room it belongs to.
- title.
- description.
- github: GitHub repository name, taken into consideration when creating GitHub specific chats.
- firstMessageAt: date of the first message sent. It is used to determine whether the user has already retrieved all messages of a chat.
- lastMessageAt: date of the last message sent.
- updatedAt: modification date.
- createdAt: creation date.

We are indexing Chats by _id, and room. _id is used everytime someone wants to enter a specific chat, whereas the room one makes it quicker to search the chats inside a Room.

5.1.4 Messages

We were expecting thousands of messages per month, so the right way to store them, according to the MongoDB official documentation, is in an individual collection. In a production environment on which we were expecting even millions of them, we might have to consider sharding the data to avoid bottlenecks, which is a topic which we have briefly covered in the Evaluation chapter.

This case is similar to the Rooms or Chats ones, but this time it is taken to the extreme, "One-to-Squillions".

We were no longer just expecting to store thousands in the long-term run, but we were expecting to store thousands at a fast growing pace.

We can summarize our Messages needs as follow:

1. Ability to store hundreds of messages per hour.
2. Ability to retrieve thousands of messages per hour
3. Ability to retrieve messages in chronological order (most recent first).

Notice that we are expecting to read more than to save. That is because a few chat peers are likely to retrieve recent messages more than once, and while a message is only stored once, several members are likely to read it numerous times. Thus, we wanted to design a collection schema which favored reads over writes.

Moreover, we would never want to retrieve all messages at once. Not only it would be impossible for the user to read them all, but also we would not be able to handle the load if we did that for Chats having many messages.

Schema fields:

- _id: identifier.
- chat: identifier of the chat it belongs to.
- owner: sender (User) identifier.
- content: text.
- type: content type.
  - language: code language, if the message type is code.
  - highlight: lines to highlight, if the message type is code.
  - chat: reference to the parent Chat, if the message is a fork.
- deletedAt: deletion date. Content will be removed on deletion, but their peers will be aware that the User sent a message at that time.

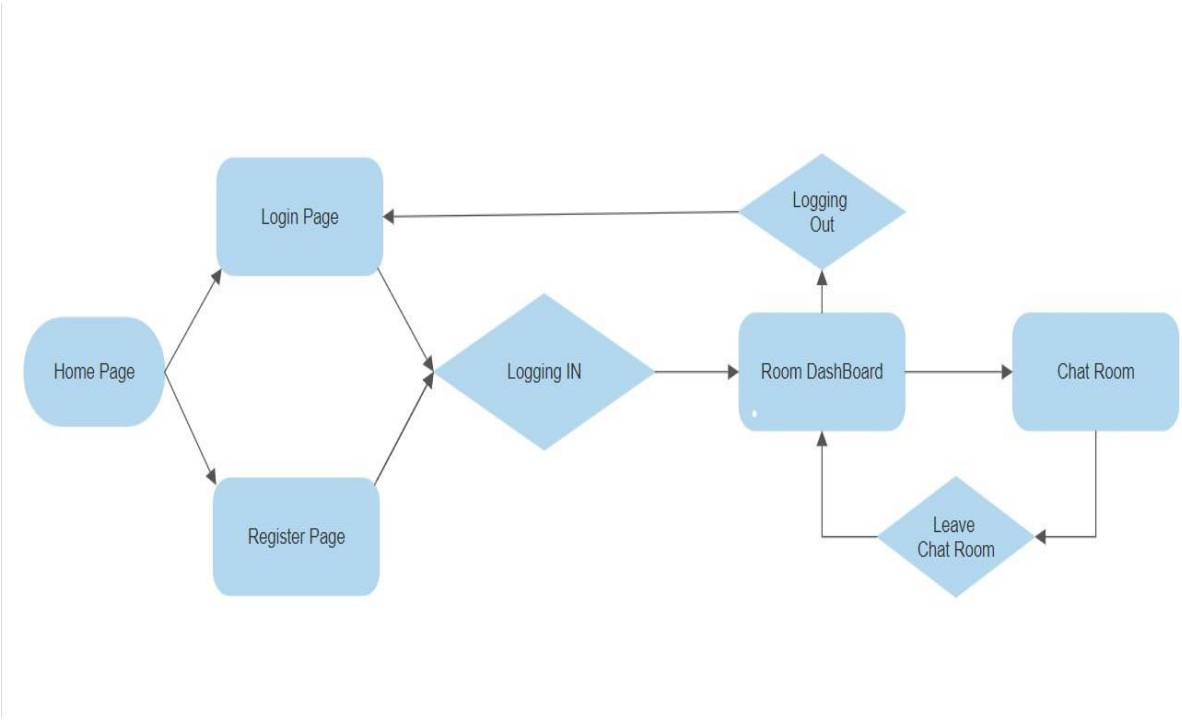
- updatedAt: modification date.
- createdAt: creation date.

