

Documentation

Realtime Messaging Prototype

Github: <https://github.com/nishchay2004>

Name: Nishchay Javara

Role: Software Development

Email: 21cs01051@iitbbs.ac.in

System Design Document

About

Chatify :This is message service prototype created using the MERN stack i.e. MongoDB, Express, React js and Node js

Features

- To sign up for the Chatify app, enter your Name, Email, and Password.
- It offers one-to-one real-time texting and the option to create groups with people who have already registered to the Chatify app.
- As long as the person has already registered for the app, you can connect with them by searching for their name or email.
- You can change the group name and add or remove participants from a group chat.
- It offers a real-time notification feature, so if you are using the app and a user sends you a message, a notification will appear in the upper right corner.
- By clicking on your profile photo in the upper right corner, you may also access your profile (name, email, and profile picture).

Setup and run app

1. Clone github repository using:

```
git clone https://github.com/nishchay2004/Chatify.git
```

2. In the backend make a ".env" file in which specify

```
MONGO_URI=mongodb+srv://nishchay2004:Nishumongodb%40123@cluster0.23rxgw  
v.mongodb.net/?retryWrites=true&w=majority  
JWT_SECRET=ANY_RANDOM_STRING
```

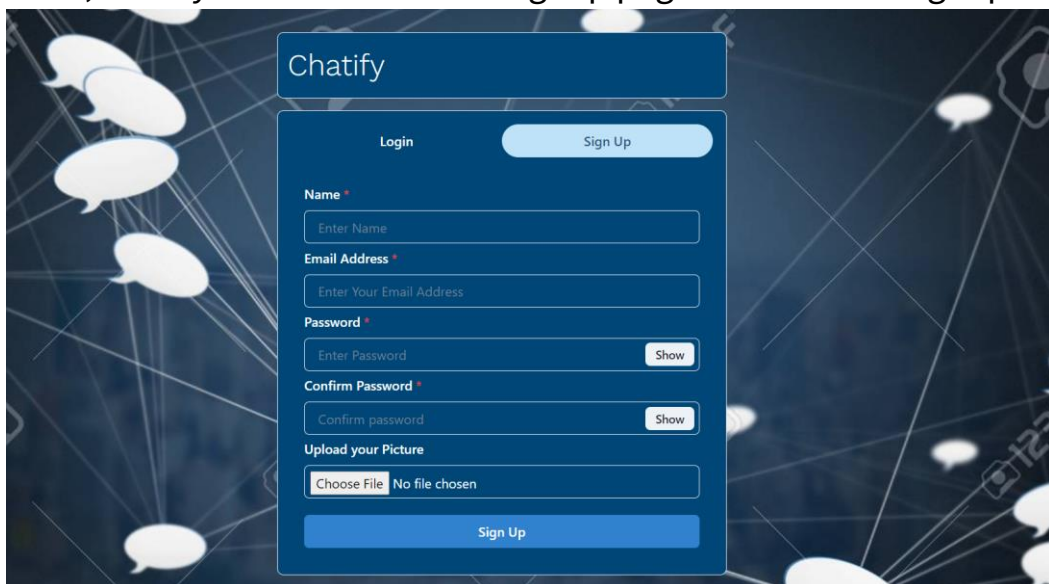
3. cd into the root folder and execute following command from terminal

```
cd backend
```

```
npm install
node server.js
cd ..
cd frontend
npm install --legacy-peer-deps
npm start
```

4. If not already opened, open localhost:3000 in your browser

5. Now, enter your credentials on Signup page and Click on Signup

A screenshot of the Chatify application's Signup page. The page has a dark blue background with a network of white nodes and lines. In the center is a white card with a dark blue header labeled 'Chatify'. Below the header are two buttons: 'Login' and 'Sign Up'. The 'Sign Up' button is highlighted. The form contains several input fields: 'Name *', 'Email Address *', 'Password *', and 'Confirm Password *'. Each field has a placeholder text. There are also 'Show' buttons for the password fields. Below these is an 'Upload your Picture' section with a 'Choose File' button and the text 'No file chosen'. At the bottom of the card is a large blue 'Sign Up' button.

