

A
MINI-PROJECT REPORT
ON

“Chat Application”

Vishal Rajkumar Sangole

ABSTRACT

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 are also taken.

TABLE OF CONTENT

Chapter		Page No.
Chapter 1: Introduction		1
1.1	Overview	1
1.2	Background Study	2
1.3	Purposes	2
Chapter 2: System Design		3
2.1	Requirements	3
2.2	Functionality	3
2.3	Architecture	3
Chapter 3: Hardware and Software Requirement		6
3.1	Hardware Requirement	6
3.2	Software Requirement	6
Chapter 4: Implementing Tools for the Project		7
4.1	HTML	7
4.2	CSS	7
4.3	JavaScript	7
4.4	Firebase	8
4.5	Netlify	8
4.6	GitHub	8
Chapter 5: Working		9
5.1	Component Architecture	9
5.2	State Management	10
5.3	Authentication	10
5.4	Messaging System	11
Chapter 6: Project Database		17

6.1	Database	17
6.2	Collections	17
Chapter 7: Project Model View		20
7.1	Login	20
7.2	Sign up	20
7.3	Home page	21
Chapter 8: Conclusion		22
8.1	Conclusion	22

CHPATER: 1

INTRODUCTION

1.1 Overview

messaging applications are platforms that allow users to communicate with each other through text messages and other forms of digital communication. These applications are designed to be interactive and user-friendly, allowing users to easily connect with friends, family, and colleagues. Social media messaging web applications are often used for personal communication but can also be used for business purposes such as customer service and marketing, collaborating with colleagues and clients, or sharing information and media. Messaging applications can run on various devices, such as smartphones, tablets, laptops, or desktop computers, and can connect users across different platforms and networks.

messaging applications are becoming increasingly popular due to their convenience and accessibility. They allow users to communicate with each other in real-time from anywhere in the world. Some popular social media messaging web applications include WhatsApp, Facebook Messenger, Viber, WeChat, and Telegram. These applications offer a wide range of features such as group chats, voice and video calls, file sharing, and more.

We are trying to build a basic Realtime messaging web-based messaging application, in which users can communicate with each other in a fast and convenient way. Our application will allow users to create and join chat rooms, send and receive text, image. Our application will also use WebSocket technology to enable bidirectional communication between the server and the clients, ensuring that messages are delivered instantly and securely, without any delays or intermediaries. Our goal is to provide a simple and user-friendly messaging platform that can meet the needs and expectations of our target audience.

1.2 Background Study

The idea for the app was born out of a need to connect students who may not have had the opportunity to meet in person due to the COVID-19 pandemic. The app aims to provide a platform for students to socialize, study together, and build meaningful relationships.

We have used a variety of web development technologies, including HTML, CSS and JavaScript, to create a real-time chat platform that could handle multiple users.

1.3 Purpose

The purpose of this project is to design a chat application, also known as a instant messaging system. The main purpose of the software is to provide users with an instant messaging tool that has the ability to handle millions of users simultaneously when needed and can be easily done.

- To provide a convenient and effective way for people to communicate with each other across different locations and devices.
- To create a community and a network of users who share similar interests, hobbies, or goals.
- To enhance the productivity and collaboration of teams and organizations by facilitating real-time communication and feedback.
- To innovate and experiment with new features and technologies that can improve the user experience and satisfaction.
- To learn about Messaging applications.

CHPATER: 2

System Design

2.1 Requirements

- **Functionality:** This app works as expected.
- **Efficiency:** It's fast responded in simple steps.
- **Intuitiveness:** It's self-explanatory and makes sense.
- **Attractiveness:** It's mut be pleasing to the eyes.

2.2 Functionality

First thing first, we must figure out the exact requirements of the target system.

