



---

# VV CHAT WEB APP

## Full Stack Messaging Service

---

System Design  
Document

---

Version: 1.0.0

Date: 03/09/2024

## Table of Contents

<b>1. Introduction-----</b>	<b>1</b>
1.1. Software Requirements Specification -----	1
1.2. Dev Dependencies Documentation -----	1
<b>2. System Overview-----</b>	<b>4</b>
2.1. System Description-----	4
2.2. System Context -----	5
<b>3. Architecture Design-----</b>	<b>6</b>
3.1. High-Level Architecture -----	6
3.2. Component Diagram -----	7
3.3. Technologies Used -----	11
<b>4. Detailed Design-----</b>	<b>13</b>
4.1. Frontend Design -----	13
4.2. Backend Design -----	23
4.3. Database Design -----	37
4.4. Security Design -----	40
<b>5. Non-functional Requirements-----</b>	<b>42</b>
5.1. Performance & Scalability -----	42
5.2. Reliability and Availability-----	42
<b>6. Deployment-----</b>	<b>43</b>
<b>7. Error handling and Logging-----</b>	<b>45</b>
<b>8. Testing-----</b>	<b>48</b>
<b>9. Abbreviations -----</b>	<b>49</b>

## List of Figures

Figure 01 => VV CHAT Web Application Landing page -----	4
Figure 02 => VV CHAT High Level Architecture Diagram -----	6
Figure 03 => MongoDB Semi-structured datatypes Rep -----	6
Figure 04 => Classification of “auth” & “root” directories-----	7
Figure 05 => Implementation of Ternary Operator -----	7
Figure 06 => Single Responsive Form for Login & Register pages ----	7
Figure 07 => Routing structure of frontend app components -----	8
Figure 08 => Encapsulation of Styling logic for the entire UI -----	9
Figure 09 => Broad Classification of Backend components -----	9
Figure 10 => Connection of Database via “mongoose” -----	10
Figure 11 => Implementation of Next-Auth middleware pattern -----	11
Figure 12 => List of Packages used, along with version details -----	12
Figure 13 => Illustration of implementing Atomic Design practice ---	13
Figure 14 => Implementation of VV CHAT Top Bar -----	14

Figure 15 => Illustration of the bottom bar of web app -----	15
Figure 16 => User Interface of Login and Register -----	16
Figure 17 => Illustration of User Alerts via “Toaster” -----	16
Figure 18 => Implementation of “useSession” React Hook for Login -	17
Figure 19 => Illustration of Loading icon used in the web application -	17
Figure 20 => Depiction of MessageBox UI component -----	18
Figure 21 => Illustration of searching of a contact from the UI -----	19
Figure 22 => Illustration of Pusher Channel -----	19
Figure 23 => Illustration of searching of a chat from the UI -----	20
Figure 24 => Diagram depicting writing of a message in a chat -----	21
Figure 25 => Illustration of Cloudinary Upload functionality -----	21
Figure 26 => Diagram depicting list of all chat conversations -----	22
Figure 27 => Illustration of retrieval of specific group chat from UI --	23
Figure 28 => Block Diagram of RESTful API -----	23

Figure 29 => Execution of GET request for Register -----	24
Figure 30 => Depiction of “User Already Exists” use case -----	24
Figure 31 => 401 Unauthorized response for Invalid Login entry ----	25
Figure 32 => Web app retrieving registered user data for credentials comparison -----	25
Figure 33 => Backend API call for starting a new chat or opening an existing chat -----	26
Figure 34 => Response Preview of POST request which is responsible for starting a new chat or opening an existing chat -----	26
Figure 35 => Backend API call to display the past chatted messages -	27
Figure 36 => Response Preview of GET request which is responsible for displaying the past conversation chats -----	27
Figure 37 => Backend API call when a new message is sent to the receiver from the web application -----	28
Figure 38 => => Response Preview of POST request which is responsible for sending new message from the UI -----	28
Figure 39 => Backend API Call for updation of Group Chat information -----	29

Figure 40 => Response Preview of POST request which is responsible for appending new message record into the database -----	29
Figure 41 => Payload of the POST request which is responsible for appending new message record into the database -----	30
Figure 42 => Backend API Call for sending a new message from the web application -----	30
Figure 43 => GET Request to fetch the details of all Registered Users -----	31
Figure 44 => Backend API Call which is responsible for searching a contact from the UI -----	32
Figure 45 => Backend API Call which is responsible for fetching all previous chat conversations -----	33
Figure 46 => Backend API Call which is responsible for handling user modifications -----	34
Figure 47 => Payload when new profile picture is uploaded for a given user from the UI -----	34
Figure 48 => Response Preview when new profile picture is uploaded for a given user from the UI -----	35
Figure 49 => Backend API Call which is responsible for searching a chat from the UI -----	35

Figure 50 => Response Preview when searching of a chat is done from the UI -----	36
Figure 51 => Diagram of the Database used in VV CHAT Web Application -----	37
Figure 52 => Database Schemas Representation Diagram of VV CHAT Web Application -----	38
Figure 53 => Illustration of a database record from “Users” table --	38
Figure 54 => Illustration of a database record from “Messages” table -----	39
Figure 55 => Illustration of a database record from “Chats” table --	39
Figure 56 => Illustration of Cloudinary Cluster details -----	39
Figure 57 => Media Explorer of Cloudinary Cluster where images shared via web application are stored -----	40
Figure 58 => Implementation of Bcrypt hashing algorithm using “bcryptjs” in order to encrypt the password -----	40
Figure 59 => Generation of user specific unique tokens by Next-Auth when logged into the web application -----	41
Figure 60 => Generation of CSRF token at the time of Login in order to prevent CSRF Attacks -----	41

Figure 61 => VV CHAT Web Application responsive UI when viewed from iPad Mini layout -----	42
Figure 62 => Deployment of VV CHAT Web Application using Vercel -- -----	43
Figure 63 => Environment Variables setup during deployment of the web application -----	44
Figure 64 => VV CHAT Web App successful deployment into Production using Vercel -----	44
Figure 65 => Diagram depicting Error Handling techniques used in the development of web application -----	45
Figure 66 => Diagram depicting Console Logging, whenever interrupts are seen in the web application -----	46
Figure 67 => Dev Logs consisting of response previews of several backend API calls-----	47
Figure 68 => Collection of Results of End-to-End Testing -----	48
Figure 69 => Diagram of ESLint Audits -----	49

## 1. Introduction

### 1.1. Software Requirements Specification

The web application “VV CHAT” is intended to provide users “Messaging Service”. It initially needs to allow users to register for availing the services. Intuitively, the web app needs to authenticate the users and prompt them to first register and then login. The core functionality of the prototype is to allow sending and receiving of text messages between the users. Next requirement is providing Group Chat functionality. Throughout the service, it must give message updates in real time. The user interface should be clean and intuitive. The data behind the interface needs to be implemented using best practices. In addition to that, data needs to be restored back whenever user requests. Efficient and Optimized code implementation is appreciable. Additional requirements suggested are providing users with the feature of Audio & Video calling and including AI services (such as User-friendly chatbots) to enhance User Experience.

### 1.2. Dev Dependencies Documentation

This section explains about the technologies, dependencies and libraries used during the Design, Develop and Deployment of the application.

- 1) **Next.js** => To build responsive frontend UI components. With respect to easy deployment in the Vercel platform, I have chosen this tech stack.
- 2) **Nodejs** => To build back-end RESTful API services. Effectively utilized the concurrent requests handling feature of nodejs. Chosen this tech stack because of its high performance and lesser loading time when linked with nextjs components.

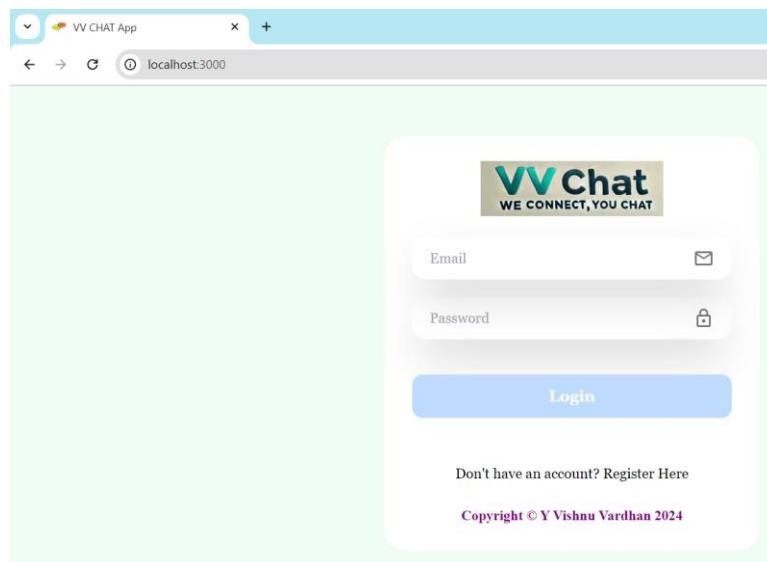
- 3) **NoSQL** => Not only SQL database picked because of its dynamic schemas ability to include unstructured and semi-structured datatypes.
- 4) **MongoDB** => Open source, non-relational database management system with easy horizontal scale-out with sharding.
- 5) **Mongoose** => MongoDB object modeling tool installed & imported to handle data validation, schema management. Used in establishing the link between the backend and database.
- 6) **BcryptJS** => Installed & Imported to make use of Bcrypt hashing algorithm for the encryption of passwords.
- 7) **Tailwind CSS** => open-source CSS framework that provides customizable styling. Installed in order to setup an organized code structure, segregating styling logic into one side and rest business logic on the other. By this act, later any UI/UX design modifications can be completed in an efficient manner.
- 8) **react-hook-form** => Installed & Imported to build responsive user login & register forms. This library enhanced user authentication and authorization functionalities.
- 9) **react-hot-toast** => Lightweight yet powerful library for creating visually appealing user alerts. Easy to use, well documented library.
- 10) **react-dom** => Package from React that provides DOM-specific methods, so that web application gets easy page navigations and user customization.
- 11) **pusher-js** => Made use of third-party software to avail real-time communication between the server and the client.

- 12) **next-cloudinary** => Installed & Imported high-performance image delivery and uploading library from Next.js which is powered by Cloudinary. Controls web application's image chatting and user profile picture upload functionalities.
- 13) **next-auth** => A complete open-source authentication solution for Next.js applications. Used this dependency to design OAuth authentication for availing token-based user login sessions.
- 14) **date-fns** => Date utility library used in the application to display the time of arrival of chat message.
- 15) **@emotion/react** => Package designed for writing responsive css styles in JavaScript.
- 16) **@emotion/styled** => Installed & Imported this package to allow end user to change the styles of a component based on the props. Used this in the web application Login button and Register button.
- 17) **@mui/icons-material** => React Material Icons installed to add icons of username, email and password on web app's Login and Register pages.
- 18) **ESLint** => plugin to ensure proper patterns and test all the configurations.

## 2. System Overview

### 2.1. System Description

The VV CHAT App (VV abbreviation stands for the developer's name – Vishnu Yardhan) is a messaging service prototype which is a real-time web application that allows users to send and receive messages. Apart from that feature, the web app also allows users to upload profile pictures. In the list of chats and contacts, provides ability to search for a required entity. Gives a responsive user interface by dynamically changing the UI components of the application. Users alerts and notifications displayed with low latency. Optimized user registration and authentication provided by the app. Mobile UI is handled efficiently, user can experience dynamic UI/UX when mobile layouts are changed. Clean & Optimized code handles retrieval of user profile details, storing them in backend database. Whenever user initiates a new chat, all the messages are stored in the backend for future retrievals. The VV Chat App apart from text messages, also allows users to share pictures through the chat. In future, enables users to upload video files too.



**Figure 01**

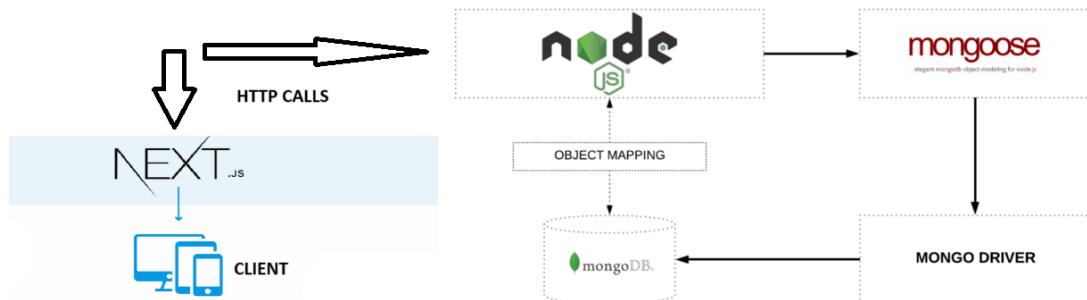
## 2.2. System Context

The Next.js web application – VV CHAT interacts with various services in order to deliver the system requirements efficiently. As Next.js UI components in terms of reliability go hand in hand with Node.js backend routes, web app developed REST APIs using nodejs programming language. VV CHAT preferred NoSQL database because, of its dynamic schemas ability. Messaging service quite often has usage of unstructured and semi-structured datatypes, which is achievable only through NoSQL. Among several NoSQL database providers, VV CHAT picked MongoDB Atlas cloud database service for storing the data. Since MongoDB is a flexible and customizable database that stores data in documents instead of traditional tables and rows, I believed CRUD operations of DBMS get executed in an optimized manner with very low latency. Specifically chosen MongoDB Atlas service, as it automates many administrative tasks which makes my job free and moreover gives me time to concentrate more on database schemas and relations. Furthermore, it also provides additional security and scalability features which will be helpful for my future implementations. VV CHAT implemented user authentication by importing NextAuth.js open-source authentication library. Any user after getting registered in our web app, will be authenticated successfully from the login page which takes forward to the Home page of VV CHAT. Strong authentications incorporated to stop users visiting inner layers of UI such as Contacts page, Profile page. Apart from that, the web application imported several other packages and dependencies to develop various functionalities such as Tailwind CSS, Pusher, Cloudinary so on. Reasons to pick those third-party tools are mentioned in Dev Dependencies Documentation.

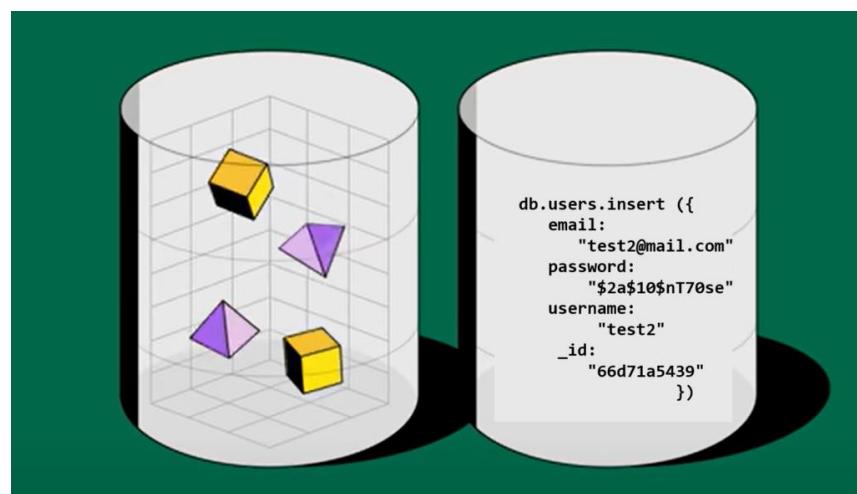
### 3. Architecture Design

#### 3.1. High-Level Architecture

VV CHAT Web application built frontend using Next.js. Since it is the React framework for the Web, has high scalability and efficient Route handling techniques. As it integrates with Node.js to give a strong backend, I used Node.js JavaScript runtime environment for executing the application on the server. The overall application performs various REST API calls in order to provide end-users single user chatting, group chatting, profile picture upload, start a new conversation and so on.



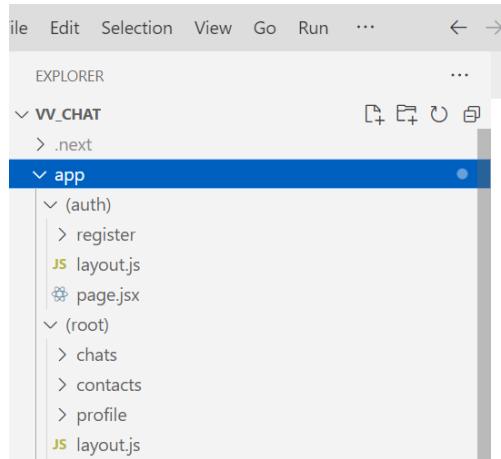
**Figure 02**



**Figure 03**

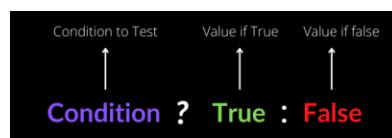
### 3.2. Component Diagram

First of all, the web application broadly differentiates Login & Register pages and rest of the pages (like Home page, Contact page, Chats page etc.) into 2 different components.