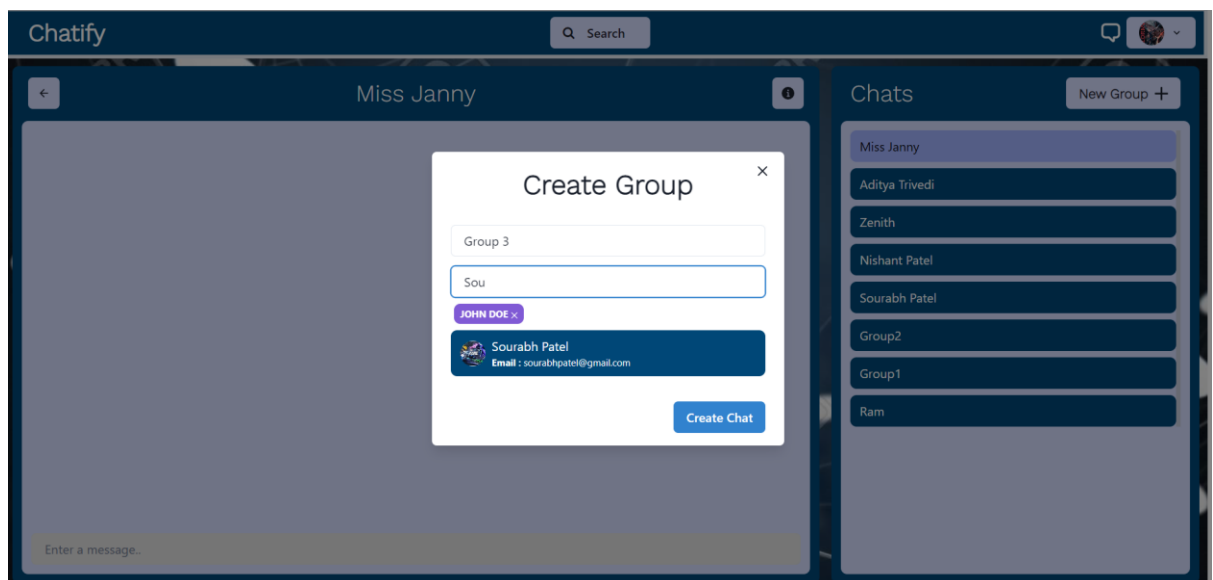
6. Refresh the page



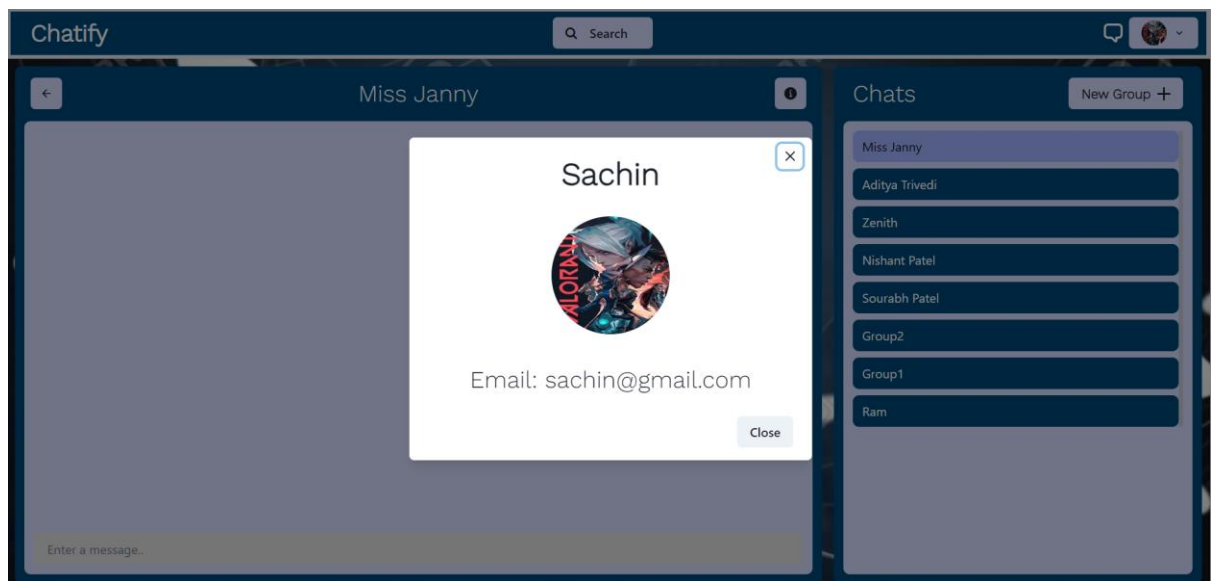
7. Open incognito and try to register with another user
8. Now search the user and create chat with him by clicking on it.