- **Direct messaging:** two users can chat with each other.
- **User must be able to add his friends to chat List.**
- **User must be able to Login, logout.**
- **New user must be able to create New User Id.**

2.3 Architecture

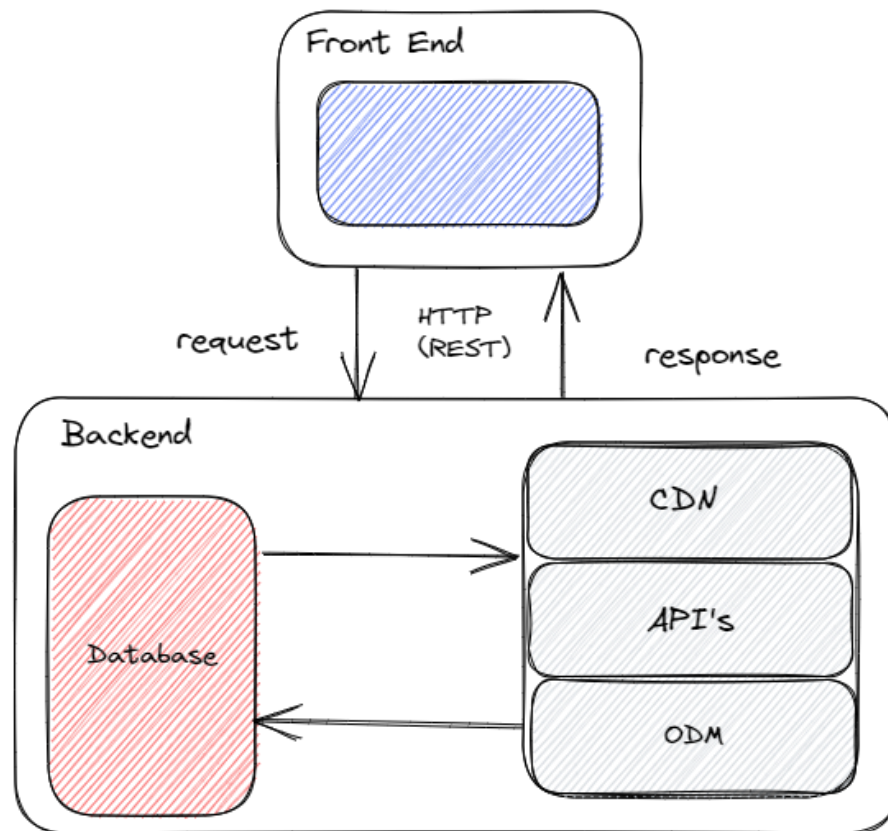
Architecture is divided into Two parts:

1. **Front-End**
2. **Back-End**

Front-end:

- **Design:**

Design should be responsive. As user can spend long time on app design should be minimum. New user learning curve should be very straight. While choosing colours we decided to go with low contrast colours.



Minimalist graphic design follows a few key rules. We created rule to refer to. Keep in mind that sometimes a minimalist design can work against you if there is no balance or if the empty spaces are too stark and overwhelming. Remember to keep it balanced, and that is a rule for all types of design.

- Client-side Logic:

Mostly client side has following task:

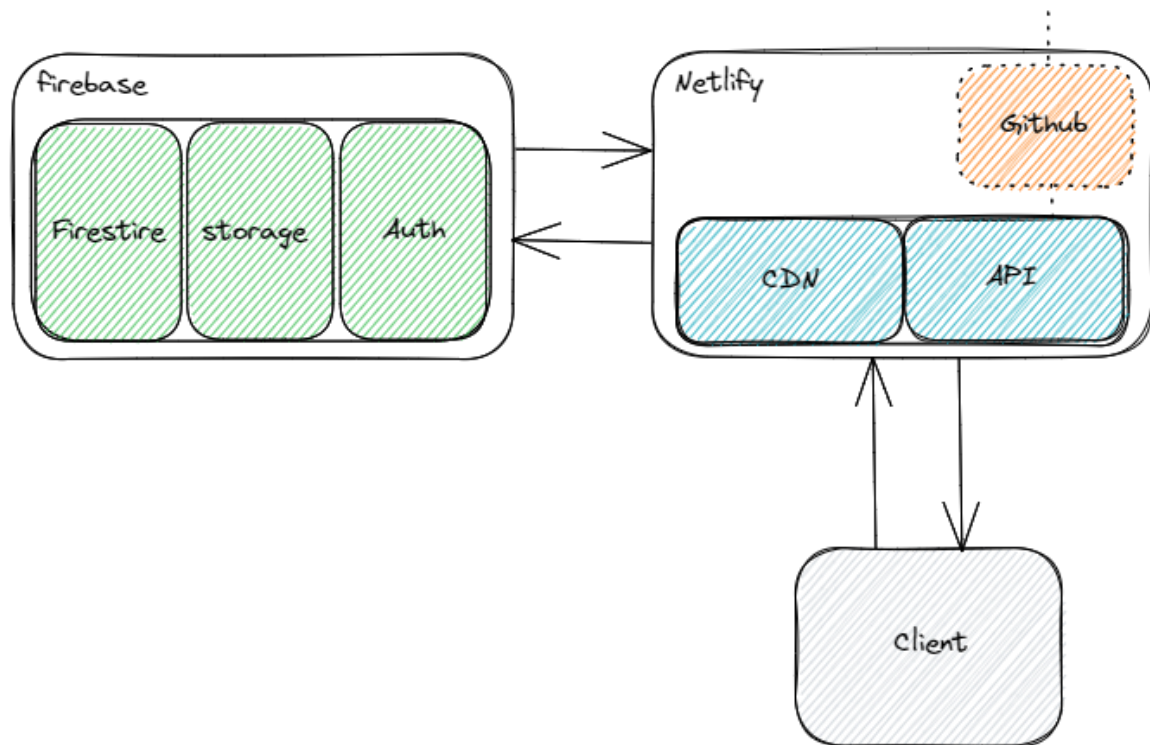
1. Dynamic UI changes (client-side renders)
2. State management
3. Fetching data from server