**Figure 04**

Dealing with initial Login and Register pages, considering code optimization I implemented both the pages using single component (Form.jsx). By importing “react-hook-form”, developing user validations task became easy. Applying ternary operator,



**Figure 05**

I have developed a responsive form when a user visits either Login page or Register page.

```

134      <button className="button" type="submit">
135        {type === "register" ? "Register" : "Login"}
136      </button>
137    </form>
138
139    {type === "register" ? (
140      <Link href="/" className="link">
141        <p className="text-center">Already registered? Log In</p>
142      </Link>
143    ) : (
144      <Link href="/register" className="link">
145        <p className="text-center">Don't have an account? Register Here</p>
146      </Link>
147    )}
  
```

**Figure 06**

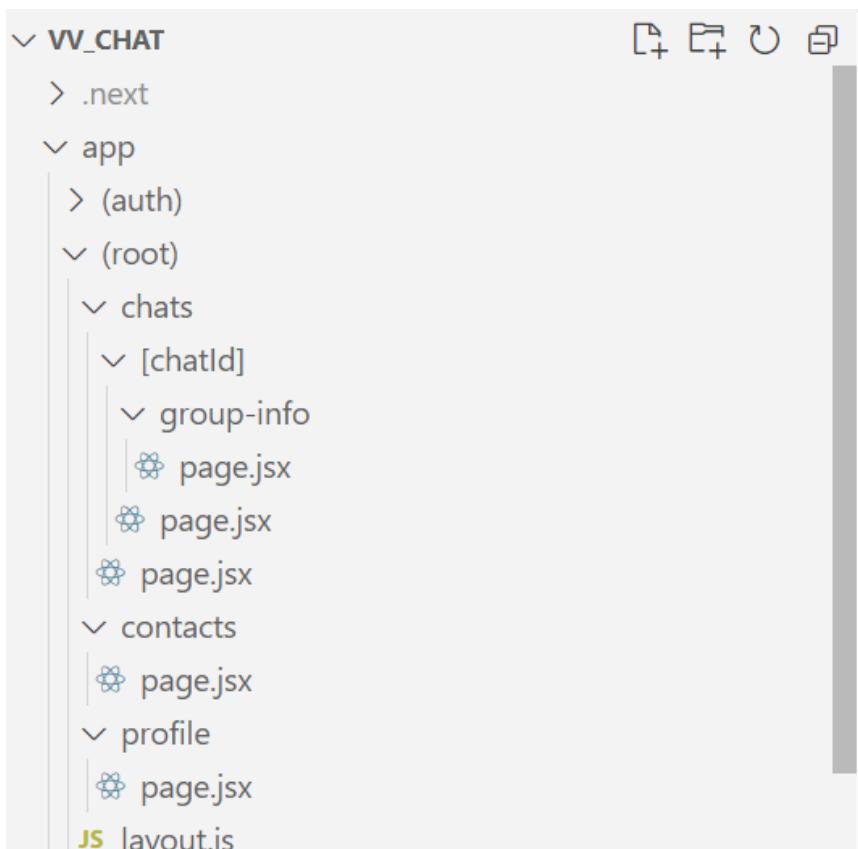
Coming to the Home page UI, using Nextjs File-based Routing feature, I divided the root directory into 3 different children:

chats

contacts

profile

Furthermore, chats directory is query-downed (chatId]) to get specific user chatting page.



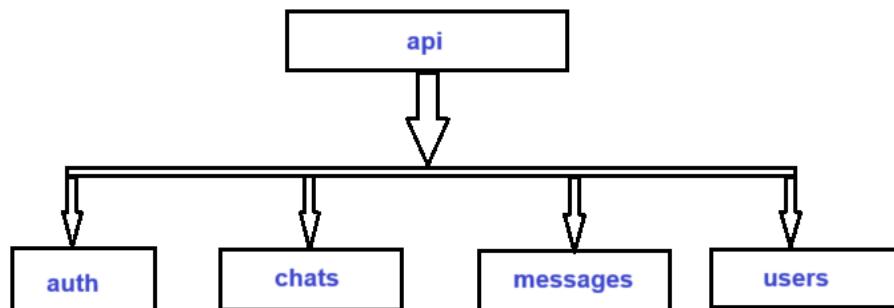
**Figure 07**

One more optimization technique adopted in styling the UI pages is, engulfing all the component's, section's and HTML tag's CSS logic into single file (globals.css).

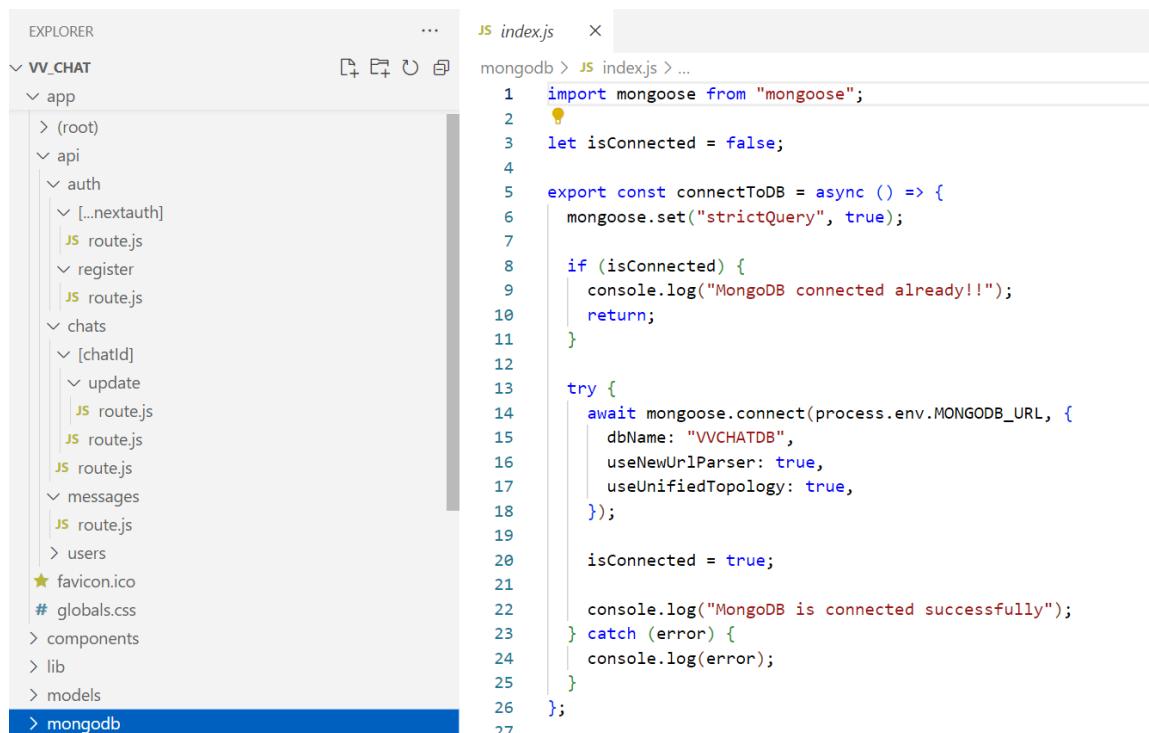
```
# globals.css 9+ X
app > # globals.css > ...
40  }
41
42 .link {
43 | @apply text-base-medium cursor-pointer hover:bg-red-50;
44 }
45
46 /* Main Container */
47 .main-container {
48 | @apply h-screen flex justify-between gap-5 px-10 py-3 max-lg:gap-8;
49 }
50
51 /* Top Bar */
52 .topbar {
53 | @apply top-0 sticky px-20 py-5 flex items-center justify-between bg-blue-2;
54 }
55
56 .menu {
57 | @apply flex items-center gap-8 max-sm:hidden;
58 }
59
60 .profilePhoto {
61 | @apply w-11 h-11 rounded-full object-cover object-center;
62 }
63
64 /* Bottom Bar */
65 .bottom-bar {
66 | @apply w-full flex justify-between items-center px-5 py-9 bg-blue-2 sm:hidden
67 }
```

**Figure 08**

Describing the REST APIs implementation, I broadly classified the main api directory into 4 different modules.

**Figure 09**

Each and every nodejs file present in the above modules will send GET/POST requests via establishing a connection to the database. This is achieved by installing and importing “mongoose”. Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js. It manages relationships between data, provides schema validation, and is used to translate between objects in code and the representation of those objects in MongoDB.



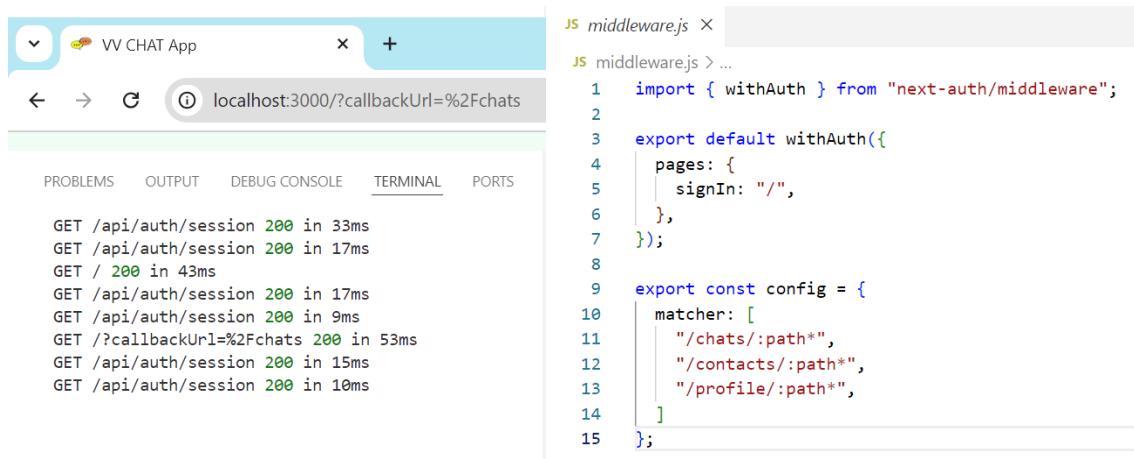
```

JS index.js
mongodb > JS index.js > ...
1 import mongoose from "mongoose";
2
3 let isConnected = false;
4
5 export const connectToDB = async () => {
6   mongoose.set("strictQuery", true);
7
8   if (isConnected) {
9     console.log("MongoDB connected already!!!");
10    return;
11  }
12
13 try {
14   await mongoose.connect(process.env.MONGODB_URL, {
15     dbName: "VVCHATDB",
16     useNewUrlParser: true,
17     useUnifiedTopology: true,
18   });
19
20   isConnected = true;
21
22   console.log("MongoDB is connected successfully");
23 } catch (error) {
24   console.log(error);
25 }
26};
27

```

**Figure 10**

Coming to protection of client and server side rendered pages as well as API routes, I implemented next-auth middleware pattern. By this, my web app allows to run a code before a request is completed. Through this functionality, I secured chats page, contacts page and profile page from intruders. As soon as these pages are called without prior login, the user redirects to the login page via callbacks.



The screenshot shows a browser window titled "VV CHAT App" with the URL "localhost:3000/?callbackUrl=%2Fchats". The terminal tab displays API logs:

```
GET /api/auth/session 200 in 33ms
GET /api/auth/session 200 in 17ms
GET / 200 in 43ms
GET /api/auth/session 200 in 17ms
GET /api/auth/session 200 in 9ms
GET /?callbackUrl=%2Fchats 200 in 53ms
GET /api/auth/session 200 in 15ms
GET /api/auth/session 200 in 10ms
```

The code editor shows the file "middleware.js" with the following content:

```
JS middleware.js ×
JS middleware.js > ...
1 import { withAuth } from "next-auth/middleware";
2
3 export default withAuth({
4   pages: {
5     signIn: "/",
6   },
7 });
8
9 export const config = {
10   matcher: [
11     "/chats/:path*",
12     "/contacts/:path*",
13     "/profile/:path*",
14   ],
15 });
16
```

**Figure 11**

### 3.3. Technologies Used

The technology stack used to Design, Develop & Deploy the VV CHAT Web Application is:

**Nextjs** = For developing Front end components.

**React Hook Form** = For developing super lightweight and responsive user forms.

**Nodejs** = For building backend framework and REST API calls.

**MongoDB** = For storing data and to exhibit persistent data.

**Tailwind CSS** = For clean and consistent UI by availing utility-first styling principles.

**Cloudinary** = To manage user profile picture upload and photo messages functionality.

**Vercel** = Cloud infrastructure to deploy the web application.

In addition to those, for development of other functionalities, I installed multiple open-source tools and modules using Node Package Manager (npm).

Below figure depicts overall packages along with their version numbers used in the development.

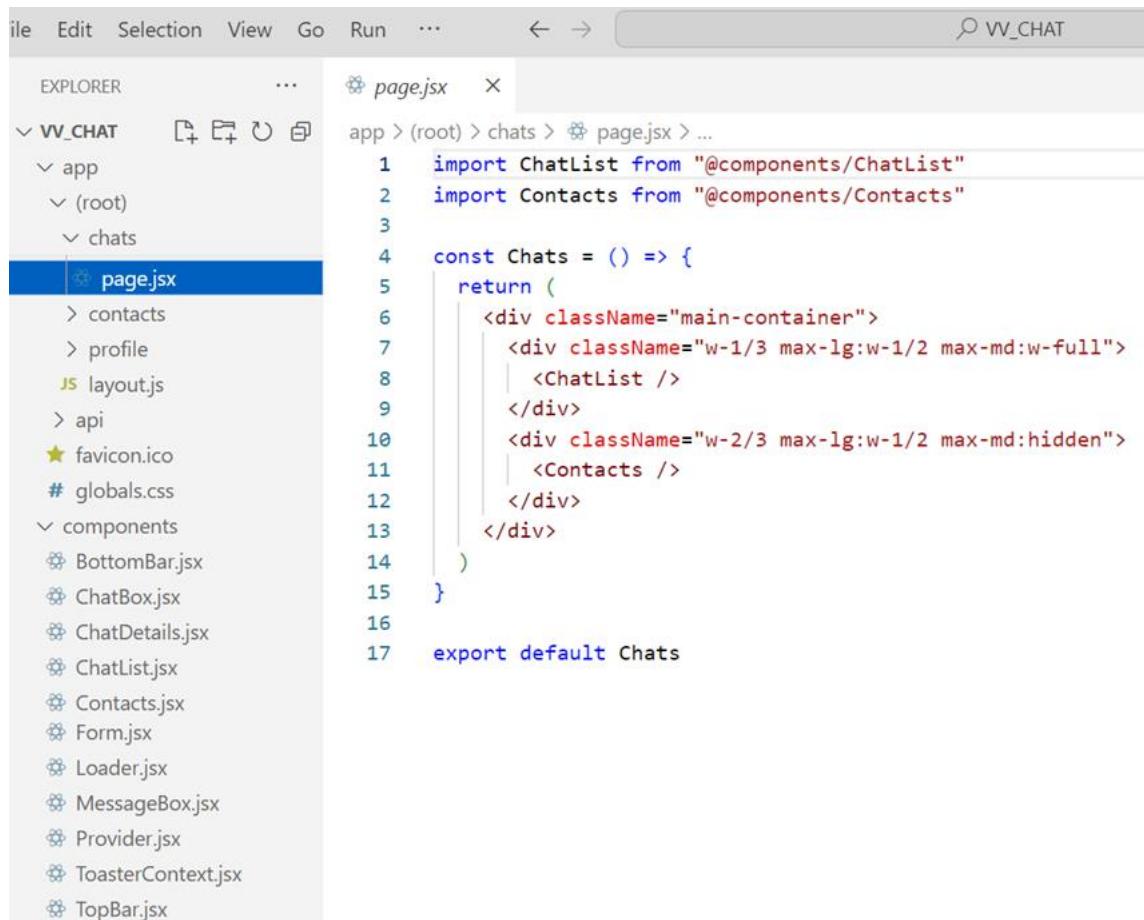
```
"dependencies": {  
    "@emotion/react": "^11.13.3",  
    "@emotion/styled": "^11.13.0",  
    "@mui/icons-material": "^6.0.1",  
    "bcryptjs": "^2.4.3",  
    "date-fns": "^3.6.0",  
    "mongoose": "^8.6.0",  
    "next": "14.2.7",  
    "next-auth": "^4.24.7",  
    "next-cloudinary": "^6.11.0",  
    "pusher": "^5.2.0",  
    "pusher-js": "^8.4.0-rc2",  
    "react": "^18",  
    "react-dom": "^18",  
    "react-hook-form": "^7.53.0",  
    "react-hot-toast": "^2.4.1"  
},  
"devDependencies": {  
    "postcss": "^8",  
    "tailwindcss": "^3.4.1"  
}
```

Figure 12

## 4. Detailed Design

### 4.1. Frontend Design

VV CHAT Web Application frontend consists of multiple UI components. Considering the Atomic Design practices, I created a separate directory containing all the components and imported them into the root folder whenever needed.