9. Now you can start realtime messaging service
- 10 You can also create group by clicking on New Group and members to it



11. You can view your profile by clicking on Upper Right Corner



Describing dependencies and libraries

```
npm i @chakra-ui/react @emotion/react @emotion/styled framer-motion
```

- Chakra UI is a modern component library for React. It comes with a set of reusable and composable React components that you may use to create front-end apps. Its power comes from its simplicity, modularity, and accessibility. This is the reason I used this library for styling my chat application

```
npm install react-router-dom@5
```

- Building single-page apps, or programmes with multiple pages or components but no page refresh since the content is dynamically downloaded from the URL, requires the use of React Router Dom. I need this package because it helps with a process known as routing that is made possible by React Router Dom.

```
npm i axios
```

- The backend can be contacted using this library. In our application, we request APIs using Axios. AJAX or fetch can also be used to accomplish the same things, but Axios has more flexibility and features, which makes it easier to build applications rapidly, which is why I used it.

```
npm i mongoose
```

- It is a popular Object Data Modeling (ODM) library for Node.js and MongoDB
- We can define data schemas for your MongoDB documents using Mongoose. This helps to guarantee the integrity and consistency of the data.
- Mongoose offers a robust and user-friendly API for creating and running complex queries. Instead of writing raw MongoDB queries, you may access and handle data in a way that is more developer-friendly by using methods **like search, findOne, and aggregate**.

```
npm i express-async-handler
```

- Without express-async-handler, our route handlers can be clogged with try-catch blocks to deal with asynchronous problems, which is why I utilised this package and for error handling. Use this package to get rid of those try-catch sections for code that is cleaner and easier to read.

```
npm i nodemon
```

- We use it during development phase so that, we do not have to stop and start the server again and again after changes.

```
npm i jsonwebtoken
```

- I used it for user authentication. It authorize user in backend.
- Consequently, when a user logs in and tries to access a resource that is made available to him. So, the user sends JWT to the backend, and the backend verifies the user.

```
npm i bcryptjs
```

- This package allowed me to create user passwords in encrypted form.

```
npm install socket.io
```

- This package is used to programme web sockets so that we may include the real-time messaging service feature.

```
npm install socket.io-client
```

- This is the client/frontend side package of socket.io for web socket programming

```
npm i @chakra-ui/icons
```

It imports the icons library from chakra-ui for styling-purpose.

Authentication

Signup

When user Signup to our app. A post request is sent to the server with credentials of user and database is populated according to the user schema.

```
const userSchema = mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true },
  password: { type: String, required: true },
  pic: {
    type: String,
    default:
      "https://www.google.com/url?sa=i&url=https%3A%2F%2Fpixabay.com%2Fvectors%2Fblank-profile-picture-mystery-man-973460%2F&psig=A0vVaw0evPp42bJh1_TQkiDu9_so&ust=1695408518222000&source=images&cd=vfe&opi=89978449&ved=0CBAQjRxqFwoTCMiWqpCvvIEDFQAAAAAdAAAAABAD"
  }
})
```

```

    },
    {
      timestamps: true
    }
  );

```

Login

When user logs in to our application by email and password.