Request to server only goes when there is database update or information retrieval. all other task are self-handled by React on Client side. This architecture proves feasible for us as we do have lot going on client side and we also don not want to contact server unless it is related to messaging

Backend:

- Hosting:

For easy development purpose we have choose Netlify for hosting our application. It can spin-off server for web site with just git repository. Master branch of git repo is set as deployment branch. Netlify auto build and deploy site if any change happens to master branch. This make development, deployment



process way faster than traditional method as our application does not need self-deployed server.

- Authentication and Database:

Main reason to choose firebase because it provides multiple services. we decide to use Authentication, Fire store database and firebase storage service. As reinventing wheel is always waste of time using these prebuild service reduces work for developer and always provide best possible service. Detailed breakdown of Architecture provided in following sections.

CHPATER: 3

Hardware and Software Requirement:

3.1 Hardware Required

- Processor: Pentium IV or Above
- RAM :2GB or above
- Hard Disk: 50GB or above
- Input Devices: Keyboard, Mouse
- Output Devices: Monitor

3.2 Software Required

- Operating System: Linux, Ubuntu, Mac, Windows XP, 7, 8, 8.1, 10
- Tools and technologies: HTML, CSS, Tailwind CSS, JavaScript, ReactJs, Firebase, Netlify, GitHub
- Software: Any browser of your choice

CHPATER: 4

Implementing Tools for the Project

4.1 HTML

HTML (Hypertext Markup Language) is the most basic building block of the Web. It defines the meaning and structure of web content. Other technologies besides HTML are generally used to describe a web page's appearance/presentation (CSS) or functionality/behaviour (JavaScript).

4.2 CSS

Cascading Style Sheets (CSS) is a stylesheet language used to describe the presentation of a document written in HTML or XML (including XML dialects such as SVG, MathML or XHTML). CSS describes how elements should be rendered on screen, on paper, in speech, or on other media.

- **Sass**

Sass is a CSS preprocessor that extends the capabilities of CSS by providing a variety of features such as variables, nesting, functions, and mixins¹. Generally, Sass is preferred to be used instead of CSS, as Sass is much rich in features than CSS².

4.3 JavaScript

JavaScript is a cross-platform, object-oriented scripting language used to make webpages interactive (e.g., having complex animations, clickable buttons, popup menus, etc.). There are also more advanced server-side versions of JavaScript such as Node.js, which allow you to add more functionality to a website than downloading files (such as Realtime collaboration between multiple computers).

- React

React is a JavaScript library created by Facebook. React allows developers to create reusable UI components and manage their state efficiently. React uses a virtual DOM (Document Object Model) which makes it faster than other frameworks.

4.4 Firebase

Firebase is a mobile and web application development platform that provides developers with a variety of tools and services to help them build high-quality applications quickly and easily. It became one of the most popular backend-as-a-service (BaaS) platforms. Firebase provides a wide range of features such as real-time database, authentication, hosting, storage, and more. It also has a robust set of APIs that allow developers to integrate Firebase with other services and platforms.