The screenshot shows a code editor interface with a navigation bar at the top. The left sidebar is labeled 'EXPLORER' and shows a tree view of the project structure under 'VV\_CHAT'. The 'chats' folder contains 'page.jsx', which is currently selected and highlighted with a blue background. The main editor area displays the code for 'page.jsx'.

```

    File Edit Selection View Go Run ...
    ← → 🔍 VV_CHAT

    EXPLORER ...
    VV_CHAT
      app
      (root)
        chats
          page.jsx
            contacts
            profile
            layout.js
            api
            favicon.ico
            # globals.css
          components
            BottomBar.jsx
            ChatBox.jsx
            ChatDetails.jsx
            ChatList.jsx
            Contacts.jsx
            Form.jsx
            Loader.jsx
            MessageBox.jsx
            Provider.jsx
            ToasterContext.jsx
            TopBar.jsx

    page.jsx
    app > (root) > chats > page.jsx > ...
    1 import ChatList from "@components/ChatList"
    2 import Contacts from "@components/Contacts"
    3
    4 const Chats = () => {
    5   return (
    6     <div className="main-container">
    7       <div className="w-1/3 max-lg:w-1/2 max-md:w-full">
    8         <ChatList />
    9       </div>
   10      <div className="w-2/3 max-lg:w-1/2 max-md:hidden">
   11        <Contacts />
   12      </div>
   13    </div>
   14  )
   15 }
   16
   17 export default Chats
  
```

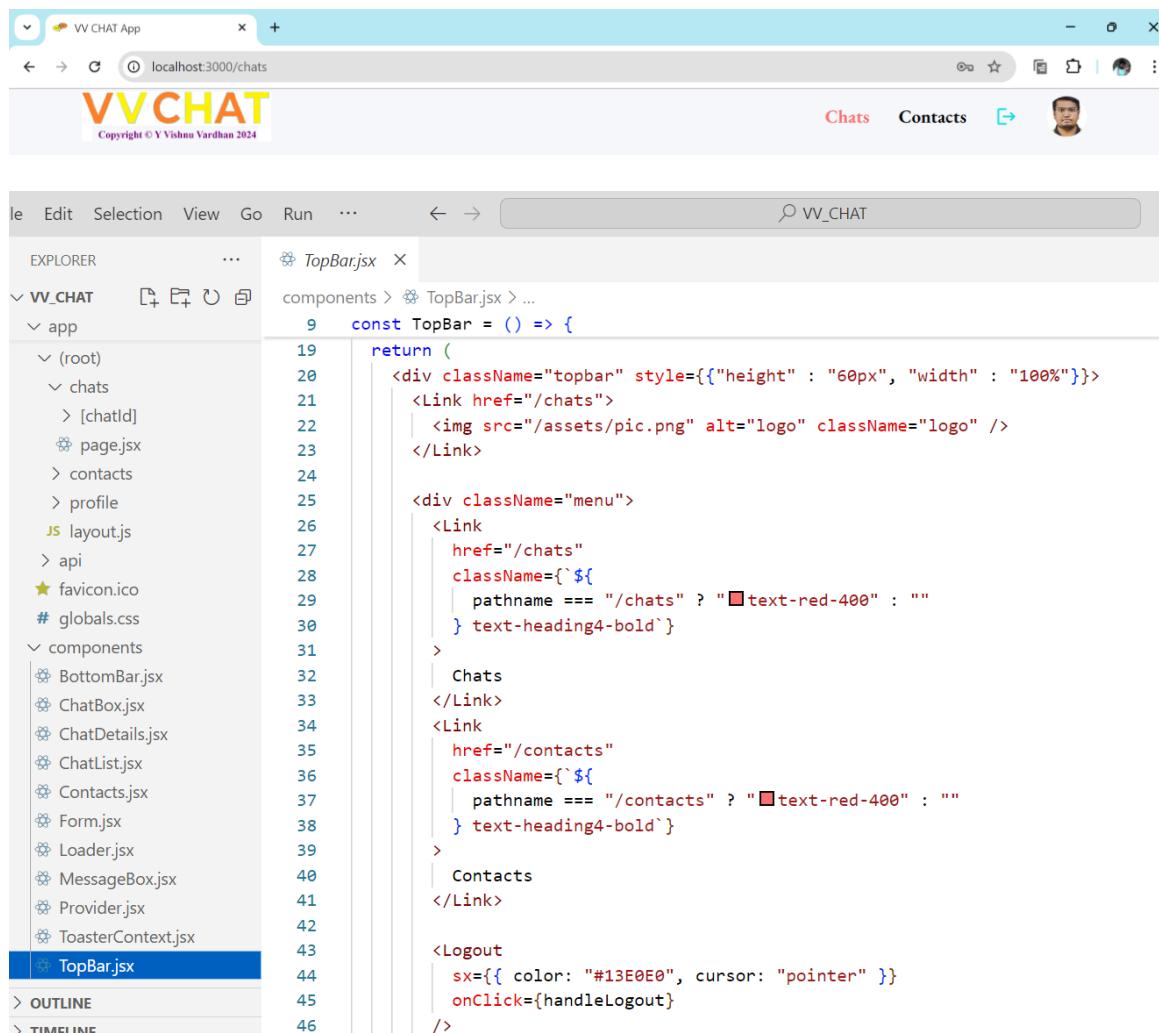
**Figure 13**

In figure 13, you can see chats component present in the root folder utilized ChatList and Contacts components by importing them into the page.jsx of chats component. Changing the logic in ChatList.jsx and Contacts.jsx components will affect all the pages who imported these.

Each and every jsx extension file inside Components directory will be rendering from the root folder. Here's the description of each file:

## TopBar.jsx

This component displays the top section of the Home page. Includes addition of web application's logo, hyperlinks which navigate users to Chat section, Contacts section. Logout button and User Icon to update profile name and upload new profile picture.



**Figure 14**

### BottomBar.jsx

This component is a responsive component which upon changes in the screen dimensions will appear at the bottom of the Home page. Includes hyperlinks which navigate users to Chat section, Contacts section. Logout button and User Icon to update profile name and upload new profile picture.

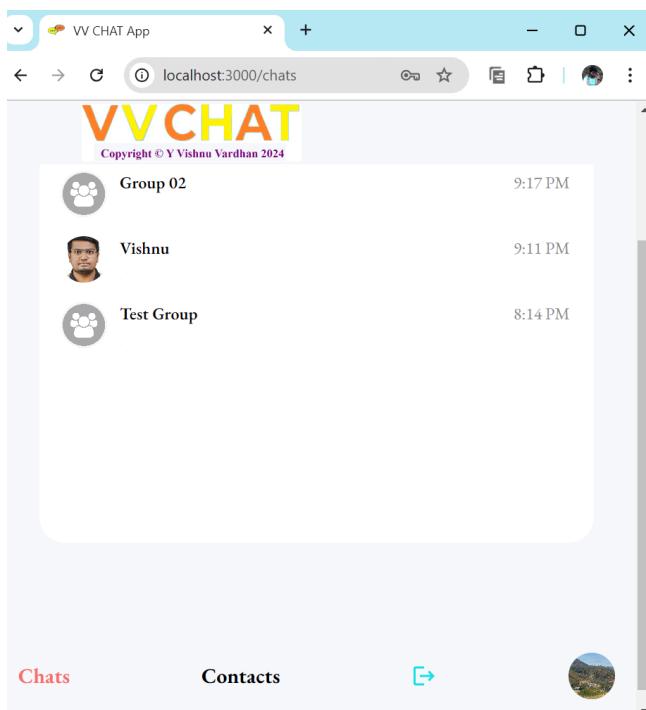
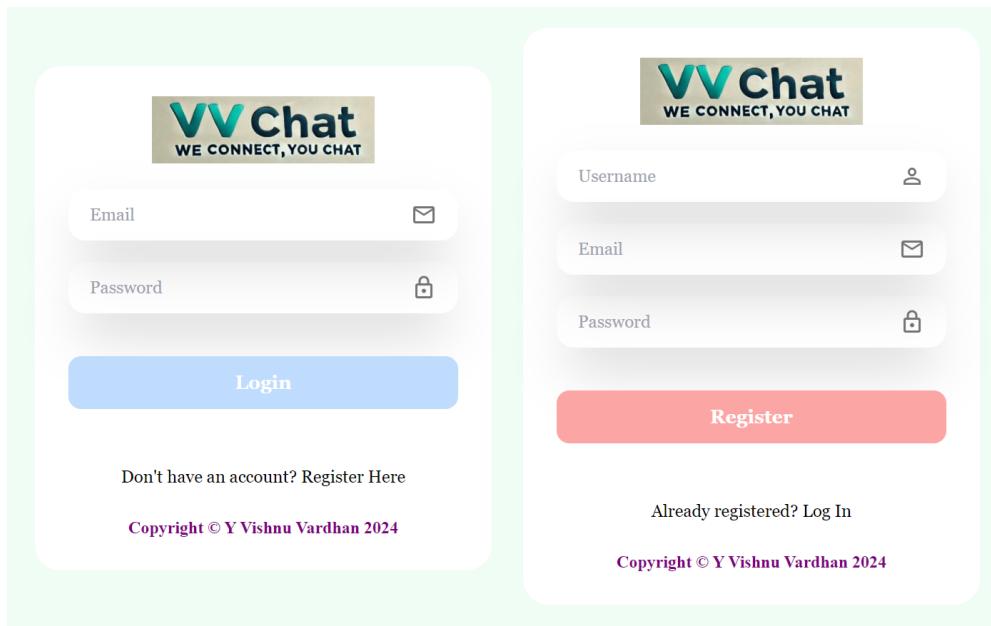


Figure 15

### Form.jsx

This component makes use of “react-hook-form” library for generating clean and clear forms. I imported “@mui/icons-material” Material UI which is an open-source React component library for depicting icons for Username, Email and Password. Responsive logic written which displays content based on whether it’s Login page or it’s Register page.



**Figure 16**

### ToasterContext.jsx

This component is responsible for generating notification pop-ups. Need to import “Toaster” from "react-hot-toast" library.



**Figure 17**

## Provider.jsx

This component uses “useSession” React Hook from "next-auth/react" library which is responsible in checking whether user signed in or not. All root’s children are wrapped inside it.

```
Provider.jsx X
components > Provider.jsx > default
1  "use client"
2
3  import { SessionProvider } from "next-auth/react"
4
5  const Provider = ({ children, session }) => {
6    return (
7      <SessionProvider session={session}>
8        {children}
9      </SessionProvider>
10    )
11  }
12
13 export default Provider
```

Figure 18

## Loader.jsx

This component displays blue color loading icon whenever requested page takes time to render onto the screen.

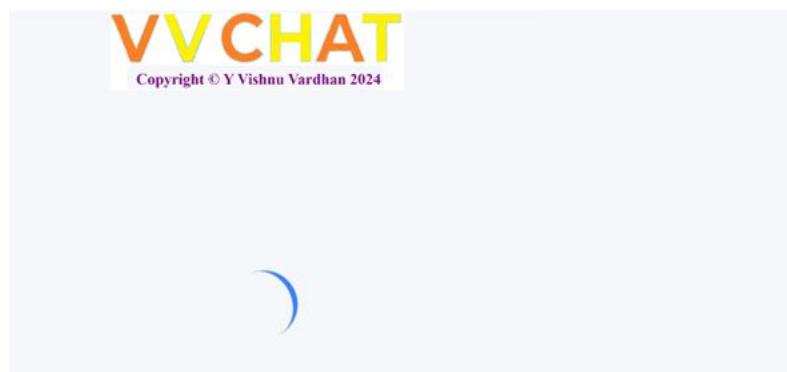
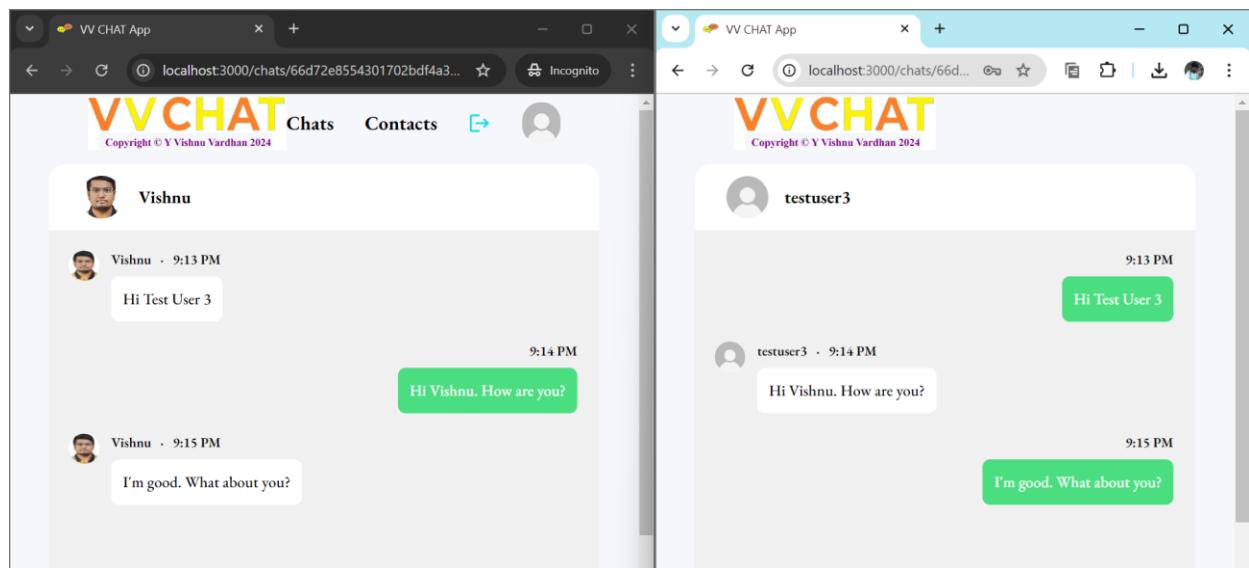


Figure 19

## MessageBox.jsx

This is one of the main component which takes care of the chatting functionality. Based on the sender of the message, this component alters the message box color. This feature implicating the source of the message gives a better user experience.



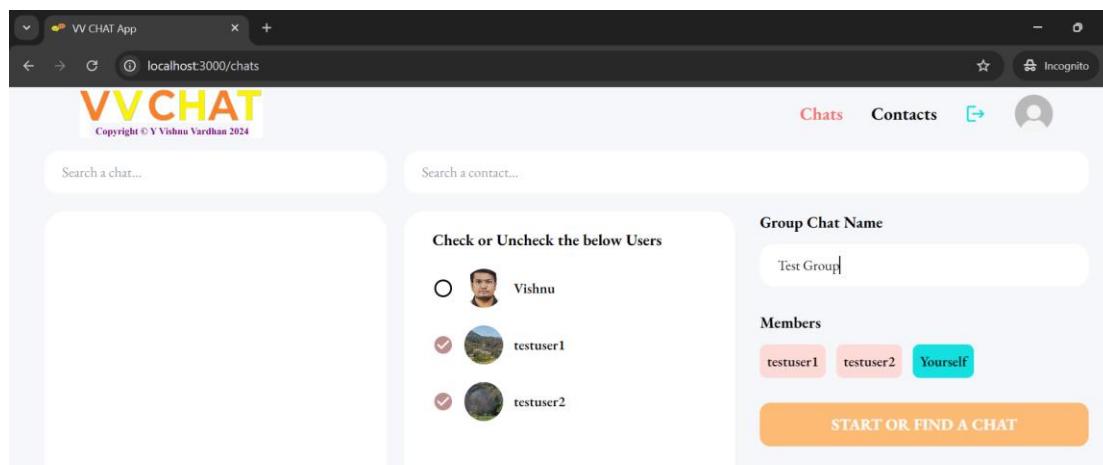
**Figure 20**

## Contacts.jsx

Another important component which controls major functionalities like Checking or Unchecking the Users in order to start a new message conversation thread. Responsive Group Chat logic has been implemented in this component.

When more than one user is selected, then UI displays Group Chat dialog which gives user option to add group name and displays members of the group.

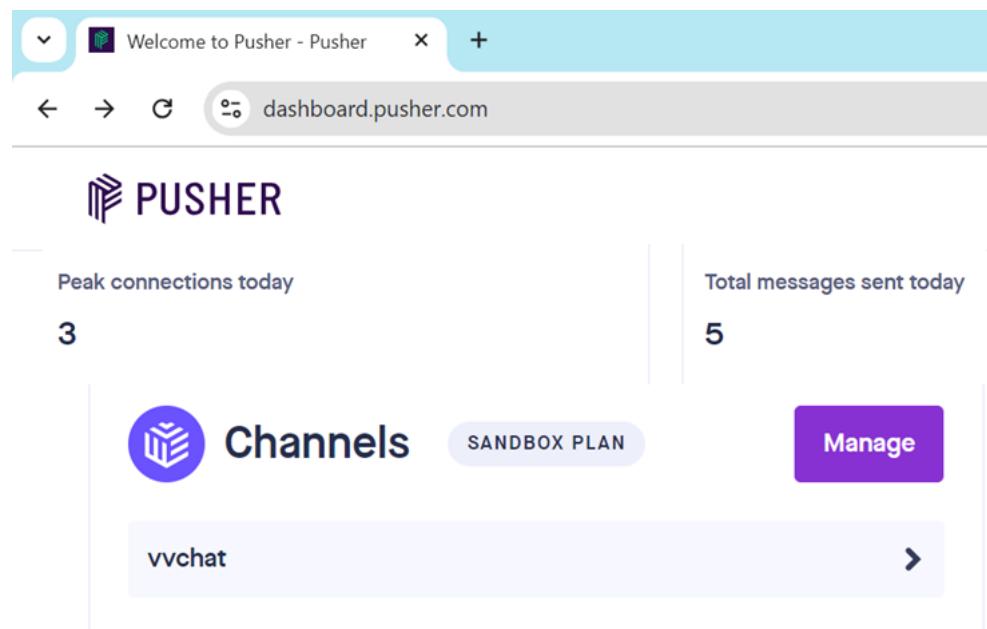
Additionally, this component is responsible for searching a specific contact.