As simple as it seems, this structure has been proved to work out for up to 1,000,000 concurrent connections.

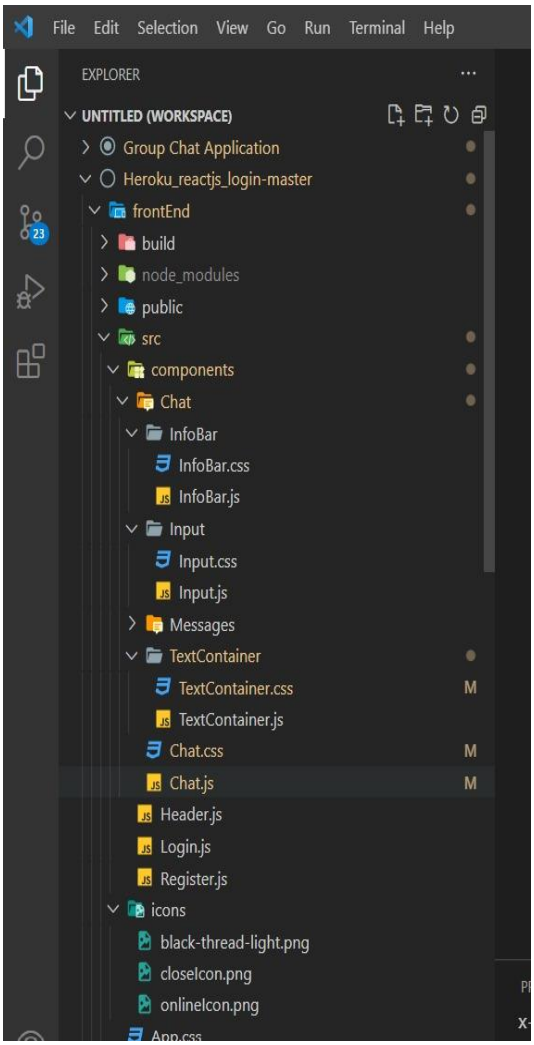
We have indexed messages by _id and chat + createdAt. The first one helps when looking for a specific message, whereas the second composed index works out well when looking for past messages. We have composed the date with the chat to filter only the messages which belong to a particular chat since we will never be interested in mixed chat messages.

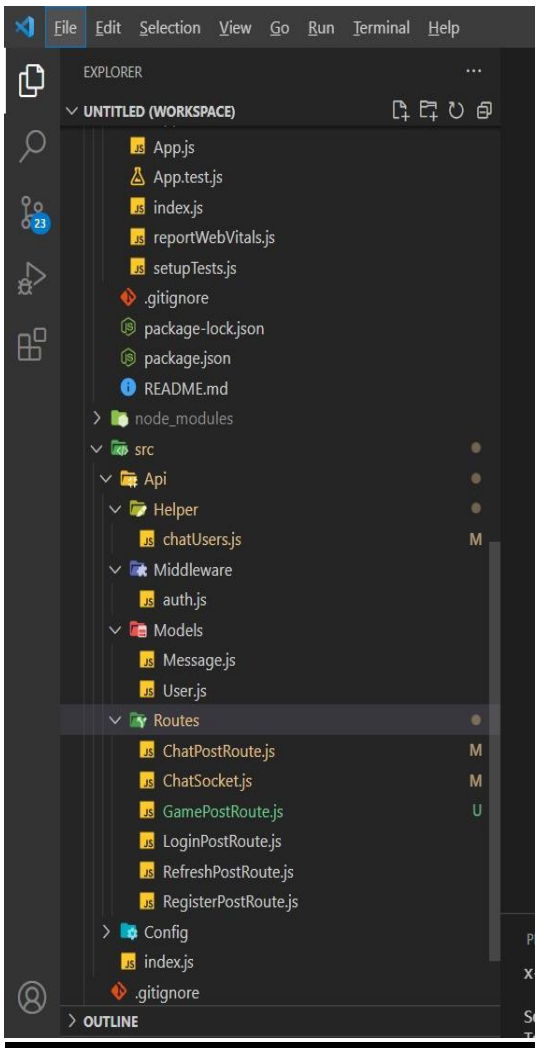
5.2 Flow Chart

Initially the user will be on the home page where he / she will receive the option for the Login or the Register page. After the user credentials are verified, the user will be redirected to the Room dashboard, where users can create or join chat rooms and use all the chat features provide. There is also an option for the user to logout which ill redirect them to the Home page.