Service we have used in this Project are:

- Authentication
- Firestore
- Storage

4.5 Netlify

Netlify is a cloud-based hosting and serverless backend service that allows developers to build, deploy, and manage web applications and websites. It provides a variety of features such as continuous deployment, serverless functions, form handling, and more.

4.6 GitHub

GitHub is a web-based platform that allows developers to store and manage their code repositories. It provides a variety of features such as version control, issue tracking, and collaboration tools that make it easy for developers to work together on projects. GitHub is built on top of Git.

CHPATER: 5

Working

5.1 Component Structure

Component are structured as above shown. Root in index page that renders as first request comes. After that ReactDOM mount every component in hierarchical fashion to <div> in index.html which has "root" as its ID.

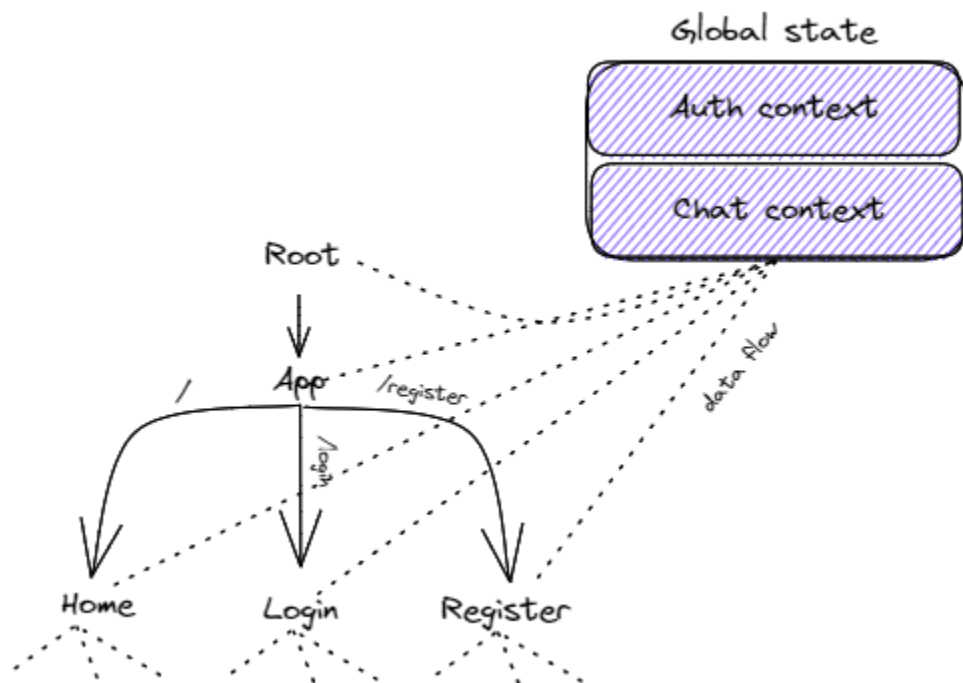
Routing:

Forward App component every component is rendered based on URL user is requesting on.

" / " : index route. Home components is rendered on this route.

" /login " : login component is rendered on this route.

" /register " : Register component is rendered in this route.



5.2 State Management

Client-side state is handle using React Context Api. Context provides a way to pass data through the component tree without having to pass props down manually at every level.

There are two Context which manages and provides two different context of data.

1. Auth context
2. Chat context

- Auth Context:

Auth context manages user session. The effect function calls the `onAuthStateChanged` function from the Firebase authentication library. The call-back function takes a user argument and is called whenever the authentication state changes. When this happens, the call-back function calls the `setCurrentUser` function to update the value of the `currentUser` state variable with the new user object.

- Chat Context:

code sets up a context for managing chat-related data within a React application.. The reducer function uses a switch statement to handle different action types. In this case, there is only one case for the "CHANGE_USER" action type. When this action is dispatched, the reducer returns a new state object with the `chatId` property set to a value that depends on the `uid` properties of the `currentUser` and the payload objects. The `user` property is set to the value of the payload.

5.3 Authentication

- Login:

Login component handles Login of user. The `handleSubmit` function is defined as an asynchronous function. This function is called when the form is submitted. it retrieves the values of the email and password fields from the form by accessing the `value` property of the first and second elements of

the target property of the event object. It does this by calling the `signInWithEmailAndPassword` function from the Firebase authentication library and passing in the `auth`, `email`, and `password` arguments. If the sign-in is successful, the function calls the `navigate` function to navigate to the root route of the application. If an error occurs during sign-in, the catch block is executed and sets the value of `err` to `true`.