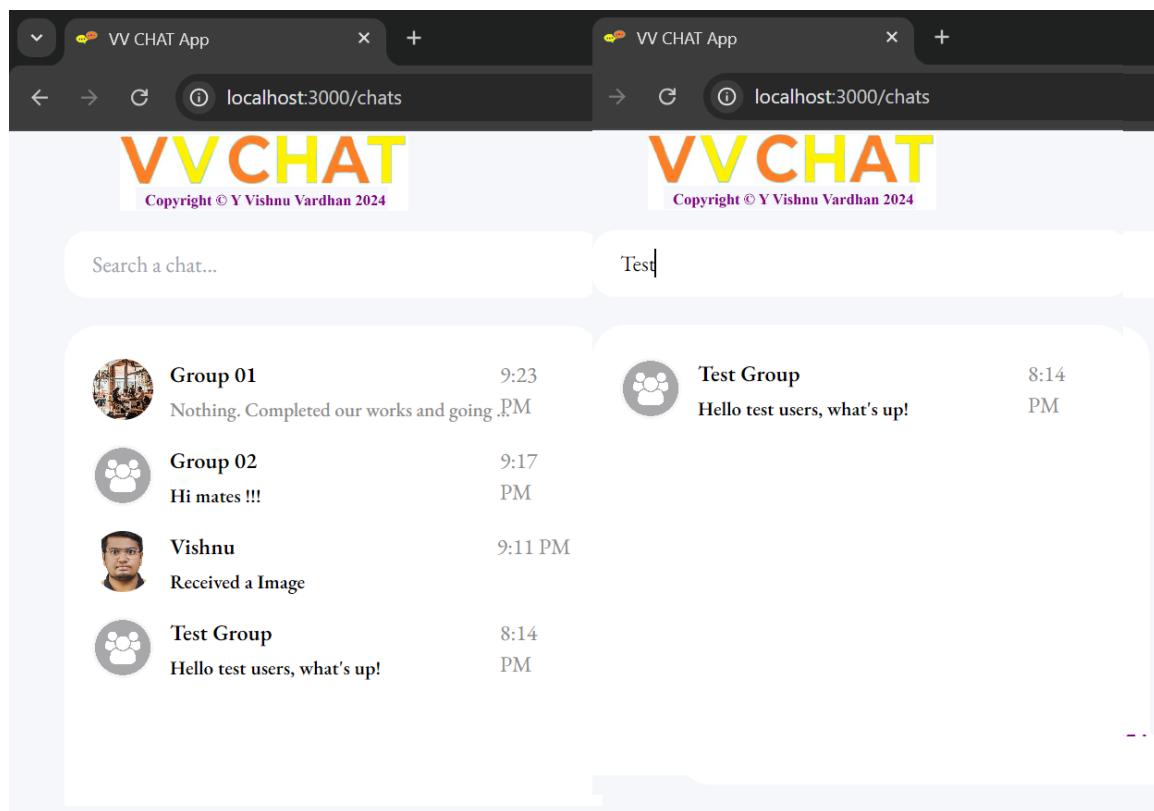
**Figure 21**

### ChatList.jsx

This component gives users the chance to search for a particular chat from the Chats list.

In addition to that using Pusher API Service which uses Web sockets, user will experience real time updates regarding new chat creation, new message conversation and chat updates.

**Figure 22**

**Figure 23**

### ChatDetails.jsx

This component catch holds vital feature of the web application i.e the chat dialog to type the message and send to the receiver. Apart from that, users can upload an image as a message. This functionality is achieved by making use of the Next Cloudinary Library. This provides a secure and comprehensive API for easy upload of media files. Currently, application allows only image files upload with limit of one at a time.

Moreover, user experiences scrolling down of the screen to the bottom whenever new message arrived.

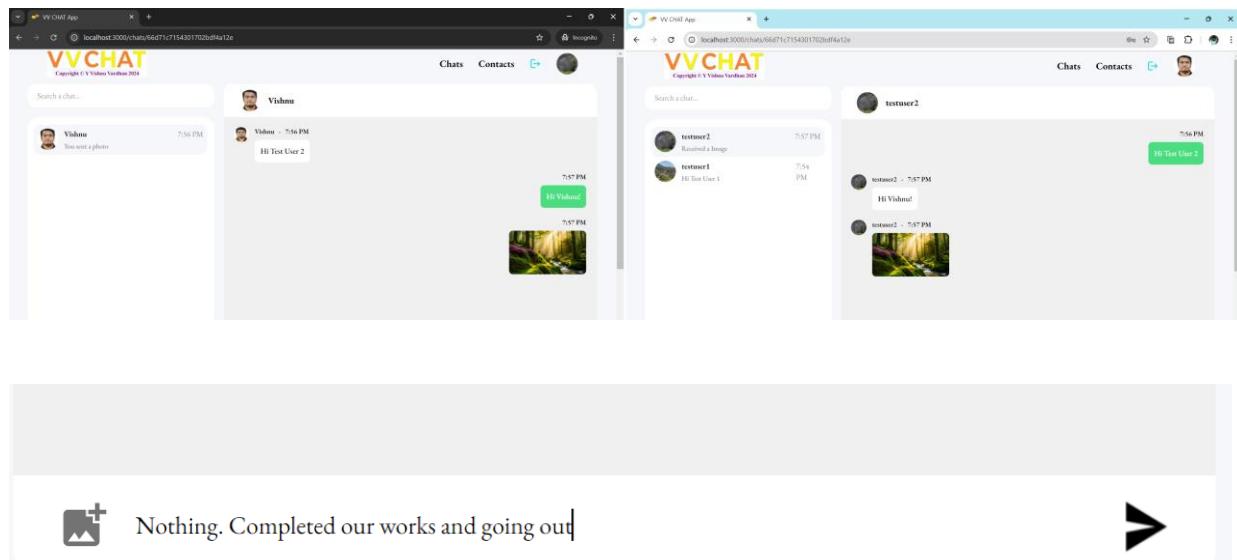


Figure 24

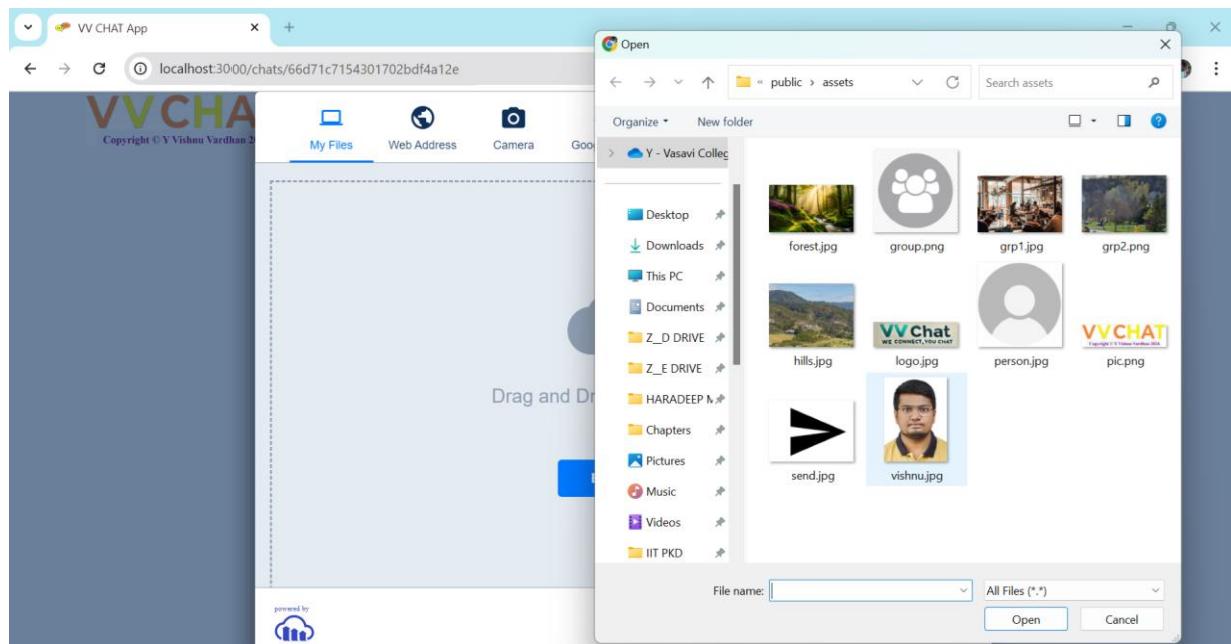
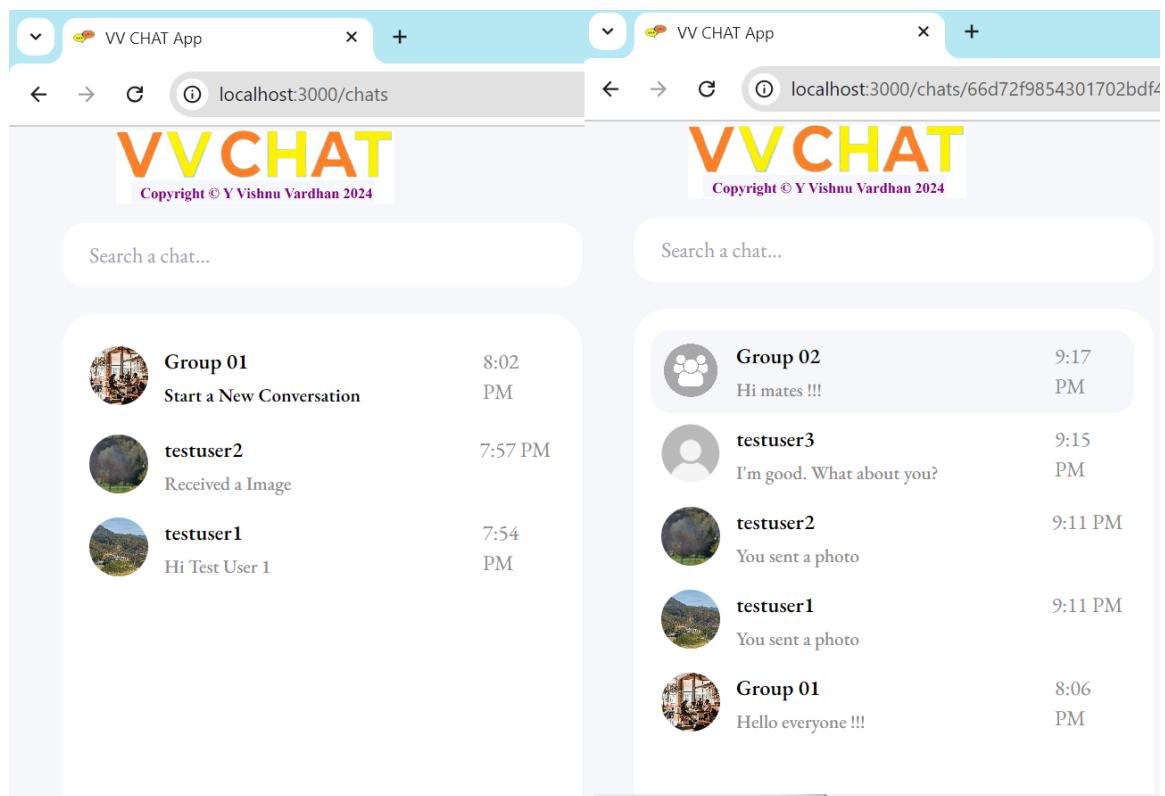


Figure 25

### ChatBox.jsx

This component stacks up the chat conversations which were initiated by the user. Based on whether it's a group chat or single user chat, by default profile photo gets displayed. Users also gets bold text notifications whenever new message arrived. Using “data-fns” JavaScript date utility library, user get to know the time at which message arrived.

For a new chat, UI persuades by displaying “Start a New Conversation” message.



**Figure 26**

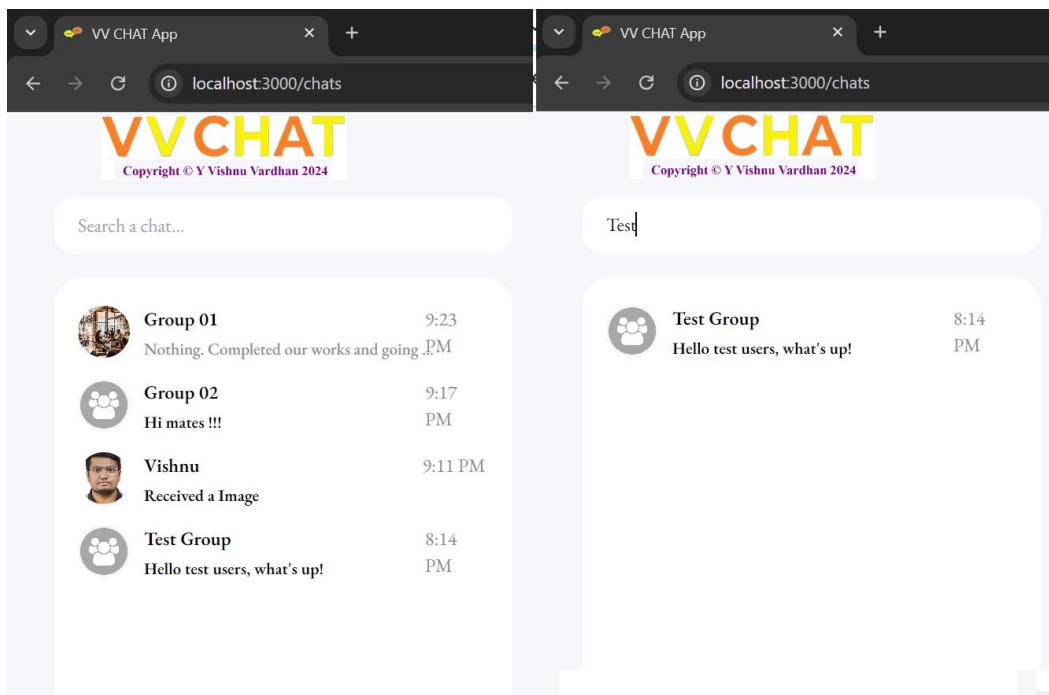


Figure 27

## 4.2. Backend Design

Nodejs framework due to its event-driven, non-blocking nature is being used to build various APIs. Broadly calls are segregated into 4 modules namely auth, chats, messages and users. I made use of RESTful APIs throughout the web application by calling standard HTTP GET/POST methods to perform the CRUD operations. Data transferred or retrieved in the form of JSON object.

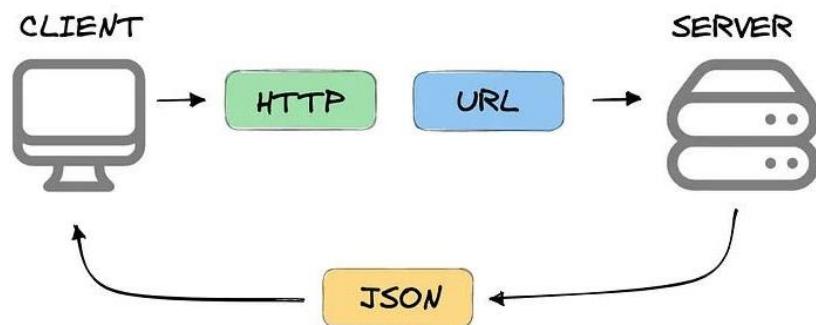
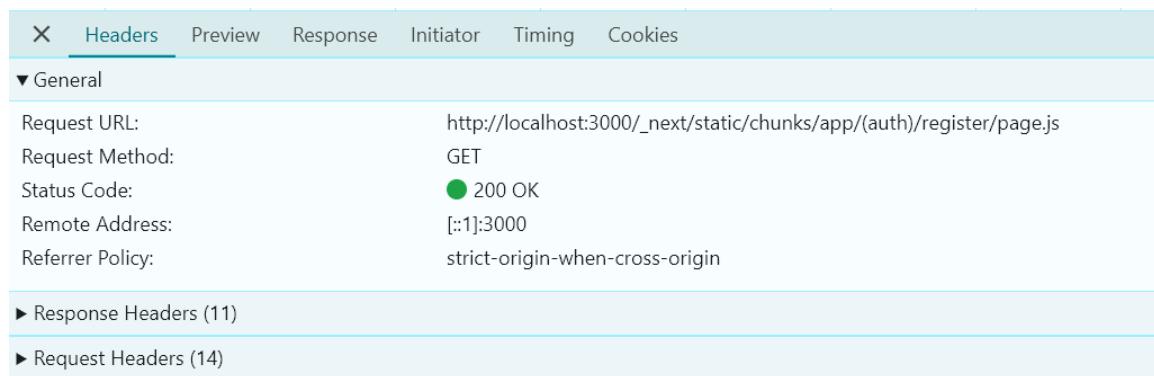


Figure 28

### List of Backend API Calls

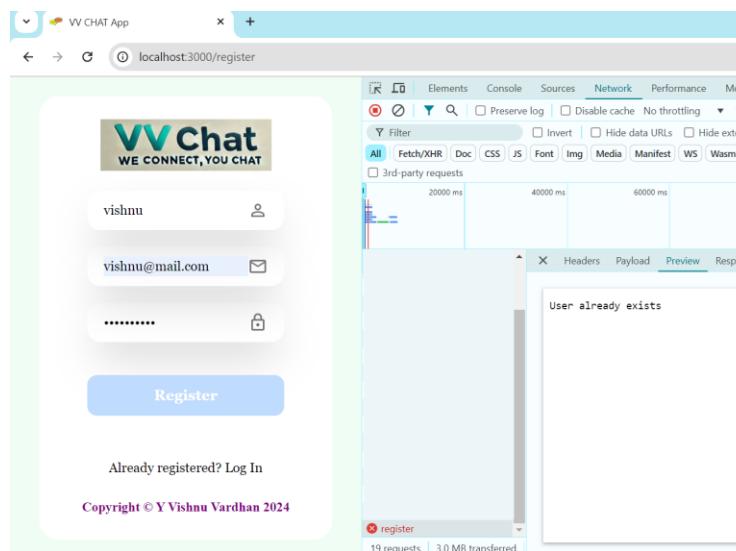
#### **api/auth/register/route.js**

I utilized “hash” function imported from “bcryptjs” library, to encrypt user entered password in order to store it in MongoDB. As Bcrypt hashing algorithm is implemented, users can experience a highly secure and protected web application prototype.



**Figure 29**

This Javascript file also sends request to the database to check whether user is already registered, if yes prompts the user about the existence of the account.



**Figure 30**

**api/auth/[...nextauth]/route.js**

Following NextAuth.js format, added the backend logic for Login page in this file path.

Using “Toaster” from "react-hot-toast" library, added user alerts whenever invalid email or password is entered.

The screenshot shows a browser window with the title 'VV CHAT App' and the URL 'localhost:3000'. The main content is a login form with fields for 'test1@mail.com' and a masked password. A red error message 'Invalid email or password' is displayed above the form. Below the form is a 'Login' button and a 'Don't have an account? Register Here' link. At the bottom, there is a copyright notice: 'Copyright © Y Vishnu Vardhan 2024'.

The right side of the screenshot shows the Network tab of the developer tools. It lists a single request to 'http://localhost:3000/api/auth/callback/credentials'. The request details show a status code of 401 Unauthorized. The response headers include 'Content-Type: application/json', 'Date: Wed, 04 Sep 2024 15:46:46 GMT', and 'Vary: RSC, Next-Router-State-Tree, Next-Router-Prefetch'.