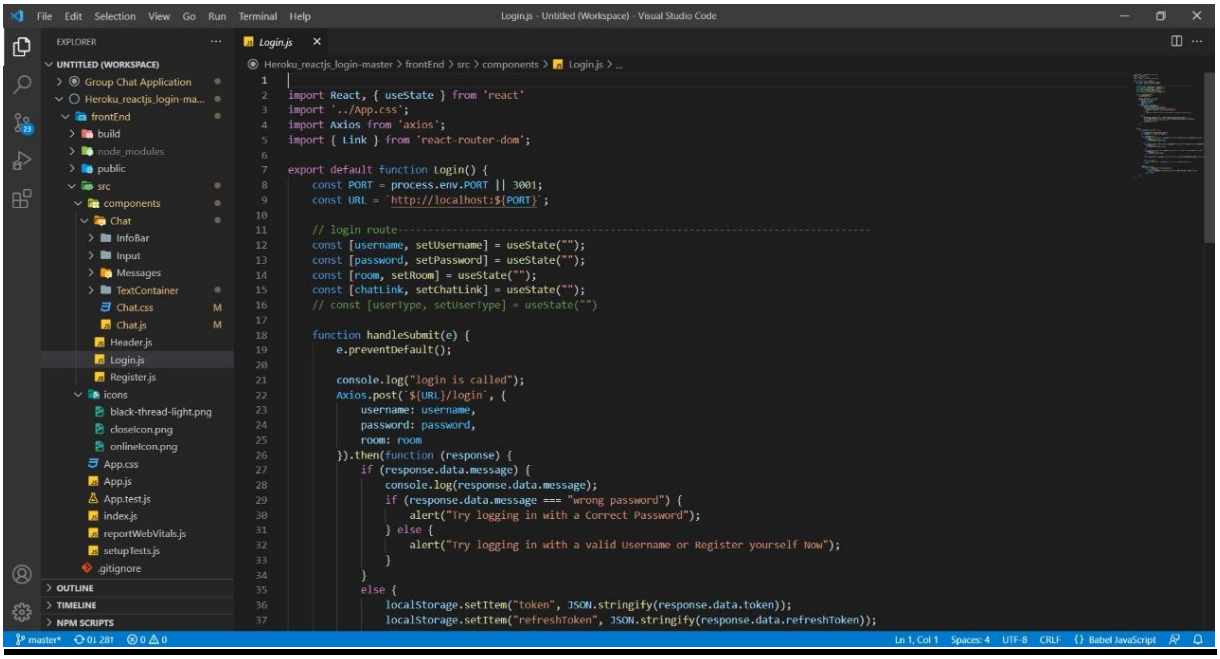


5.3 Project code snippets

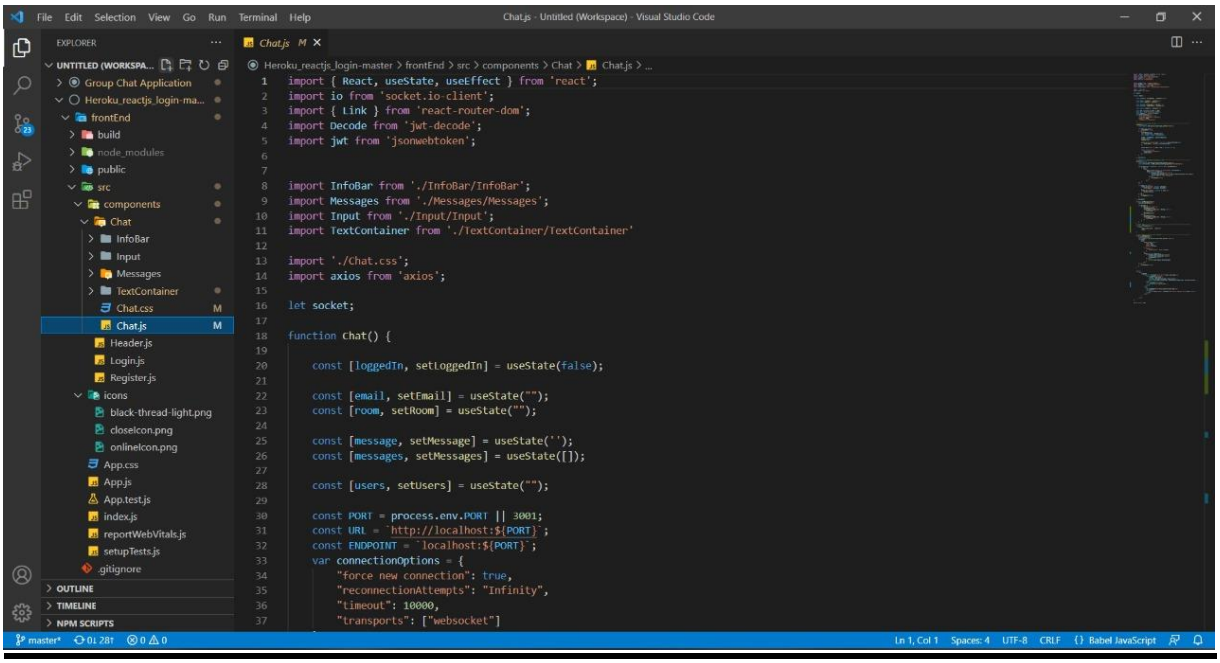




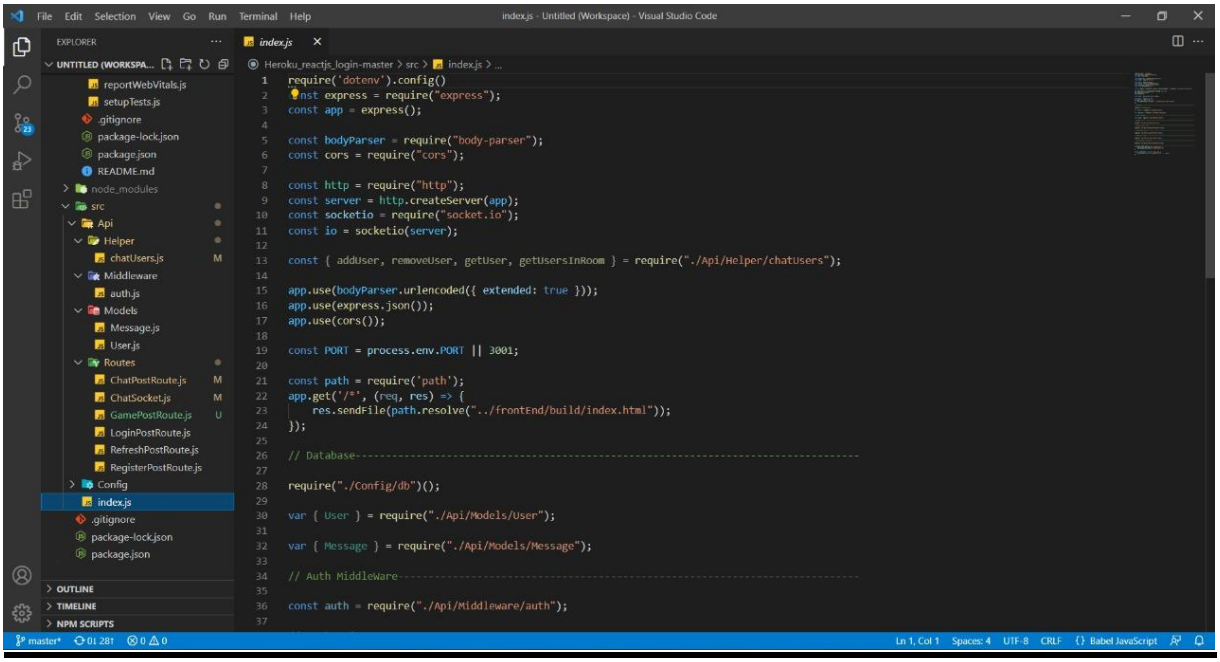
Code for Login.js



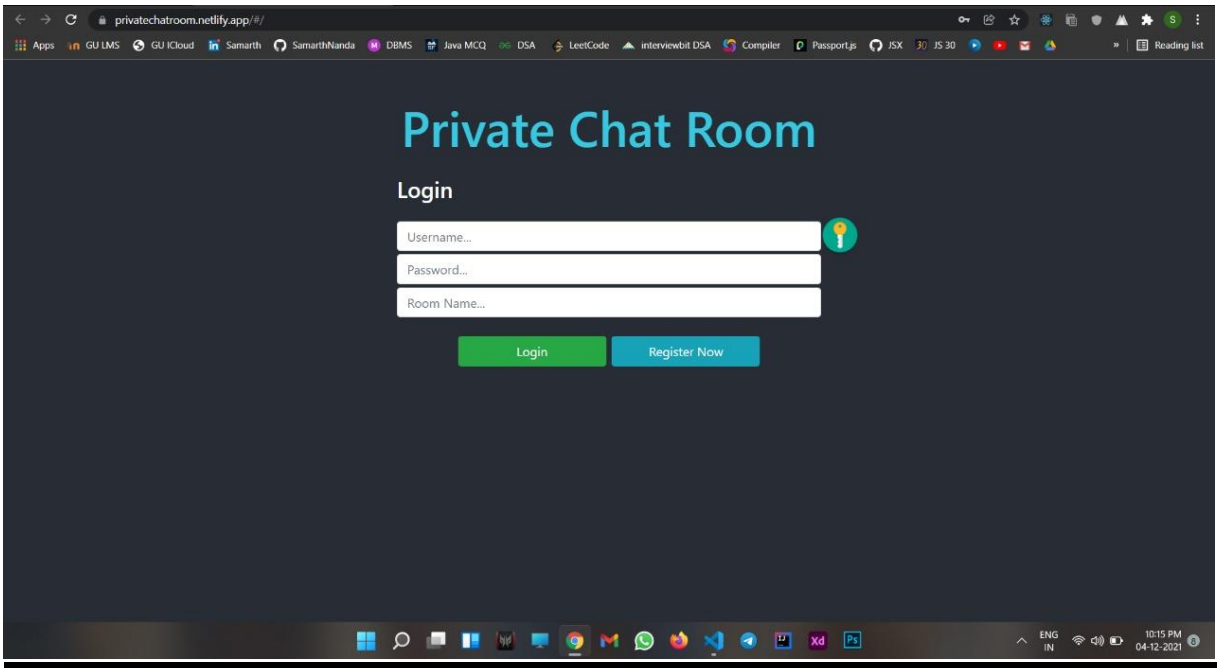
Code for Chat.js

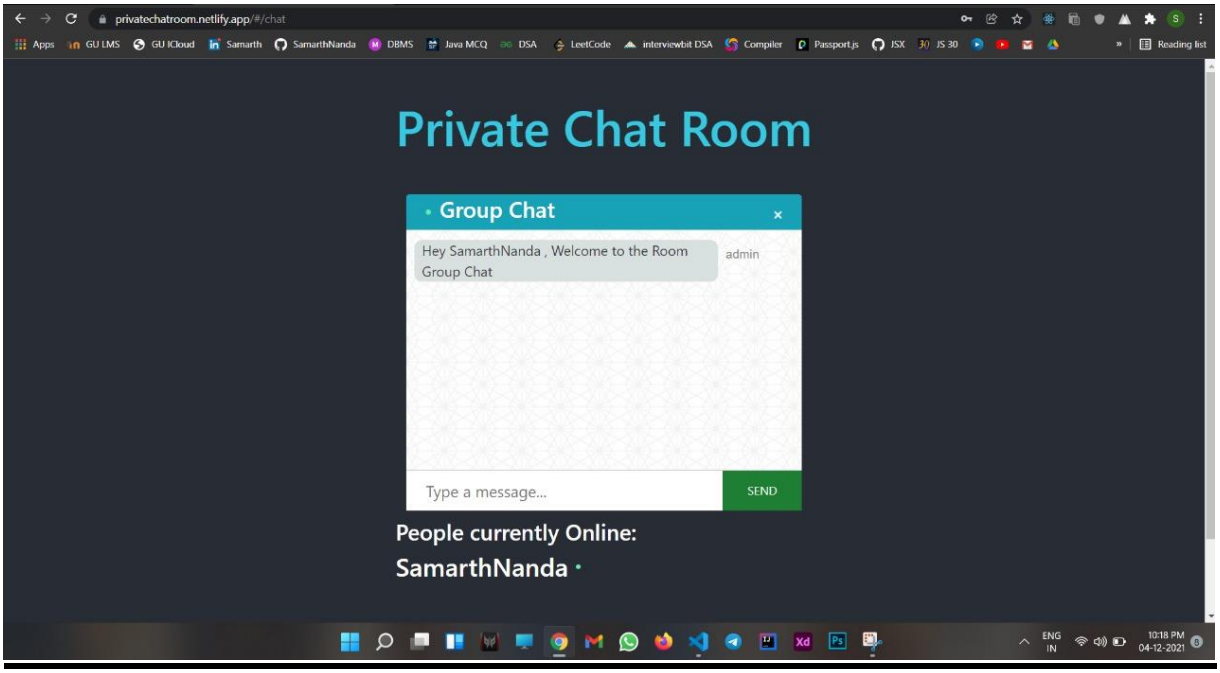


Code for index.js



Output




privatechatroom.netlify.app/#/chat
Private Chat Room
Group Chat
Hey SamarthNanda , Welcome to the Room Group Chat
admin
Type a message...
SEND
People currently Online:
SamarthNanda

CHAPTER-6

Results and Discussion

The Group Chat application creates a platform for users to communicate to one another. Chat apps are dynamic tools that allow workers to engage with one another, share meaningful ideas, work through company problems and better plan for your business's future. They often offer task management features, chat features, video calling services, and other communication and productivity management tools.

The goal of these communication tools should be to simplify things, not make them more complicated.

CHAPTER -7

Conclusion And Future Scope

7.1 Conclusion