- Register:

Register component handles user registration. The `handleSubmit` function is defined as an asynchronous function. This function is called when the form is submitted. it retrieves the values of the `displayName`, `email`, `password`, and file selected by the user from the form by accessing the `value` property of the first, second, and third elements of the target property of the event object. It try to create user by calling the `createUserWithEmailAndPassword` function from the Firebase authentication library and passing in the `auth`, `email`, and `password` arguments. If the user creation is successful, the function creates a reference to a storage location using the `ref` function from the Firebase storage library and passing in the storage object and `displayName`.

Finally, it navigates to root route of application by calling `navigate("/")`. If an error occurs during any of these operations, catch block is executed and sets value of `err` to `true`.

5.4 Messaging System

- Message submission:

```
export const Input = () => {
  const { currentUser } = useContext(AuthContext);
  const { data } = useContext(ChatContext);
  const [text, setText] = useState("");
  const [img, setImg] = useState(null);
  const handleSend = async () => {
    if(img){
      const storageRef = ref(storage, uuid());
      const uploadTask = uploadBytesResumable(storageRef, img);
      uploadTask.on(
        (error) => {
          console.log(error)
          // setErr(true);
        }
      );
    }
  }
}
```



```

    },
    () => {
      getDownloadURL(uploadTask.snapshot.ref).then(async(downloadURL) => {
        await updateDoc(doc(db , "chats", data.chatId), {
          messages: arrayUnion({
            id: uuid(),
            text,
            img:downloadURL,
            senderId: currentUser.uid,
            date: Timestamp.now(),
          }),
        });
      });
    }
  );
}else{
  await updateDoc(doc(db , "chats", data.chatId), {
    messages: arrayUnion({
      id: uuid(),
      text,
      senderId: currentUser.uid,
      date: Timestamp.now(),
    })
  })
  await updateDoc(doc(db, "userChats", currentUser.uid), {
    [data.chatId + ".lastMessage"]: {
      text,
    },
    [data.chatId + ".date"]: serverTimestamp(),
  })
  await updateDoc(doc(db, "userChats", data.user.uid), {
    [data.chatId + ".lastMessage"]: {
      text,
    },
    [data.chatId + ".date"]: serverTimestamp(),
  })
}
setText("");
setImg(null);
}
const handleChangeText = (e) => {
  setText(e.target.value);
}
const handleChangeImg = (e) => {
  setImg(e.target.files[0]);
}
return( .. ..
)
}

```

This code sets up functionality for sending messages within chat application. The handleSend function is defined as an asynchronous function that is called when the user wants to send a message. The function

first checks if the value of the `img` state variable is truthy. If it is, this means that the user has selected an image to send. In this case, the function creates a reference to a storage location using the `ref` function from the Firebase storage library and passing in the storage object and a unique identifier generated using the `uuid` function. It then starts an upload task for uploading the selected image to this storage location by calling the `uploadBytesResumable` function and passing in the storage reference and image. The upload task has an event listener attached using its `on` method.

If the upload completes successfully, the completion callback is called. This callback uses a promise chain to get a download URL for the uploaded image by calling the `getDownloadURL` function and passing in a reference to the uploaded file. It then updates a document in Firestore database with information about new message by calling `updateDoc()` method with `doc(db, "chats", data.chatId)` as first argument and object with new message data as second argument. If value of `img` state variable is falsy it means that user wants to send text message. In this case function updates document in Firestore database with information about new message by calling `updateDoc()` method with `doc(db, "chats", data.chatId)` as first argument and object with new message data as second argument.

After that it updates documents in Firestore database for both users with information about last message by calling `updateDoc()` method with `doc(db, "userChats", currentUser.uid)` or `doc(db, "userChats", data.user.uid)` as first argument and object with last message data as second argument. At end of function, it sets value of `text` state variable to empty string and value of `img` state variable to null. `ThandleChangeText` and `handleChangeImg` functions are event handlers for changes to the text and image inputs, respectively.