**Figure 31**

Through prior database connection, this file calls GET request to retrieve the registered user's credentials and compares with the entered ones. If mismatch occur, instantly the UI alerts the user.

The screenshot shows two tabs of the developer tools: 'Form Data' and 'Request Cookies'.

**Form Data:** Shows the following data sent in the POST request:

- email: test1@mail.com
- password: test2@123
- redirect: false
- csrfToken: b8d5b9589f73e8b824f1df56e1b2ceb6c0800f3255990eda6775479ff28ffc4d
- callbackUrl: http://localhost:3000/
- json: true

**Request Cookies:** Shows the following cookies present in the request:

Name	Value	Dom...	Path	Expir...	Size	Http...
_clerk_db_jwt	dvb_2INVY2LQXsMvxzAazPnmnaqKh5	localh...	/	2025...	45	
_clerk_db_jwt_-SUKGP3S	dvb_2INVY2LQXsMndzAazPnmnaqKh5	localh...	/	2025...	54	
_client_uat	0	localh...	/	2025...	13	
_client_uat_-SUKGP3S	0	localh...	/	2025...	22	
_xrf	2 c2ba84390e7b825a00913a1a0c395af7...	localh...	/	2024...	59	
next-auth.callback-url	http%3A%2F%2Flocalhost%3A3000%2F	localh...	/	Session	54	✓
next-auth.csrf-token	b8d5b9589f73e8b824f1df56e1b2ceb6c08...	localh...	/	Session	151	✓

**Figure 32**

## api/chats/route.js

This JavaScript file is responsible for starting a new chat or opening an existing chat.

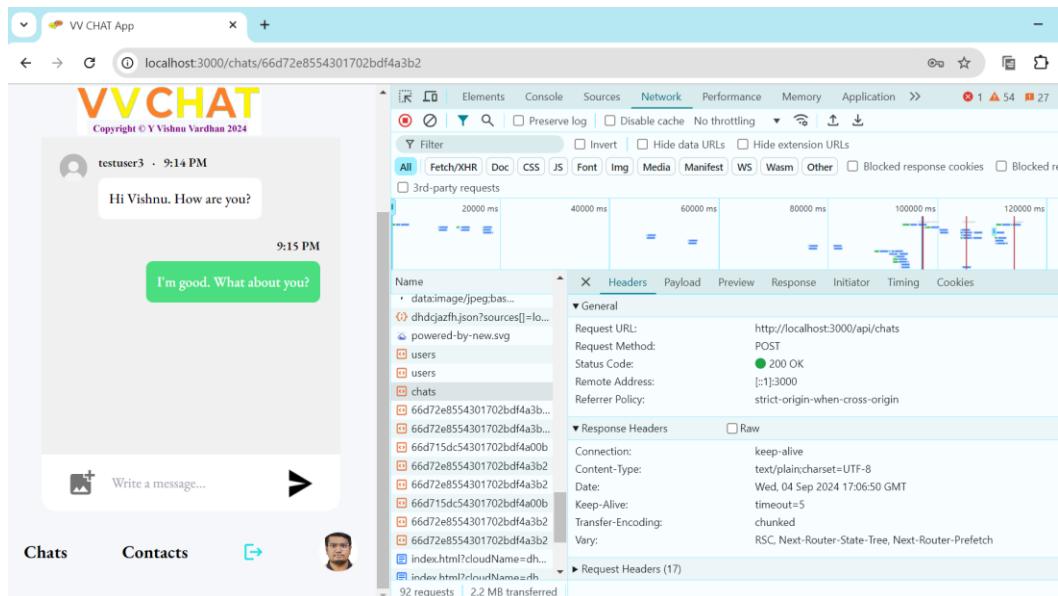


Figure 33

POST request is sent whenever, user tries to open a chat by selecting usernames from the Contacts page. In the above figure, since testuser3 chat is selected, in the response of the POST request, we will be able to see the “chatId” of testuser3 as member and all the messages already sent as “messageIds”.

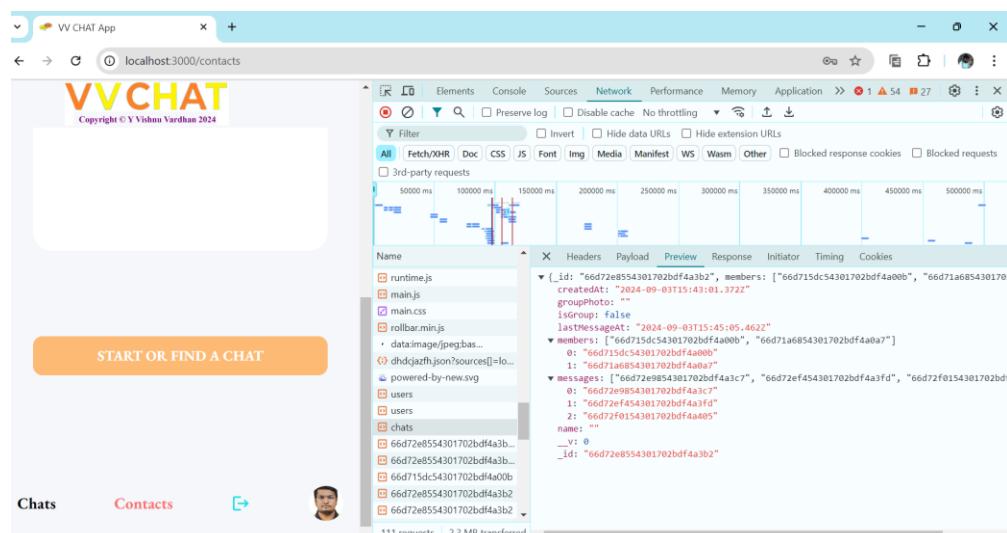


Figure 34

## api/chats/[chatId]/route.js

This JavaScript file handles GET and POST requests of chatting messages. Makes use of all the tables present in the database and by mongoose driven connectToDB function, it will ensure to perform addition and modification of database tuples. To display the past chatted messages, this file calls GET request which has unique [chatId] in the endpoint.

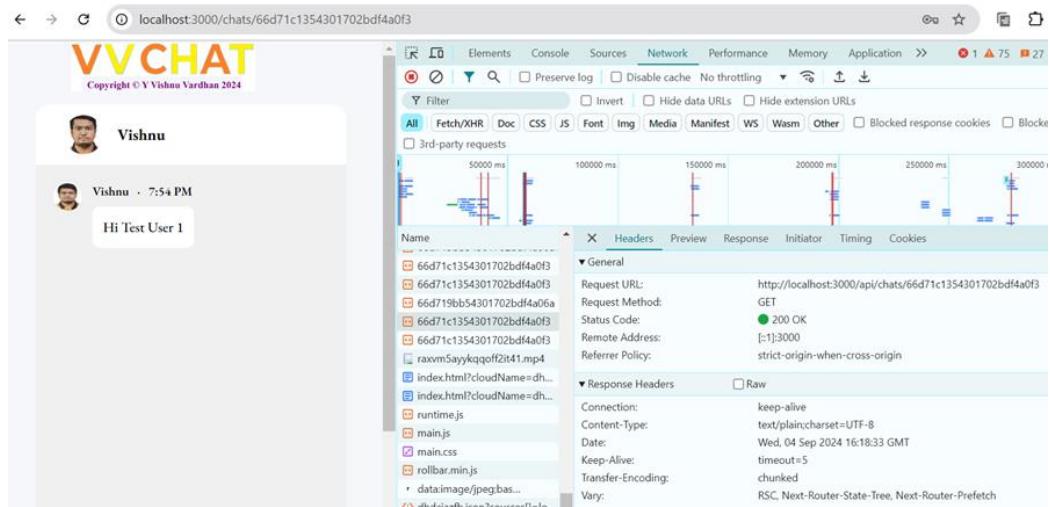


Figure 35

Upon successful 200 OK Response, whenever user opens a previous chat box, as a JSON object this request will fetch all the details of both sender & receiver.

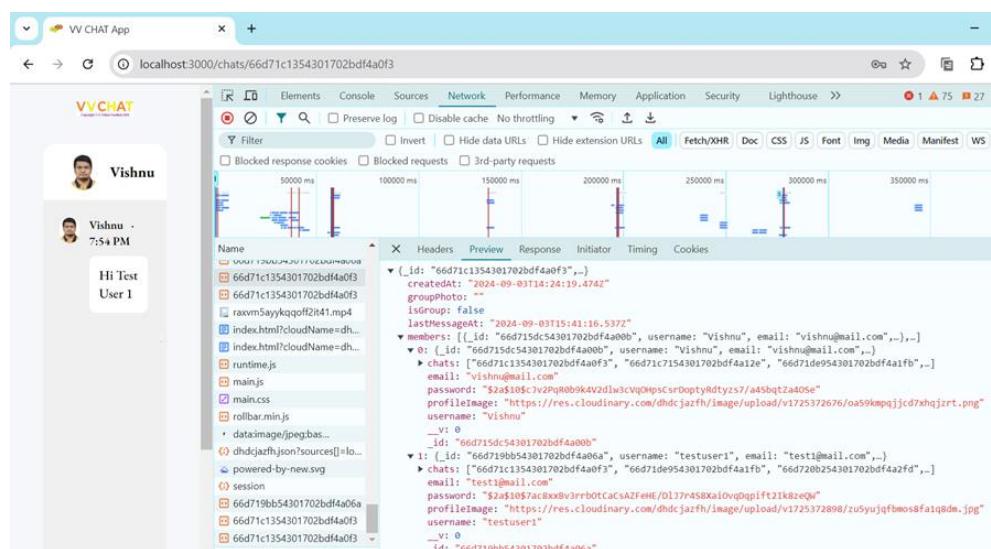


Figure 36

When a new message is sent to the receiver, then POST request is called which will add a new message record into the database. Every individual message gets a unique messageId which is later used by GET request in retrieving the past messages information.

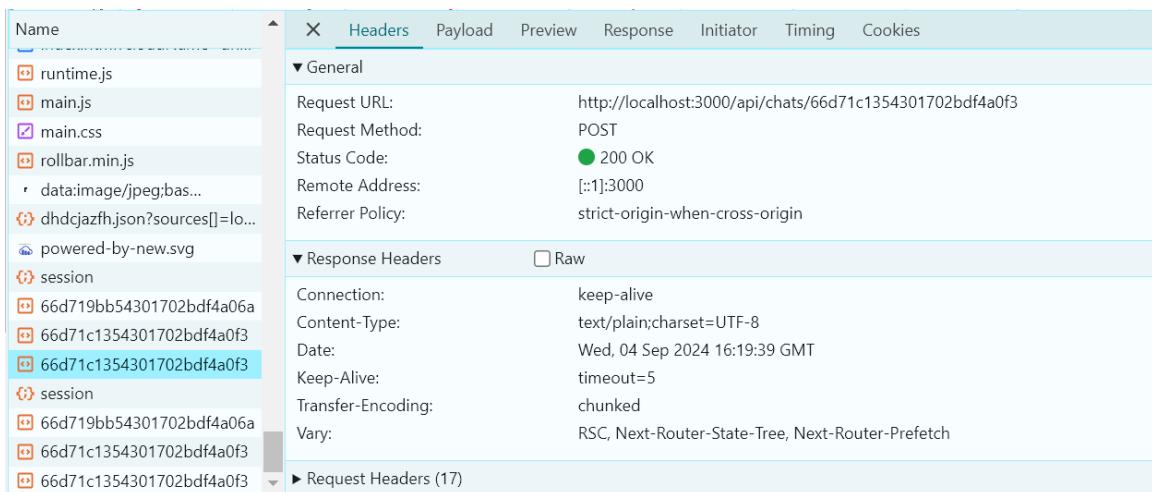


Figure 37

Upon successful delivery of the message from the receiver, whenever the user views the sent message, this file notifies that all the messages were seen by the current user.

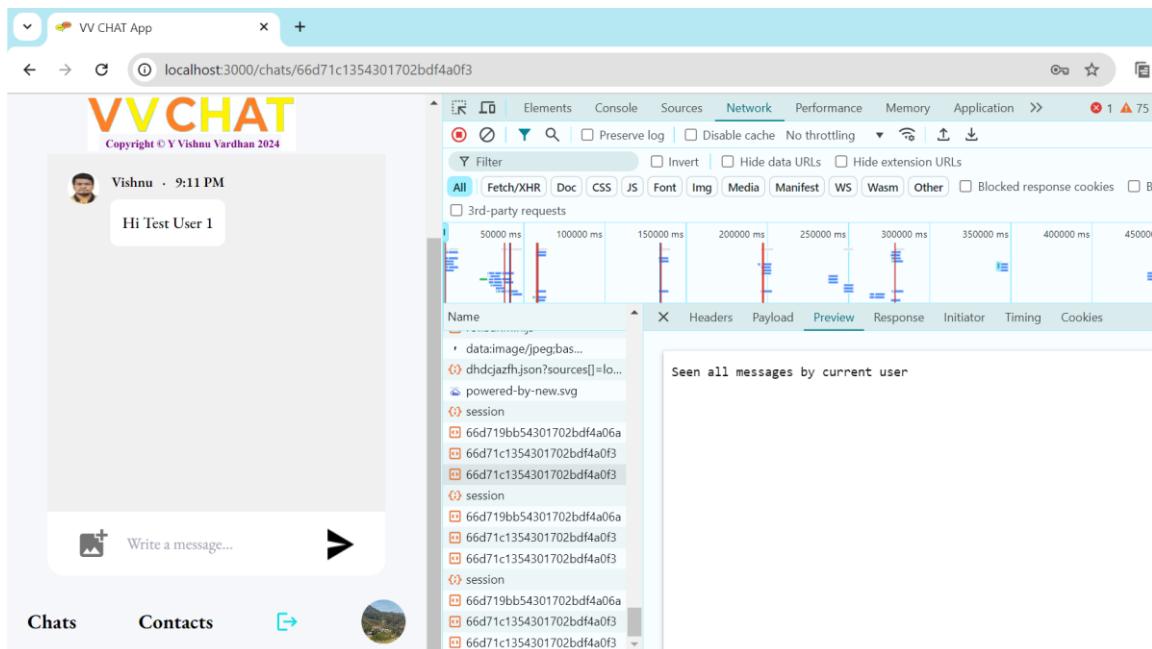


Figure 38

### api/chats/[chatId]/update/route.js

This file initiates POST request which takes Group chat new name and new profile picture as body. Upon successful response generation it will update the UI accordingly. If error is generated, it catches and logs 500 status Response with error message as “Failed to update group chat info”.

The screenshot shows a browser window for the 'VV CHAT App'. The main content is the 'Edit Group Info' page, which has a group name 'Group Name 02' and a profile picture placeholder. Below the profile picture is an 'Upload' button. A list of users ('Vishnu', 'testuser1', 'testuser2') is shown. At the bottom is an orange 'Save Changes' button. To the right of the browser window is the Chrome DevTools Network tab. It lists several requests, with one highlighted as a POST request to 'http://localhost:3000/api/chats/66d72f9854301702bdff4a449/update'. The 'Headers' tab shows the request headers, and the 'Response' tab shows the JSON response:

```

Request URL: http://localhost:3000/api/chats/66d72f9854301702bdff4a449/update
Request Method: POST
Status Code: 200 OK
Remote Address: [::]:3000
Referrer Policy: strict-origin-when-cross-origin
Connection: keep-alive
Content-Type: text/plain; charset=UTF-8
Date: Wed, 04 Sep 2024 16:14:54 GMT
Keep-Alive: timeout=5
Transfer-Encoding: chunked
Vary: RSC, Next-Router-State-Tree, Next-Router-Prefetch
  
```

Figure 39

### api/messages/route.js

This file also initiates POST request which handles addition of new record in Messages database table. Below figure illustrates, when user send “Hi” message, that text is sent along with unique “\_id” and “createdAt” timestamp.

The screenshot shows the VV CHAT mobile application interface. On the left, there's a camera icon and a message input field with the placeholder 'Write a message...'. In the center, a message bubble contains the text 'Hi' with a timestamp '11:10 PM'. At the bottom, there are tabs for 'Chats' and 'Contacts', and a navigation bar with icons for back, forward, and search. To the right of the app is the Chrome DevTools Network tab. It shows a POST request to 'http://localhost:3000/api/messages'. The 'Headers' tab shows the request headers, and the 'Response' tab shows the JSON response:

```

{
  "chat": "66d71c7154301702bdff4a12e",
  "createdAt": "2024-09-04T17:40:13.993Z",
  "photo": "",
  "seenBy": ["66d715dc54301702bdff4a00b"],
  "sender": {
    "_id": "66d715dc54301702bdff4a00b",
    "username": "Vishnu",
    "email": "vishnu@mail.com"
  },
  "text": "Hi",
  "_id": "66d89b7d5da0ff3435e2386d"
}
  
```

Figure 40

As soon as a new message is shared by clicking the send icon, a POST request is called including the “chatId”, “currentUserId” and “text” as Payload.

Name	Value
dhdcjazfh.json?sources[]=lo...	dhdcjazfh.json?sources[]=lo...
powered-by-new.svg	powered-by-new.svg
session	66d715dc54301702bd4a00b
	66d71c7154301702bd4a12e
	66d71c7154301702bd4a12e
messages	messages
session	66d715dc54301702bd4a00b
	66d71c7154301702bd4a12e
	66d71c7154301702bd4a12e

Headers	Payload	Preview	Response	Initiator	Timing	Cookies
<b>Request Payload</b> view source <pre>{   chatId: "66d71c7154301702bd4a12e",   currentUserId: "66d715dc54301702bd4a00b",   text: "Hi" }</pre>						
<b>Response</b> view source <pre>HTTP/1.1 200 OK Content-Type: application/json; charset=UTF-8 Date: Wed, 04 Sep 2024 17:40:14 GMT Keep-Alive: timeout=5 Transfer-Encoding: chunked Vary: RSC, Next-Router-State-Tree, Next-Router-Prefetch</pre>						

Figure 41

Upon successful delivery of the message, 200 OK Response is achieved.