A post request is sent and using findOne function we find the email of user and then we check password if both password matches, then user is logged in successfully.

```

const authUser = asyncHandler(async (req, res) => {
  const { email, password } = req.body;

  const user = await User.findOne({ email });

  if (user && (await user.matchPassword(password))) {
    res.json({
      _id: user._id,
      name: user.name,
      email: user.email,
      isAdmin: user.isAdmin,
      pic: user.pic,
      token: generateToken(user._id),
    });
  } else {
    res.status(401);
    throw new Error("Invalid Email or Password");
  }
});

```

Search

When we search for a user our browser will send a query to our database using a get request and we will be comparing all the users' name or email. The data object of users matching to our query will be sent by our database.

```

const allUsers = asyncHandler(async (req, res) => {
  const keyword = req.query.search
  ? {
    $or: [
      { name: { $regex: req.query.search, $options: "i" } },

```



```

        { email: { $regex: req.query.search, $options: "i" } },
      ],
    }
  : {});

  const users = await User.find(keyword).find({ _id: { $ne: req.user._id } });
  res.send(users);
});

```

Creating One to One Chat

When we click on a particular user its userId is sent using a post request to our server and it creates chat to that particular user.

```

var chatData = {
  chatName: "sender",
  isGroupChat: false,
  users: [req.user._id, userId],
};

try {
  const createdChat = await Chat.create(chatData);
  const FullChat = await Chat.findOne({ _id: createdChat._id
}).populate(
  "users",
  "-password"
);
  res.status(200).json(FullChat);
} catch (error) {
  res.status(400);
  throw new Error(error.message);
}
}

```

Here users has two id's i.e. userId which is selected by logged in user and the id of logged in user itself.

All data as per userSchema except password is populated into users object of chatModel.

```
const chatModel = mongoose.Schema({
  chatName: { type: String, trim: true },
  isGroupChat: { type: Boolean, default: false },
  users: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: "User",
    },
  ],
  latestMessage: {
    type: mongoose.Schema.Types.ObjectId,
    ref: "Message"
  },
  groupAdmin: [
    {
      type: mongoose.Schema.Types.ObjectId,
      ref: "User",
    },
  ],
}, {
  timestamps: true,
});
```

Creating Group Chat

When a user clicks on New Group Chat and select users, he sends a post requests to the server by providing GroupName and the userId's of users in stringify format. In backend, the users' data are parsed into a object users and later we also push the id of logged in user to that object. All the users id's are stored in user array of object as per chatModel. Also as per ChatModel/Schema we initialise isGroupChat to true and groupAdmin to current logged in user.

```

const createGroupChat = asyncHandler(async (req, res) => {
  if (!req.body.users || !req.body.name) {
    return res.status(400).send({ message: "Please Fill all the feilds" });
  }

  var users = JSON.parse(req.body.users);

  if (users.length < 2) {
    return res
      .status(400)
      .send("2 or less users cannot form group");
  }

  users.push(req.user);

  try {
    const groupChat = await Chat.create({
      chatName: req.body.name,
      users: users,
      isGroupChat: true,
      groupAdmin: req.user,
    });

    const fullGroupChat = await Chat.findOne({ _id: groupChat._id })
      .populate("users", "-password")
      .populate("groupAdmin", "-password");

    res.status(200).json(fullGroupChat);
  } catch (error) {

```

Send Message

A user sends a post request by giving chatId (of an individual or Group) and the content to our server. Server stores the given data as per messageSchema inside our mongodb database.

```

const messageModel = mongoose.Schema({
  sender:{type: mongoose.Schema.Types.ObjectId, ref:"User"},
  content:{type: String, trim: true},
  chat:{ type:mongoose.Schema.Types.ObjectId, ref:"Chat"},
},
{
  timestamps :true,
});

```

Here, content and chatId is provided by user while sender's id is fetched using req.user._id

Fetching Messages

To fetch all the messages from our database, we query our database with chatId.

```
const allMessages = asyncHandler(async (req, res) => {
  try {
    const messages = await Message.find({ chat: req.params.chatId })
      .populate("sender", "name pic email")
      .populate("chat");
    res.json(messages);
  } catch (error) {
    res.status(400);
    throw new Error(error.message);
  }
});
```