- Message displaying:

```
export const Messages = () => {
  const [ messages, setMessages ] = useState([]);
  const { data } = useContext(ChatContext);
  useEffect(() => {
    const unsub = onSnapshot(doc(db, "chats", data.chatId), (doc => {
      doc.exists() && setMessages(doc.data().messages)
    }))
    return ()=>{
      unsub();
    }
  }, [data.chatId])
}
```

```

    }
  }, [data.chatId])
  return(
    <div className="messages">
      {messages.map(message => <Message message={message} key={message.id}/>)}
    </div>
  )
}

```

This code sets up a component for displaying a list of messages within a chat application. The component uses the `useEffect` hook to set up a side effect that runs when the component is mounted and whenever the value of the `data.chatId` property changes. The effect function calls the `onSnapshot` function from the Firebase Firestore library and passes in a reference to a document in the database and a callback function.

The callback function takes a `doc` argument and is called whenever the data in the specified document changes. When this happens, the callback function checks if the document exists by calling its `exists` method. If it does, it calls the `setMessages` function to update the value of the `messages` state variable with the messages data from the document. The effect function returns a cleanup function that calls the `unsub` function returned by the `onSnapshot` function. This unsubscribes from the document snapshot listener when the component is unmounted or when the value of `data.chatId` changes.

- Searching for Friend:

```

export const Search = () => {
  const [username, setUsername] = useState("");
  const [user, setUser] = useState(null);
  const [err, setErr] = useState(false);
  const { currentUser } = useContext(AuthContext);
  const handleChange = (e) => {
    setUsername(e.target.value);
  }
  const handleSearch = async (e) => {
    const q = query(collection(db, "users"), where("displayName", "==", username));
    try{
      const querySnapshot = await getDocs(q);
      querySnapshot.forEach((doc) => {
        setUser(doc.data())
      });
    }catch(err){
      setErr(true);
    }
  }
}

```

```

    }
  }
  const handleKey = (e) => {
    e.code === "Enter" && handleSearch();
  };
  const handleSelect = async () => {
    const combinedId = currentUser.uid > user.uid
      ? currentUser.uid + user.uid
      : user.uid + currentUser.uid;
    try {
      const res = await getDoc(doc(db, "chats", combinedId))
      if(!res.exists()){
        await setDoc(doc(db, "chats", combinedId), {messages: []});
        await updateDoc(doc(db, "userChats", currentUser.uid), {
          [combinedId+".userInfo"] : {
            uid:user.uid,
            displayName: user.displayName,
            photoURL: user.photoURL,
          },
          [combinedId+".date"]:serverTimestamp()
        });
        await updateDoc(doc(db, "userChats", user.uid), {
          [combinedId+".userInfo"] : {
            uid:currentUser.uid,
            displayName: currentUser.displayName,
            photoURL: currentUser.photoURL,
          },
          [combinedId+".date"]:serverTimestamp()
        });
      }
    } catch (err) {
      setErr(true);
    }
    setUser(null);
    setUsername("");
  }

  return( .. ..
    )
}

```

This code defines a `Search` component using React. The `handleSearch` function is an asynchronous function that is called when the user wants to search for a user by their display name. It first creates a query object by calling the `query` function from the Firebase Firestore library and passing in a reference to the `users` collection in the database and a condition created using the `where` function. This condition specifies that only documents where the `displayName` field is equal to the value of the `username` state variable should be returned. The function then execute this query by calling the `getDocs` function and passing in the query object. If the query is successful, it iterates over each document in the query snapshot by calling its `forEach` method and passing in a callback function. The callback function

takes a ``doc`` argument and calls the ``setUser`` function to update the value of the ``user`` state variable with the data of this document. If an error occurs during query execution, catch block is executed and sets value of `err` state variable to `true`.

The `handleKey` function is an event handler for keydown events on username input. It checks if `code` property of event object is equal to `"Enter"`. If it is it calls `handleSearch()` method. The `handleSelect` function is an asynchronous function that is called when user wants to select user from search results. It first creates `combinedId` variable which is concatenation of `currentUser` `uid` and selected user `uid` sorted in ascending order. It then uses try-catch block to attempt to get document from Firestore database with `id` equal to `combinedId` by calling `getDoc()` method with `doc(db, "chats", combinedId)` as argument. If document doesn't exist it creates new document in Firestore database with `id` equal to `combinedId` and `messages` property set to empty array by calling `setDoc()` method with `doc(db, "chats", combinedId)` as first argument and object with `messages` property set to empty array as second argument. After that it updates documents in Firestore database for both users with information about new chat by calling `updateDoc()` method with `doc(db, "userChats", currentUser.uid)` or `doc(db, "userChats", user.uid)` as first argument and object with new chat data as second argument.