Name	Value
dhdcjazfh.json?sources[]=lo...	dhdcjazfh.json?sources[]=lo...
powered-by-new.svg	powered-by-new.svg
session	66d715dc54301702bd4a00b
	66d71c7154301702bd4a12e
	66d71c7154301702bd4a12e
messages	messages
session	66d715dc54301702bd4a00b
	66d71c7154301702bd4a12e
	66d71c7154301702bd4a12e

Headers	Payload	Preview	Response	Initiator	Timing	Cookies
<b>General</b> Request URL: http://localhost:3000/api/messages Request Method: POST Status Code: 200 OK Remote Address: [:1]:3000 Referrer Policy: strict-origin-when-cross-origin						
<b>Response Headers</b> Connection: keep-alive Content-Type: text/plain;charset=UTF-8 Date: Wed, 04 Sep 2024 17:40:14 GMT Keep-Alive: timeout=5 Transfer-Encoding: chunked Vary: RSC, Next-Router-State-Tree, Next-Router-Prefetch						
<b>Request Headers (17)</b>						

Figure 42

## api/users/route.js

This file contains a GET request which retrieves details of all the users. Whenever new user registers in our web application, a new record is appended to the database. Hence, the response of this request will get all the “\_id” of users.

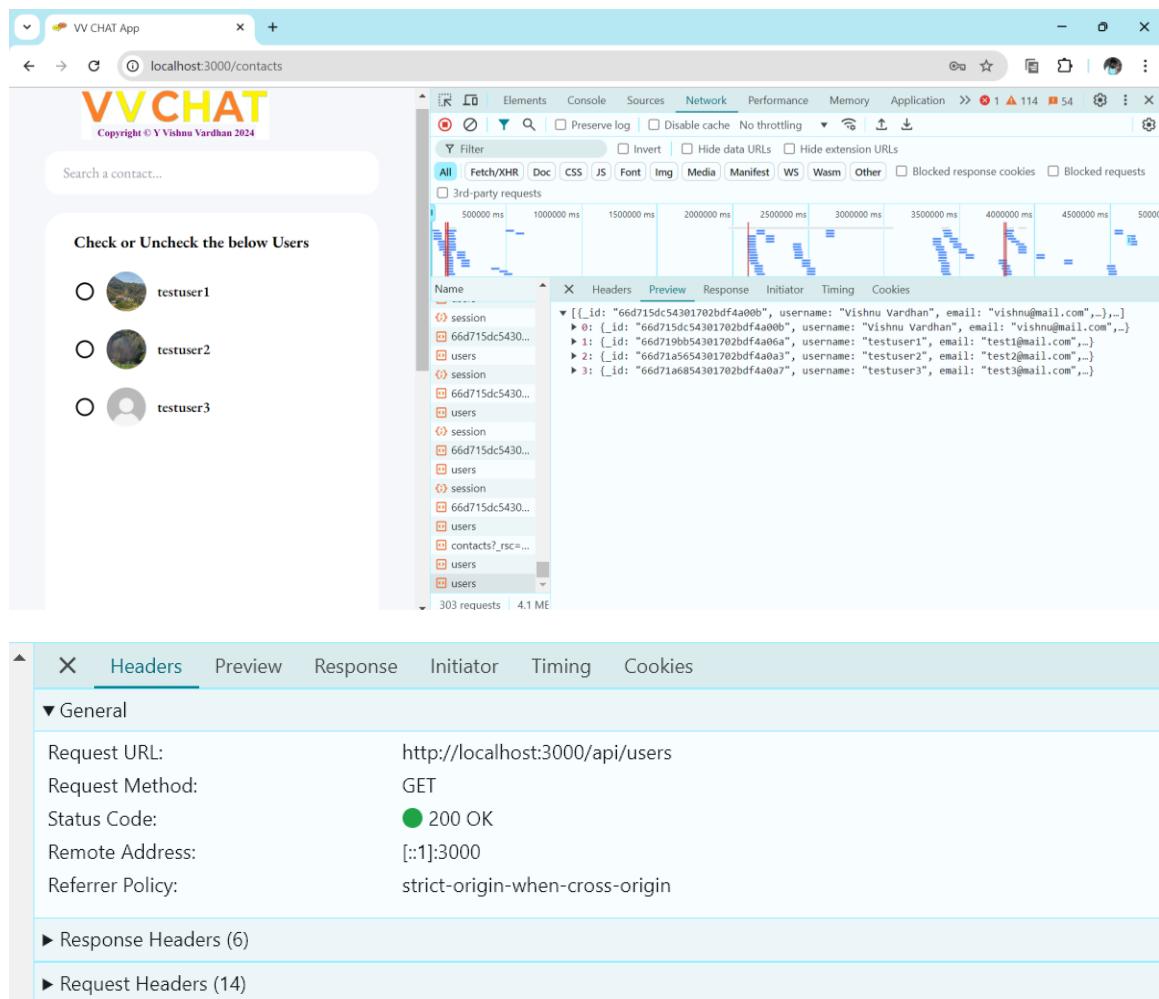


Figure 43

## api/users/searchContact/[query]/route.js

This file contains GET request which helps users to search for a particular contact, so that user can continue to start a new chat or continue existing chat.

Below figure depicts among various users, through this endpoint how username is filter out.

The screenshot shows two browser windows side-by-side.

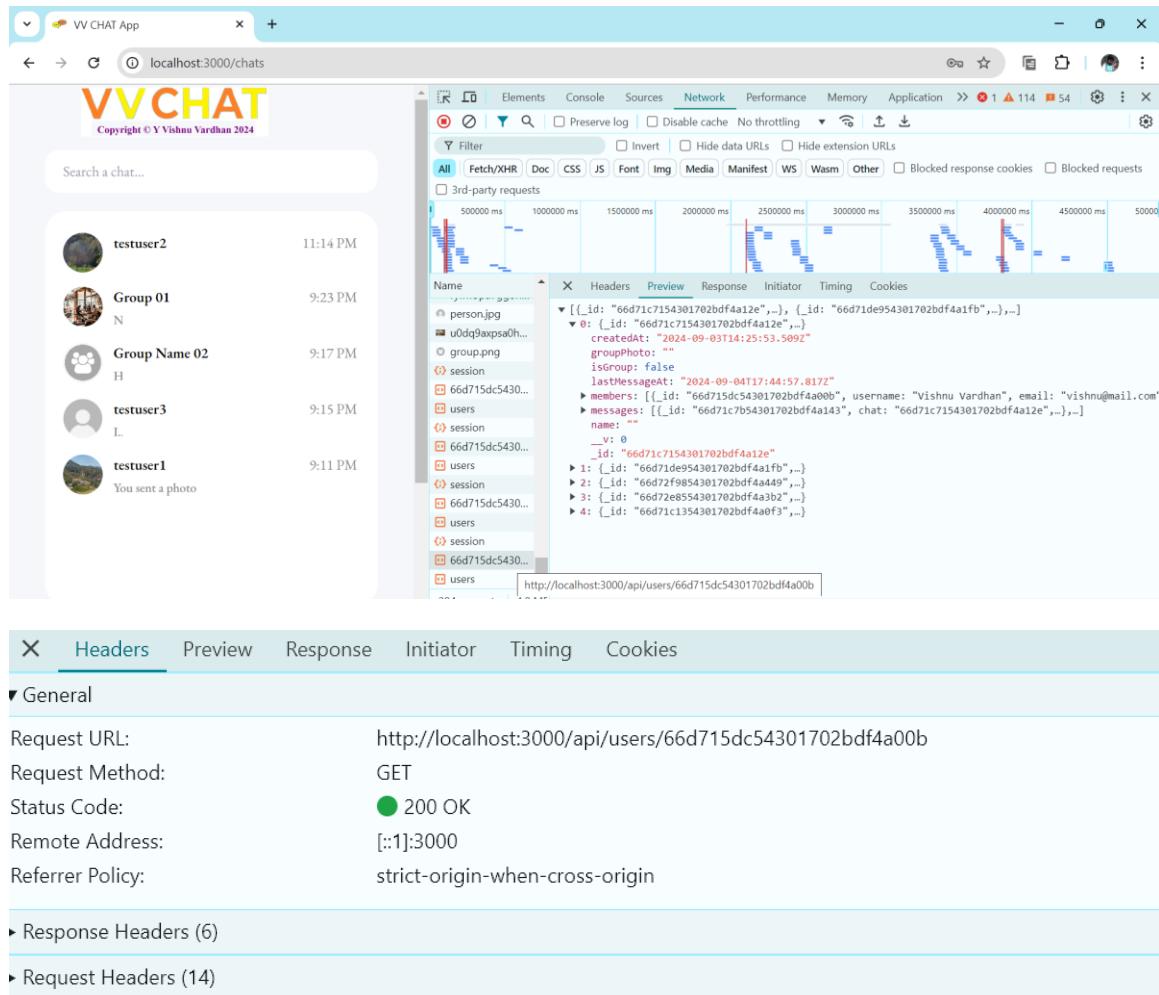
**Left Window:** A web browser window titled "VV CHAT App" showing the URL "localhost:3000/contacts". The page displays the VVCHAT logo and copyright information. It features a search bar labeled "Search a contact..." and a section titled "Check or Uncheck the below Users" containing three user entries: "Vishnu Vardhan" (selected), "testuser2", and "testuser3".

**Right Window:** A web browser window titled "VV CHAT App" showing the URL "localhost:3000/contacts". This window includes a developer tools interface with the "Network" tab selected. The Network tab displays a timeline of network requests. One specific request is highlighted: a GET request to "http://localhost:3000/api/users/searchContact/Vishnu". The request details show a status code of 200 OK and a referrer policy of "strict-origin-when-cross-origin".

Figure 44

## api/users/[userId]/route.js

This file contains a GET request which retrieves all chats of the current user. To do that, this component gets interacted with all the 3 tables (Chat, Message, User) of the database.



**Figure 45**

### api/users/[userId]/update/route.js

This backend component will handle user modification of username or profile picture.

The screenshot shows the VV CHAT application interface. At the top, there's a header with the logo and the text "VV CHAT" and "Copyright © Y Vishnu Vardhan 2024". Below it is a "Update Your Profile" form with fields for "Name" (Vishnu Vardhan) and "Upload" (a placeholder for a profile picture). A "Save Changes" button is visible. At the bottom, there are tabs for "Chats" and "Contacts". On the right side of the screen, the browser's developer tools Network tab is open, showing a list of requests. One request is highlighted: "profile?\_rsc=1...". The details panel for this request shows the following information:

- Request URL:** http://localhost:3000/api/users/66d715d54301702bd4a00b/update
- Request Method:** POST
- Status Code:** 200 OK
- Remote Address:** [::]:13000
- Referrer Policy:** strict-origin-when-cross-origin
- Response Headers:**
  - Connection: keep-alive
  - Content-Type: text/plain; charset=UTF-8
  - Date: Wed, 04 Sep 2024 18:08:21 GMT
  - Keep-Alive: timeout=5
  - Transfer-Encoding: chunked
  - Vary: RSC, Next-Router-State-Tree, Next-Router-Prefetch
- Request Headers (17):** A list of 17 header names and their values.

Figure 46

A POST request is sent giving the new username entered or new profile picture uploaded as payload. Upon successful updation, we achieve 200 OK Response.

The screenshot shows the Network tab of the developer tools with the "Payload" tab selected. The left sidebar lists various files and resources. In the main pane, under "Request Payload", the JSON payload is displayed:

```

{
  "username": "Vishnu Vardhan",
  "profileImage": "https://res.cloudinary.com/dhdcjazfh/image/upload/v1725372676/oa59kmpqjjcd7xhqjzrt",
  "username": "Vishnu Vardhan"
}

```

Figure 47

Whenever a new picture is uploaded, then Cloudinary cloud-based service will upload the new image with a specific URL and shares it to the backend in order to store it as “profileImage” attribute. That attribute is sent via Payload to the POST request.

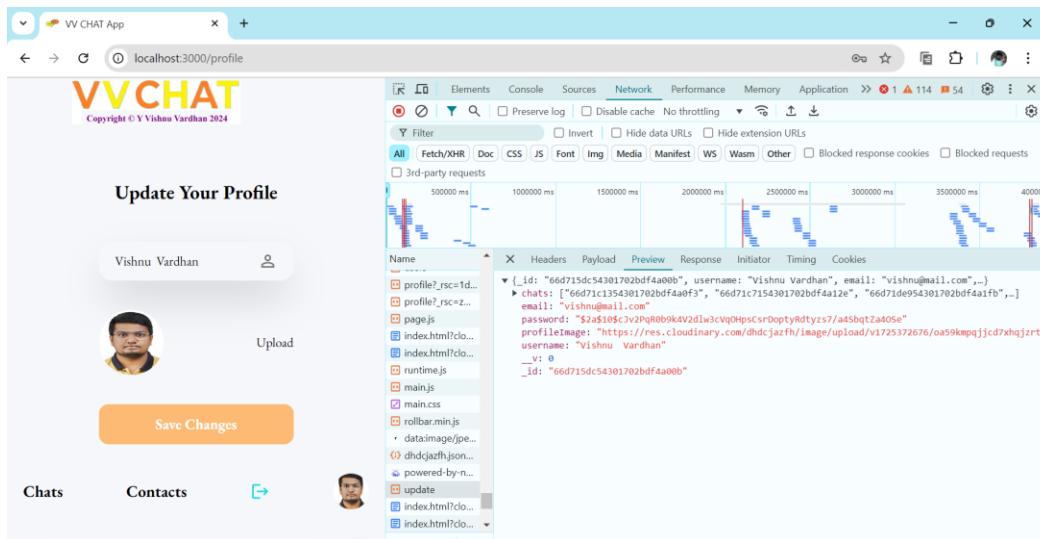


Figure 48

### api/users/[userId]/searchChat/[query]/route.js

This file helps users in searching a particular chat from the web application home page. GET request is called with endpoint including current logged in “userId” and [query] path is filled with the text entered in the UI search bar.

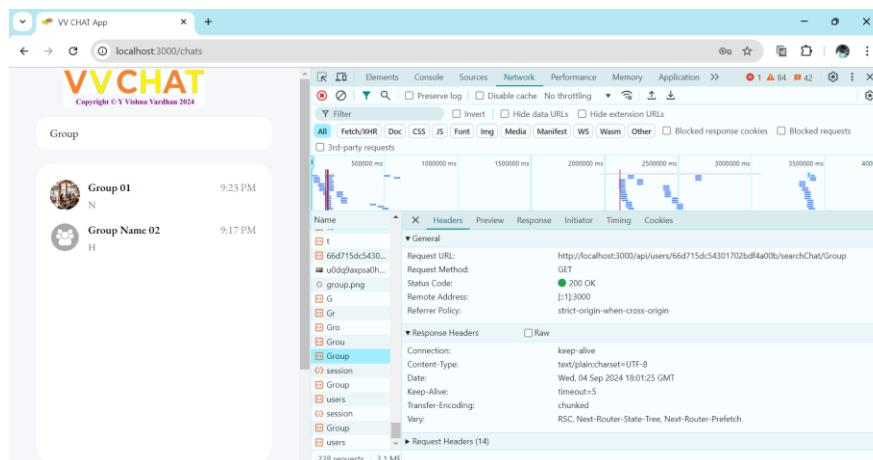


Figure 49

The response of the GET request contains the filtered ids of chats which got matched with the search bar text. In the below figure, we able to see when user searched chat entering “Group”, response retrieved has 2 chats with names “Group 01” & “Group Name 02”.

```

X Headers Preview Response Initiator Timing Cookies
▼ [ {_id: "66d71de954301702bdf4a1fb",...}, {_id: "66d72f9854301702bdf4a449",...} ]
  ▼ 0: {_id: "66d71de954301702bdf4a1fb",...}
    createdAt: "2024-09-03T14:32:09.819Z"
    groupPhoto: "https://res.cloudinary.com/dhdcjazfh/image/upload/v1725374127/u0dq9axpsa0hqyvojhn.jpg"
    isGroup: true
    lastMessageAt: "2024-09-03T15:53:04.001Z"
  ► members: [{_id: "66d715dc54301702bdf4a00b", username: "Vishnu", email: "vishnu@mail.com",...},...]
  ► messages: [{_id: "66d71f0554301702bdf4a259", chat: "66d71de954301702bdf4a1fb",...},...]
    name: "Group 01"
    __v: 0
    _id: "66d71de954301702bdf4a1fb"
  ▼ 1: {_id: "66d72f9854301702bdf4a449",...}
    createdAt: "2024-09-03T15:47:36.975Z"
    groupPhoto: ""
    isGroup: true
    lastMessageAt: "2024-09-03T15:47:45.735Z"
  ► members: [{_id: "66d715dc54301702bdf4a00b", username: "Vishnu", email: "vishnu@mail.com",...},...]
  ► messages: [{_id: "66d72fa154301702bdf4a45f", chat: "66d72f9854301702bdf4a449",...}]
    name: "Group Name 02"
    __v: 0
    _id: "66d72f9854301702bdf4a449"
  
```

**Figure 50**

### 4.3. Database Design

MongoDB is used for storing the data of users who registered, into “users” table

Similarly when a chat is created between two users or when a group chat is started, all the details of it are stored into “chats” table.

To avail persistent data regarding chat messages, each and every message sent through chat will be stored into the database in “messages” table.