There is always a room for improvements in any apps. Right now, we are just dealing with text communication. There are several chat apps which serve similar purpose as this project, but these apps were rather difficult to use and provide confusing interfaces. A positive first impression is essential in human relationship as well as in human computer interaction.

7.2 Future Scope

Although the application itself works well, much was learned during its development.

For this reason, we wrote a list of possible improvements/changes, some of which are easy to execute, others might require rewriting a significant amount of the current source code. Apart from that, the Ideal application was too ambitious, which resulted in many features not being able to be implemented during the course of the project.

Express to Hapi

Express works well for small projects, it is easy to set up and you can have an API working within minutes. However, it is very minimalistic. As the project gets bigger, you are forced to write much middleware code yourself, which does not only take time but it can lead to security risks if not properly tested. Hapi is a more modern Node.js framework, with security in mind and designed to handle big loads. Hapi by itself can handle things such as input validation, server-side caching, cookieparsing or logging. Although moving to Hapi is not a requirement, we believe it is a wise move since it would ease a lot of future work.

Move the whole API to an MVCS architecture

Back when we started our back end, we had models, simple controllers, and JSON

responses as our views. Nonetheless, as the application grew, a few controllers logic 88 Implementation of a chat application for developers code got huge and repetitive.

In some cases, we even needed to share logic between different application topics (i.e. authentication and chat rooms).

In order to fix this, we started by abstracting controllers into separate functions, but the separation of concerns was not clear. We had controllers, and controller "helpers".

This project hopes to develop a chat service Web app with high quality user interface.

In future we may be extended to include feature such as:

•File Transfer

•Voice Message

•Audio Call

•Group Call

•Video Call

•Event Managing

Remaining features

The model platform, described in the "Features" section, had plenty of features. Many of them remain undone:

• Notifications

• Status

• Room roles

• File sharing

• Voice and videocalls

• Public API

• etc.

While our application already provides the basics to programmers who want to talk and share code themselves, having more of these model features done would probably attract the attention of more of them.

REFERENCES

1. Ammar H. Ali, Ali Makki Sagheer (2017). Design of a secure android chatting application using end to end encryption, Journal of software engineering & intelligent systems.

2. Ms. Swara Pampatwar. Vinisha Kalyani, Prachi Shamdasani, Urja Ramwani (2020). Multi-layered Data Encryption/Decryption Chatting Application, Annals of R. S. C.

3. Ben Feher, Lior Sidi, Asaf Shabtai, Rami Puzis (2016). The Security of WebRTC, arXiv,

4. Kwok-Fai Ng, Man-Yan Ching, Yang Liu, Tao Cai, Li Li, Wu Chou (2014). A P2P-MCU Approach to Multi-Party Video Conference, International Journal of Future Computer and Communication.

5. Mohamed, M. A., Muhammed, A. & Man, M. (2015). A Secure Chat Application Based on Pure Peer-to-Peer Architecture. Journal of Computer Science, 11(5), 723-729.

6. J. Rosenberg and H. Schulzrinne. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264 (Proposed Standard), June 2002. Updated by RFC 6157.

7. P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Address Format. RFC 6122 (Proposed Standard), March 2011.

8. P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120 (Proposed Standard), March 2011.