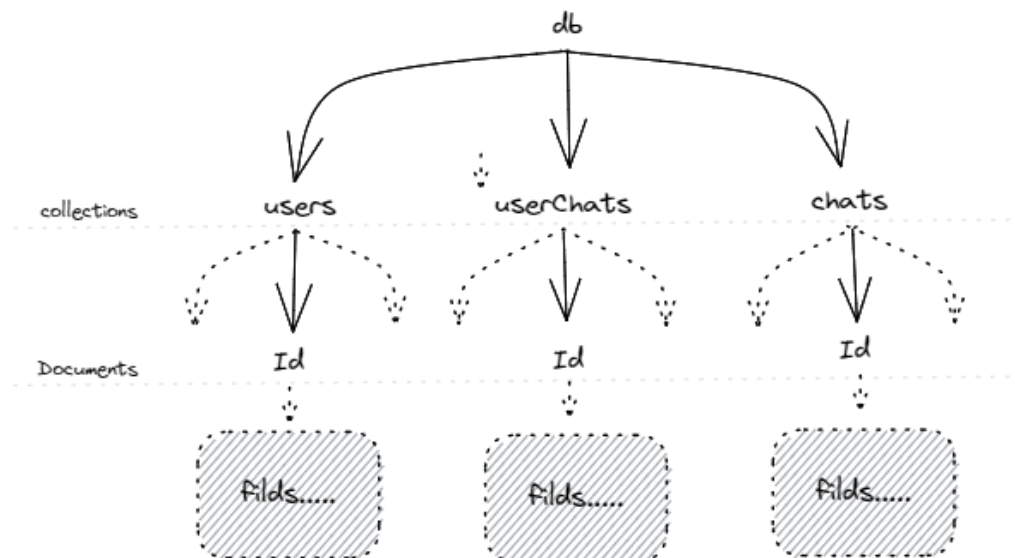
At end of function, it sets value of `user` state variable to `null` and value of `username` state variable to empty string.

CHPATER: 6

Project Database

6.1 Database

Project uses firestore from firebase. Firestore is a NoSQL document database built by Google. Firestore lets you easily store, sync, and query data for your mobile and web apps at a global scale. It has been designed to provide a better developer experience and simplify the development process



6.2 Collections

Database has 3 collections users, userChats and chats. Let's understand them one by one

- users:

users document schema:

```
{
  displayName: String,
  Email: String,
  photoURL: String,
  uid: String,
}
```

Each user has unique user id given by "uuid" package. Document in user collection is created using unique user id assign to user. Each document has 4 field as shown in above schema. Each stores user related information. Display name stores Name of the user, email stores email id user authenticated with, photoURL store's location to user profile photo which is stored in firebase storage, and uid stores unique user id.

- userChats:

userChats document schema:

```
{
  UserId+ friendId: {
    {
      date: Timestamp,
      lastMessage: {
        text: String,
      },
      userInfo: {
        displayName: String,
        photoURL: String,
        Uid: String,
      }
    },
    .. .. .
  }
}
```

To keep track of user chat record, who are user friend, their last messages with timestamp, this database is used. Collections has documents with unique user id. whenever a user add friend to their chatting list two document for parties with their uid is created in userChats collection. Each document then follows above schema. Document has first root field for first user it added to chat list which is combination of user uid and friends uid. For each unique friend same procedure is followed. Root filed is nested with 3 more filed which are further nested. Date stores last message time stamp, lastMessage stores text field with last sent message and user info store friends info.

- chats:

chats document schema:

```
{
  messages: {
    {
      date: Timestamp,
      id: String,
      senderId: String,
      text: String,
    },
    .. .. .
  }
}
```

This collection stores actual messages between two users. Collection stores Document with unique id combined using user id and friend's id with whom user is chatting to. By this id user and friend both can access their conversation with each other. This document has root field message which is nested inside. It stores all the messages between user the collection belongs to. Each sub document has 4 field. date stores timestamp of message, id is key for each message, senderId distinguish message between two users. text stores actual message. All messages are stored in these formats.