#### Database name in MongoDB Atlas: VVCHATDB

The screenshot shows the MongoDB Atlas interface. The top navigation bar includes 'Atlas', 'Vishnu's Org...', 'Access Manager', and 'Billing'. Below this, the 'vvchat' project is selected. The main menu has tabs for 'Data Services', 'App Services', and 'Charts', with 'Data Services' currently active. On the left, a sidebar lists 'DEPLOYMENT' (Database, Data Lake), 'SERVICES' (Device & Edge Sync, Triggers, Data API, Data Federation, Atlas Search, Atlas Vector Search, Stream Processing, Migration), and 'vvchat' under 'vvchat'. The 'vvchat' section shows 'Overview', 'Real Time', 'Metrics', 'Collections' (selected), and 'Atlas Search'. The 'Collections' tab displays 'VVCHATDB' with 'LOGICAL DATA SIZE: 5.84KB'. It lists three collections: 'chats', 'messages', and 'users'. Each collection has a 'Collection Name' and 'Documents' column.

Collection Name	Documents
chats	
messages	
users	

Figure 51

In order to define schemas for these tables, under “models” directory I have created 3 JavaScript extension files.

**models/User.js**

**models/Message.js**

**models/Chat.js**

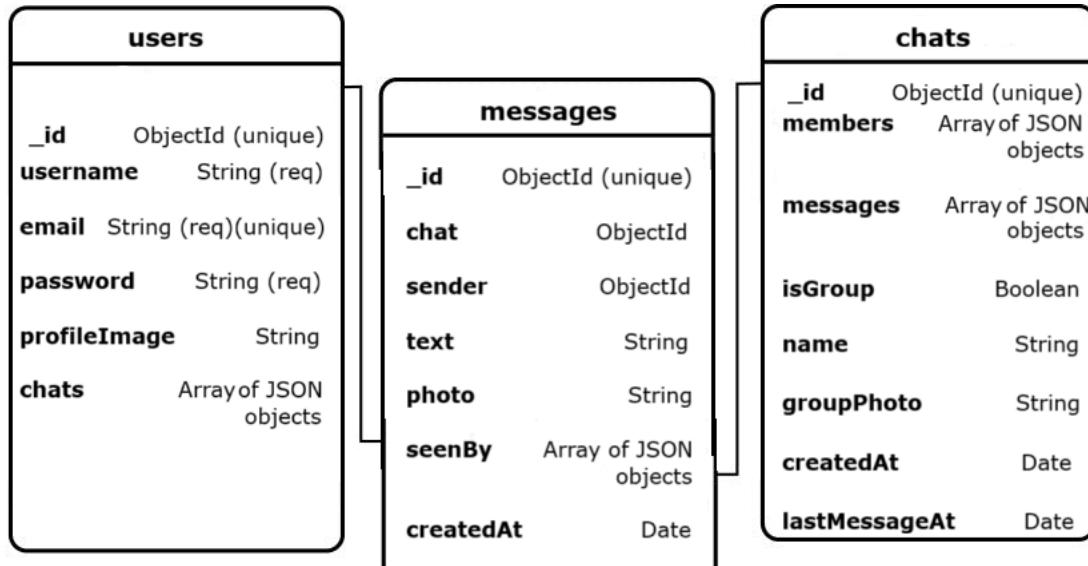


Figure 52

Below figures illustrate how data is stored in MongoDB.

QUERY RESULTS: 1-4 OF 4

```

users
{
  "_id": ObjectId('66d715dc54301702bdf4a00b'),
  "username": "Vishnu Vardhan",
  "email": "vishnu@mail.com",
  "password": "$2a$18$cJv2PqR0b9k4V2dlw3cVq0HpsCsrDoptyRdtyzs7/a4SbqtZa40Se",
  "profileImage": "https://res.cloudinary.com/dhdcjazfh/image/upload/v1725372676/oa59kmpq-",
  "chats": [
    {
      "0": ObjectId('66d71c1354301702bdf4a0f3'),
      "1": ObjectId('66d71c1354301702bdf4a12e'),
      "2": ObjectId('66d71de954301702bdf4a1fb'),
      "3": ObjectId('66d72e8554301702bdf4a3b2'),
      "4": ObjectId('66d72f9854301702bdf4a449'),
      "5": ObjectId('66d8aeb75da0ff3435e239c4')
    }
  ]
}
  
```

Figure 53



The screenshot shows a MongoDB Compass interface with a collection named "messages". A single document is selected, labeled "QUERY RESULTS:". The document contains the following data:

```

_id: ObjectId('66d71c3254301702bdf4a10a')
chat: ObjectId('66d71c1354301702bdf4a0f3')
sender: ObjectId('66d715dc54301702bdf4a00b')
text: "Hi Test User 1"
photo: ""
seenBy: Array (2)
  0: ObjectId('66d715dc54301702bdf4a00b')
  1: ObjectId('66d719bb54301702bdf4a06a')
createdAt: 2024-09-03T14:24:50.534+00:00

```

Figure 54



The screenshot shows a MongoDB Compass interface with a collection named "chats". A single document is selected, labeled "QUERY RESULTS:". The document contains the following data:

```

_id: ObjectId('66d71de954301702bdf4a1fb')
members: Array (4)
  0: ObjectId('66d715dc54301702bdf4a00b')
  1: ObjectId('66d719bb54301702bdf4a06a')
  2: ObjectId('66d71a5654301702bdf4a0a2')
  3: ObjectId('66d71a6854301702bdf4a0a7')
messages: Array (3)
  0: ObjectId('66d71f0554301702bdf4a259')
  1: ObjectId('66d7307c54301702bdf4a4c2')
  2: ObjectId('66d730e054301702bdf4a4fc')
isGroup: true
name: "Group 01"
groupPhoto: "https://res.cloudinary.com/dhdcjazfh/image/upload/v1725374127/u0dq9exp_"
createdAt: 2024-09-03T14:24:19.474+00:00
lastMessageAt: 2024-09-03T15:41:16.537+00:00

```

Figure 55

For the VV CHAT Web Application, Cloudinary Software as a Service (SaaS) cloud computing model is used as a database for storing image files. Upon successful integration of cloud details into the development code, we were able to see each and every picture uploaded from the web app gets stored with a unique Public ID for future retrievals.

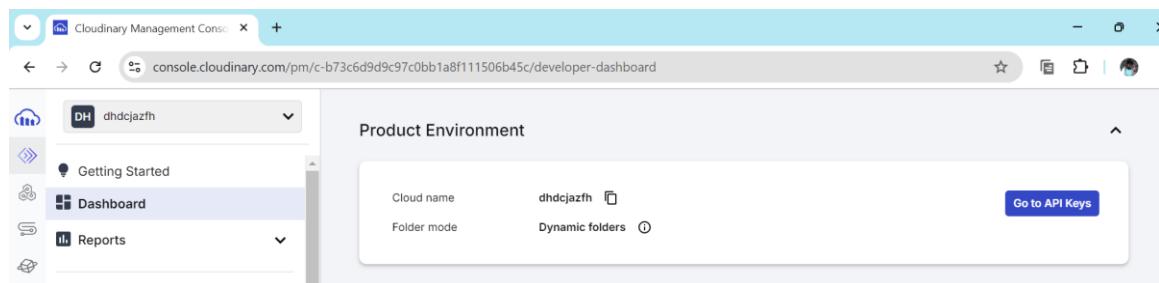
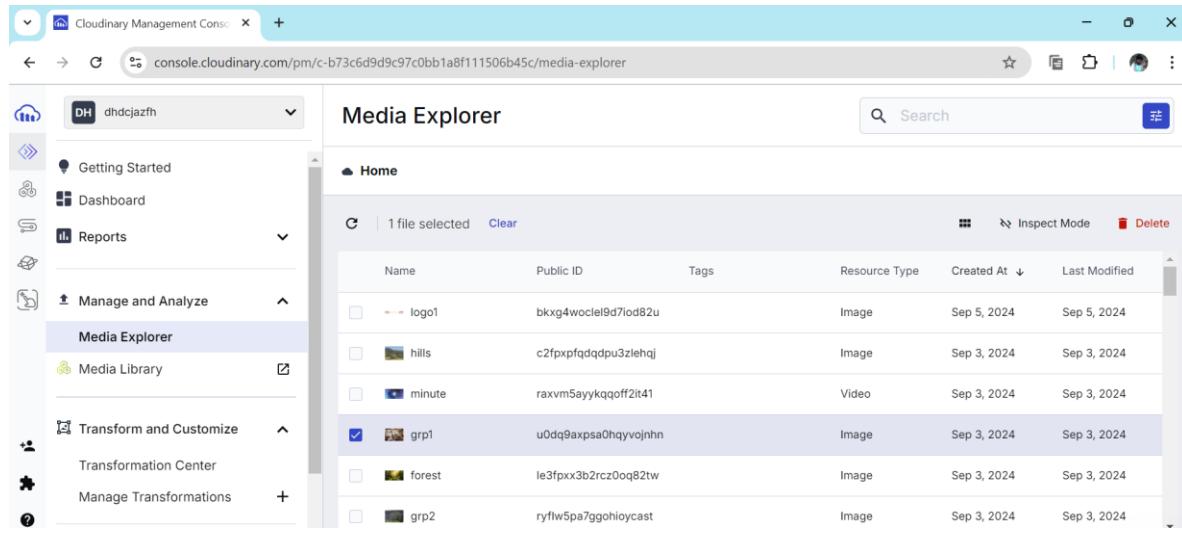


Figure 56

Below figure depicts, the Media Explorer of the Cloudinary where images sent via chatting and images uploaded as profile picture are stored.



**Figure 57**

#### 4.4. Security Design

The VV CHAT Web application handles sensitive data such as user passwords and message contents. In order to provide encryption, I implemented Bcrypt hashing algorithm by importing “hash” function from “bcryptjs” library. The algorithm uses salt rounds for creating hashes. It takes the plaintext password and a salt factor (optional) as input parameters and returns the hashed password asynchronously.

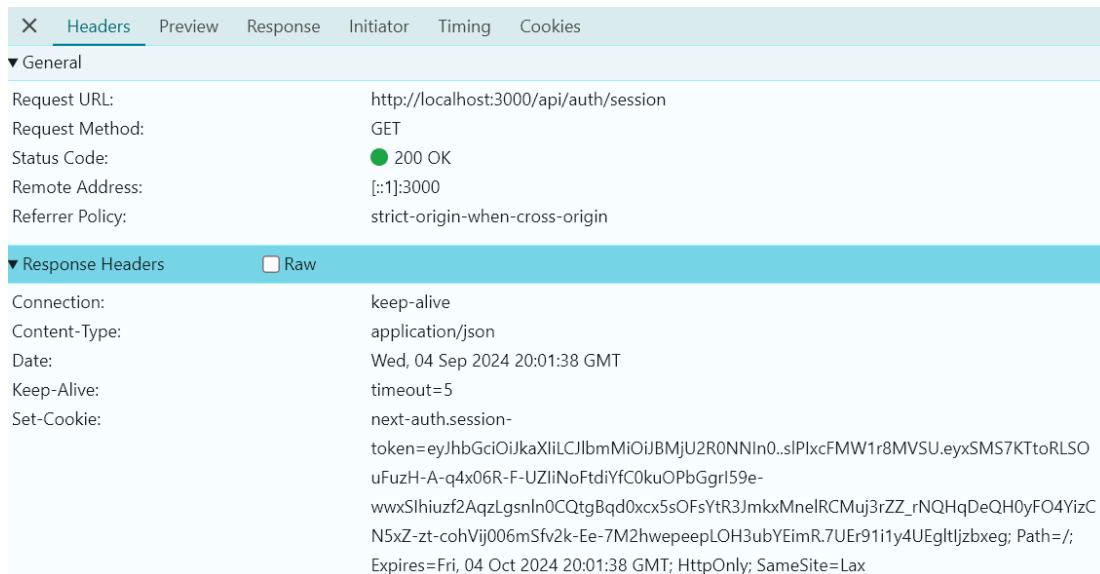
```
const hashedPassword = await hash(password, 10);

const newUser = await User.create({
  username,
  email,
  password: hashedPassword,
});

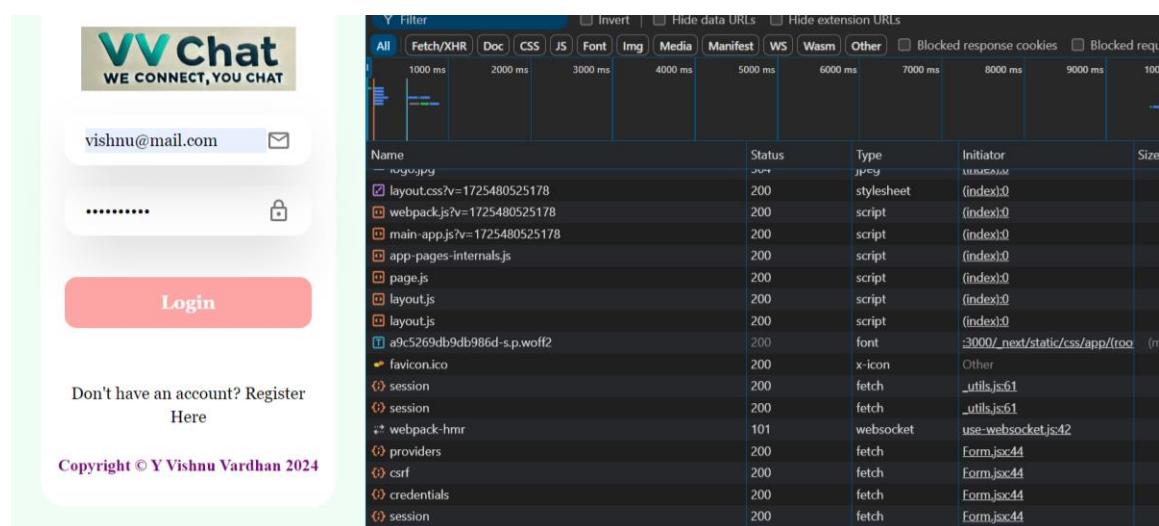
await newUser.save();
```

**Figure 58**

As the web app imported "next-auth version 4.24.7", it ensures user sessions by generating unique tokens.

**Figure 59**

Special attention taken during the initial Login page of the web app to enhance user authentication and authorization. In the Form component, I imported “signIn” module from the Nextauth.js Client API library. As soon as the user logs into the web application, this generates a CSRF token to prevent any CSRF attacks.

**Figure 60**

## 5. Non-Functional Requirements

### 5.1. Performance and Scalability

VV CHAT Web application was developed using optimized business logic codes by which future scalabilities are achievable. Backend API calls responses are retrieved with minimal latency. Until the response is acquired, UI guides the end users with loading icon. Since Atomic Design practice is used during development of the UI components, in future addition of new functionalities can be done in efficient manner and fast production is obtained. The fact that most of the files doing multiple jobs parallelly, I believe the data size of the complete project is appreciable during deployments and scaling.

### 5.2. Reliability and Availability

The VV CHAT Web Application is an UI responsive application, which gives users experience a good messaging service in any screen dimensions. Making use of various cascading styling libraries such as Tailwind CSS, React based @mui/icons-material and @emotion/styled make application more reliable.

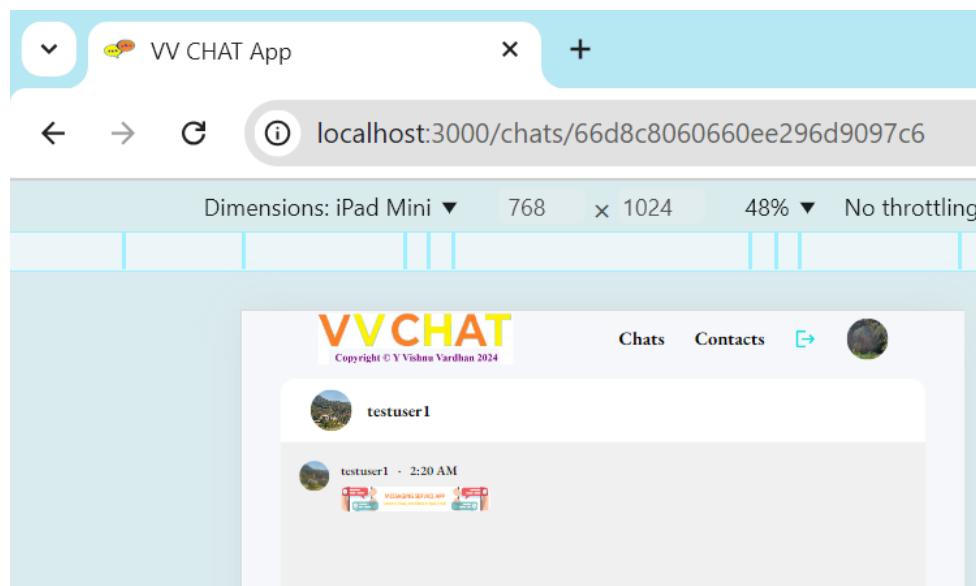
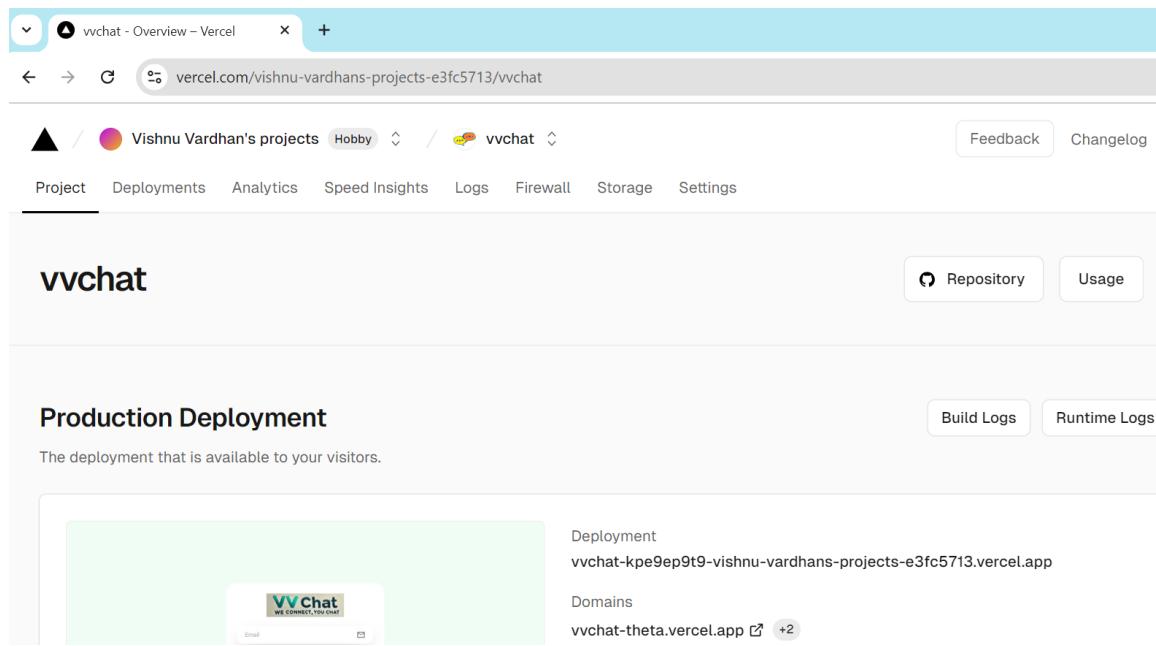


Figure 61

## 6. Deployment

I have chosen Vercel cloud platform as a service to deploy my VV CHAT web app. Since Vercel being the developer of Next.js, offers several features that make the deployment of Next.js project process easier and more efficient.