CHPATER: 7

Project Model View

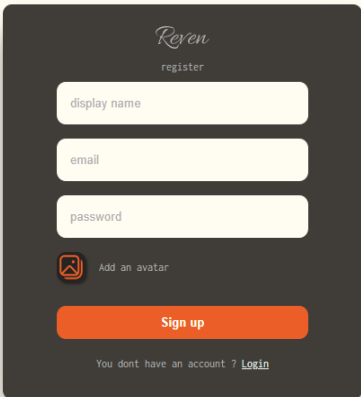
7.1 Login



The login form is a dark gray card centered on a light yellow background. It features the 'Reyen' logo at the top, followed by the word 'Login'. Below this are two white input fields labeled 'email' and 'password'. An orange 'Sign up' button is positioned below the password field. At the bottom, a link reads 'You have an account ? [Register](#)'.

- After registration the user has to login to the chat application using the registered email and password.

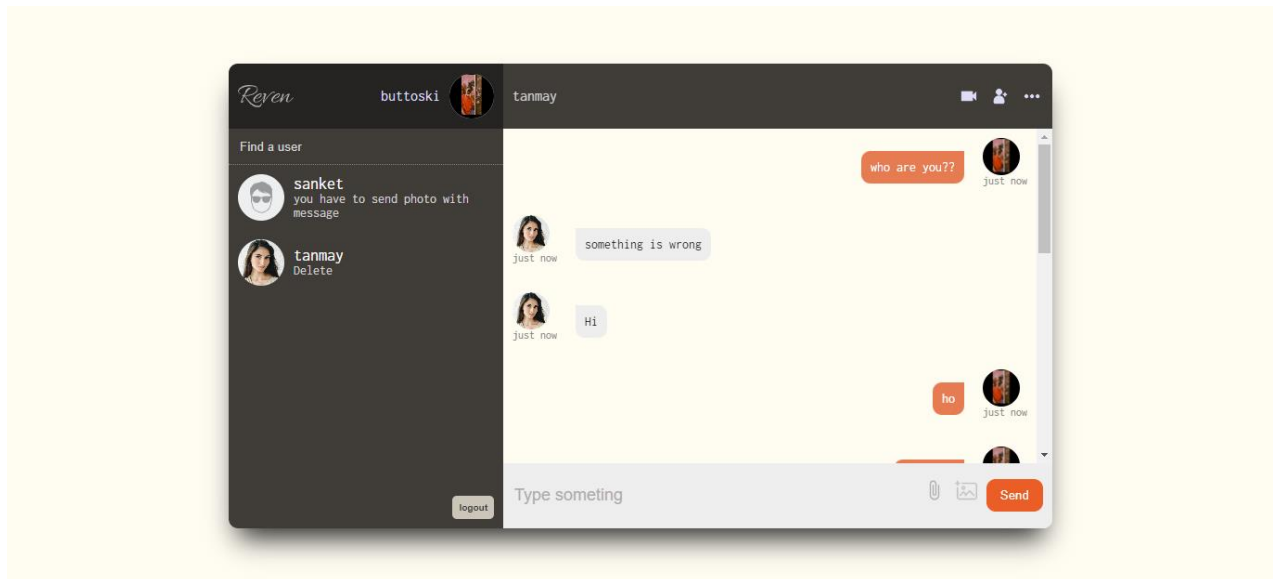
7.2 Sign up



The sign up form is a dark gray card centered on a light yellow background. It features the 'Reyen' logo at the top, followed by the word 'register'. Below this are three white input fields labeled 'display name', 'email', and 'password'. An 'Add an avatar' button with a camera icon is located below the password field. An orange 'Sign up' button is positioned below the avatar field. At the bottom, a link reads 'You dont have an account ? [Login](#)'.

- User must be able to sign up for the application using an Email, Display Name and Password.
- On Opening the application, user must be able to register themselves or they can directly login if there have an account already.
- The user's email will be the unique identifier of his/her account on Chat Application.

7.3 Home Page



- On the home page the user can chat with whoever is online on the application.
- User should be able to send instant message to any contact on his/her Chat Application contact list.
- User should be notified when message is successfully delivered to the recipient by coloring message.

CHPATER: 8

CONCLUSION

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