**Figure 62**

**Link for the deployed web application: <https://vvchat-theta.vercel.app/>**

For the access of local dependencies present in the business logic of the application, various third-party environment variables are updated in the Vercel dashboard. MongoDB database driver for accessing the database for information retrievals. Next-auth.js secret key to carry forward user authentications. Cloudinary Cluster name in order to access the image files uploaded through the web application. Lastly, Pusher credentials in order to facilitate real time message updates to the end users.

The screenshot shows the Vercel project overview for 'vvchat'. The 'Environment Variables' section is selected. It lists several environment variables with their values and deployment stages:

- MONGODB\_URL (Development, Preview, Production)
- NEXTAUTH\_SECRET (Development, Preview, Production)
- NEXT\_PUBLIC\_CLOUDINARY\_CLOUD\_NAME (Development, Preview, Production)
- PUSHER\_APP\_ID (Development, Preview, Production)
- NEXT\_PUBLIC\_PUSHER\_APP\_KEY (Development, Preview, Production)
- PUSHER\_SECRET (Development, Preview, Production)

Each variable has a 'Last Updated' timestamp (1d ago) and a three-dot menu icon.

**Figure 63**

Upon successful connection establishment to my Git Repository, Vercel deployed my application onto a valid domain taking care of all sorts of integrations, Data cache, Logs and Source Protection, Git Fork Protection, and Vercel’s authentication.

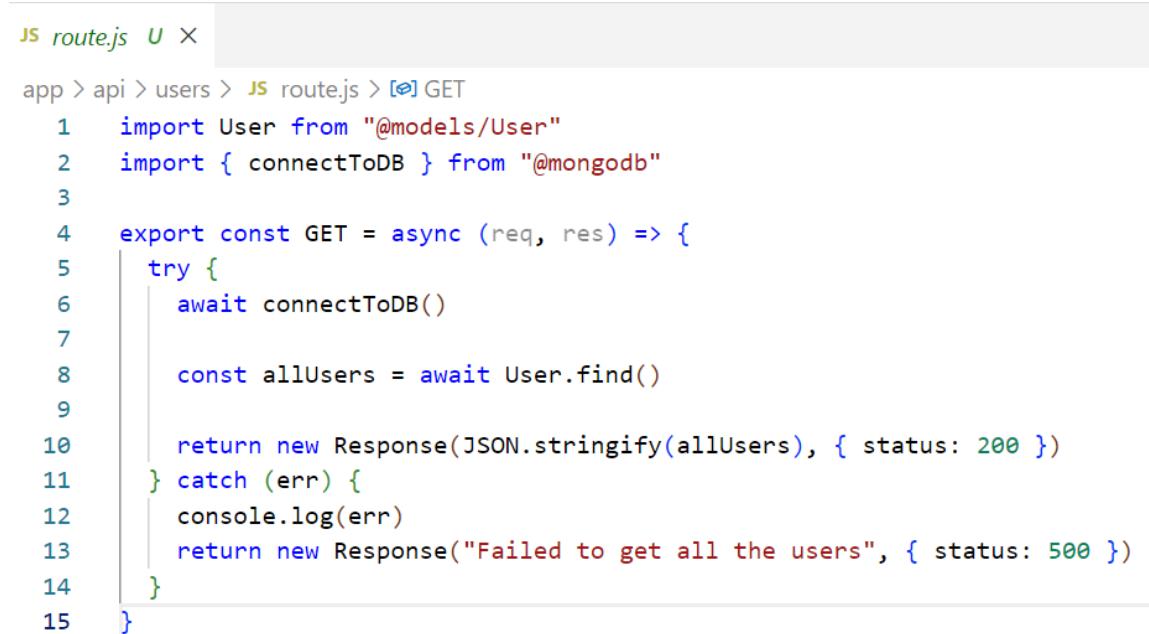
The screenshot shows two sections of the Vercel interface:

- Connected Git Repository:** Shows a connection to 'VishnuVardhanmusic/VV\_CHAT' which was connected 1d ago. A 'Disconnect' button is available.
- Deployment Status:** Shows the deployment 'vvchat-theta.vercel.app' is in 'Production'. It indicates a 'Valid Configuration' and is 'Assigned to main'. Buttons for 'Refresh' and 'Edit' are present.

**Figure 64**

## 7. Error Handling and Logging

Throughout the development of backend components, I have made use of try-catch block mechanism to handle user driven errors.



```

JS route.js U X
app > api > users > JS route.js > [o] GET
1 import User from "@models/User"
2 import { connectToDB } from "@mongodb"
3
4 export const GET = async (req, res) => {
5   try {
6     await connectToDB()
7
8     const allUsers = await User.find()
9
10    return new Response(JSON.stringify(allUsers), { status: 200 })
11  } catch (err) {
12    console.log(err)
13    return new Response("Failed to get all the users", { status: 500 })
14  }
15}

```

**Figure 65**

Based on the JSON Response, code firstly enters the try block where required business logic is written. If status other than 2xx (Success) is received, then the code enters the catch block where the error message, depending on whether to display it to the end user or console log at the backend will be done.

As a good practice, I figured out the error status for a few API calls and incorporated those in the error message. Rarely, 4xx client error and 5xx server error are encountered, which are efficiently handled by the web application.

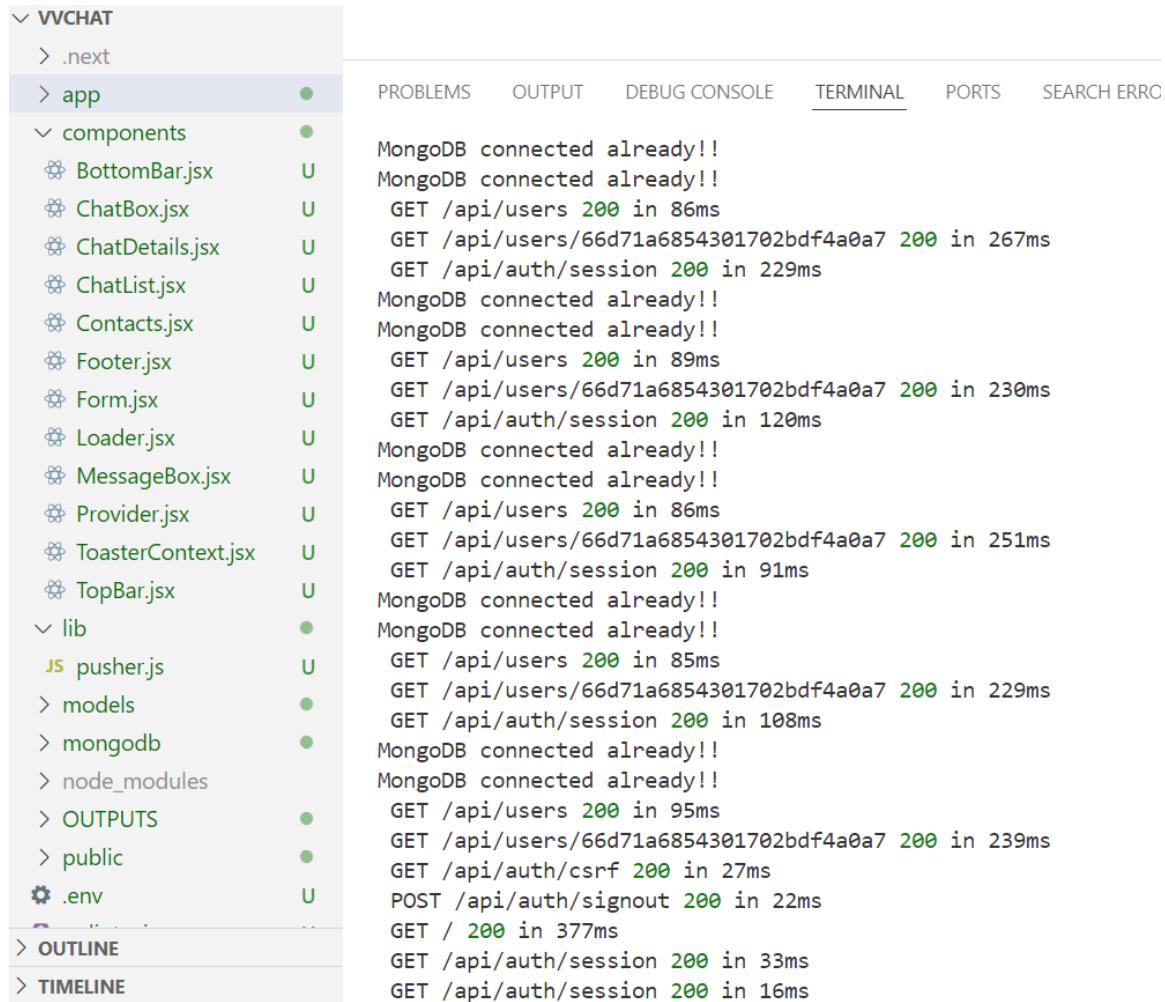
Below figure illustrates, when user tries to register with already entered email. Then 400 Bad Request response is logged onto the console.

The figure consists of three main parts:

- VV CHATDB Compass:** A screenshot of the MongoDB Compass interface showing the "users" collection. It displays a document with the following fields and values:
  - `_id: ObjectId('66d719bb54301702bdf4a06a')`
  - `username : "testuser1"`
  - `email : "testi@mail.com"`
  - `password : "$2a$10$7ac8xxBv3rrb0tCaCsAZFeHE/DlJ7r4S8Xai0vqDqpift2Ik8zeQW"`
  - `profileImage : "https://res.cloudinary.com/dhdcjazfh/image/upload/v1725372898/zu5yujqf..."`
  - `chats : Array (6)`
  - `__v : 0`
- Browser Screenshot:** A screenshot of a web browser window titled "VV CHAT App" showing the URL "localhost:3000/register". The page displays a registration form with fields for "username", "email", and "password". Below the form, a message says "Already registered? Log In". The developer tools' Network tab shows a POST request to "http://localhost:3000/api/auth/register" with a status of 400 (Bad Request).
- Network Tab in Chrome DevTools:** A screenshot of the Network tab in Chrome's developer tools. It shows a request named "register" with a status of "1 User already exists". The "Response" tab shows the JSON response: `{ "error": "User already exists" }`.

Figure 66

Logging Strategy is vital during the run of any full stack application. Since there will be numerous function calls rendering which need to be logged properly for analyzing any code breaks and business login failure cases, I console logged all appropriate messages which were considered during unit testing and end-to-end testing.



The screenshot shows the VS Code interface with the terminal tab selected. The terminal window displays a series of log messages from the application's runtime. The log includes MongoDB connection status, API requests (GET /api/users, GET /api/auth/session), and other internal logs. The left sidebar shows the project structure under the 'VVCHAT' folder, including '.next', 'app' (with components like BottomBar.jsx, ChatBox.jsx, etc.), 'lib' (with pusher.js), 'models', 'mongodb', 'node\_modules', 'OUTPUTS', 'public', and '.env'.

```

VVCHAT
  > .next
    > app
      > components
        > BottomBar.jsx
        > ChatBox.jsx
        > ChatDetails.jsx
        > ChatList.jsx
        > Contacts.jsx
        > Footer.jsx
        > Form.jsx
        > Loader.jsx
        > MessageBox.jsx
        > Provider.jsx
        > ToasterContext.jsx
        > TopBar.jsx
      > lib
        > pusher.js
      > models
      > mongodb
      > node_modules
      > OUTPUTS
      > public
      > .env
    > OUTLINE
    > TIMELINE

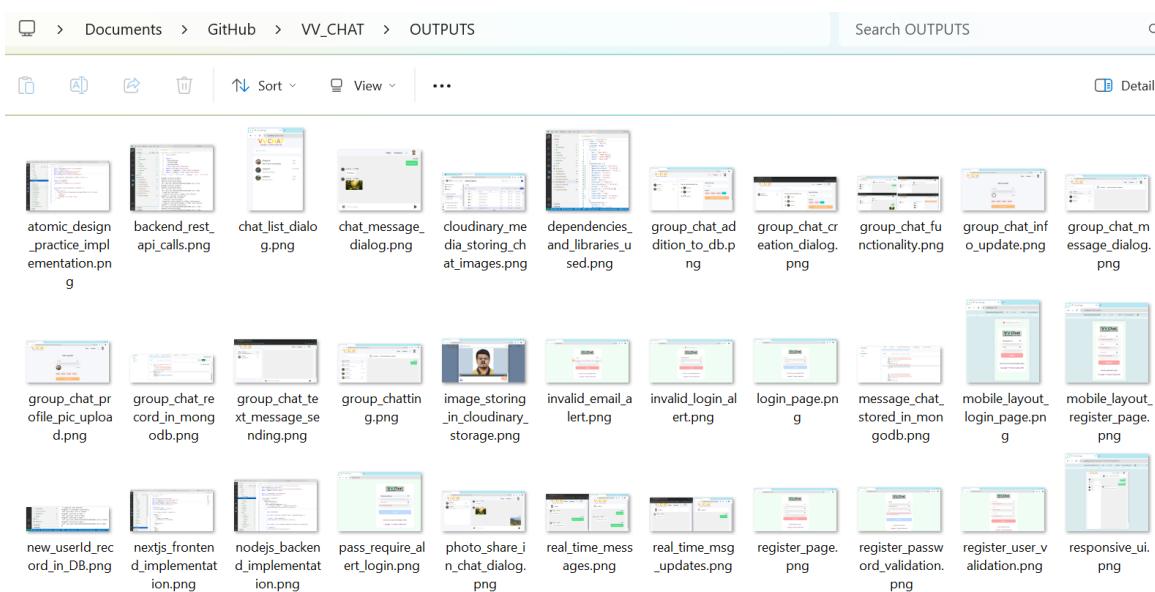
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   SEARCH ERRORS
MongoDB connected already!!
MongoDB connected already!!
GET /api/users 200 in 86ms
GET /api/users/66d71a6854301702bdf4a0a7 200 in 267ms
GET /api/auth/session 200 in 229ms
MongoDB connected already!!
MongoDB connected already!!
MongoDB connected already!!
GET /api/users 200 in 89ms
GET /api/users/66d71a6854301702bdf4a0a7 200 in 230ms
GET /api/auth/session 200 in 120ms
MongoDB connected already!!
MongoDB connected already!!
GET /api/users 200 in 86ms
GET /api/users/66d71a6854301702bdf4a0a7 200 in 251ms
GET /api/auth/session 200 in 91ms
MongoDB connected already!!
MongoDB connected already!!
GET /api/users 200 in 85ms
GET /api/users/66d71a6854301702bdf4a0a7 200 in 229ms
GET /api/auth/session 200 in 108ms
MongoDB connected already!!
MongoDB connected already!!
GET /api/users 200 in 95ms
GET /api/users/66d71a6854301702bdf4a0a7 200 in 239ms
GET /api/auth/csrf 200 in 27ms
POST /api/auth/signout 200 in 22ms
GET / 200 in 377ms
GET /api/auth/session 200 in 33ms
GET /api/auth/session 200 in 16ms

```

Figure 67

## 8. Testing

By inculcating popular testing strategies in order to build error-free, optimized web application, I have done testing after each Git commit. During the development stage, frequently ensured the VV CHAT app having proper interactions with each and every component. Appropriate snapshots to depict the PASS of testing scenarios, are engulfed into OUTPUTS directory.



**Figure 68**

Utilized the npm-package-json-lint to check multiple audits like validity of data types in nodes, whether a strings declared are lowercases, whether a version number is a valid, the presence of a given module, the presence of any bugs and validation of dependencies and licenses.

```
added 237 packages, and audited 510 packages in 22s

158 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

Figure 69

## 9. Abbreviations

<b>VV CHAT</b> => Vishnu Vardhan CHAT	<b>UI</b> => User Interface
<b>SQL</b> => Structured Query Language	<b>AI</b> => Artificial Intelligence
<b>REST</b> => Representational State Transfer	<b>NoSQL</b> => Not Only SQL
<b>API</b> => Application programming interface	<b>UX</b> => User Experience
<b>CRUD</b> => Create, Read, Update, and Delete	<b>CSS</b> => Cascading Style Sheets
<b>DBMS</b> => Data Base Management Systems	<b>CSRF</b> => Cross-site request forgery
<b>HTML</b> => Hypertext Markup Language	<b>OAuth</b> => Open Authorization
<b>HTTP</b> => Hypertext Transfer Protocol	<b>JSON</b> => JavaScript Object Notation

\*\*\*\*\*