React Native Chat App using Socket.io

First, create a React Native Project using React Native CLI and run it as usual.

Initializing Socket.io Backend:

Create a separate folder named "*socket.io-backend*" and in there, initialise npm by writing in terminal:

cd socket.io-backend
npm init --yes

Now, inside it, we've "*package.json*" file so we can install npm packages as required. Now, install "*socket.io*" package:

npm install socket.io

Then, add a file named "*server.ts*" in which we'll make the socket.io server. So, add the below code in it:

```
import {Server} from "socket.io";

const io = new Server({});

io.listen(3001);
```

Now, run this file using the command:

node server.ts
OR
nodemon server.ts

Now, in the terminal, we'll receive "*[nodemon] starting `ts-node server.ts`"* which means our server is now running.

Now, we'll add a connection event or listener so, in "*server.ts*", add:

```
io.on('connection', function () {
   console.log('User Connected!');
});
```

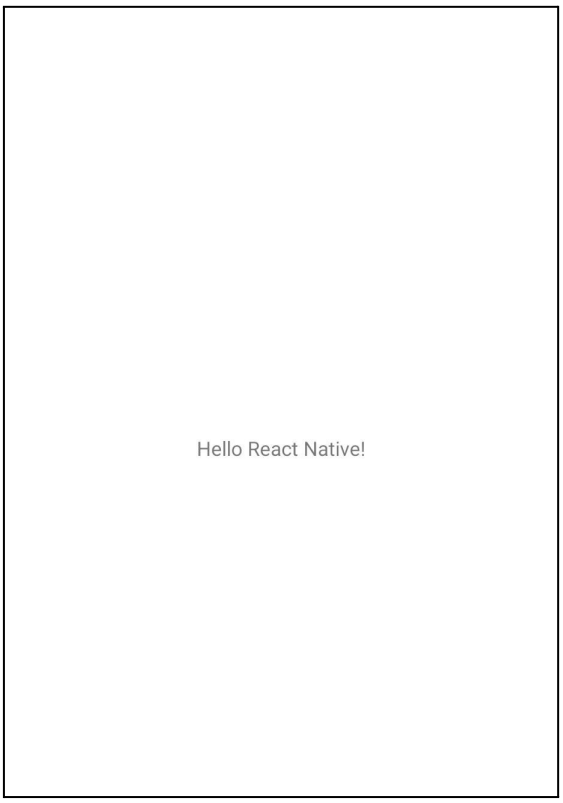
In this, the function will trigger when the app is connected to the server or connection happens.

Connecting to Socket.io from React Native:

Now, we'll add some functionality to React Native App so that it'll connect to the "*socket.io server*" that we've created above. So, first replace the default code of "*App.tsx*" to:

```
import React from 'react';
import {StyleSheet, Text, View} from 'react-native';

export default function App() {
  return (
    <View style={styles.container}>
      <Text>Hello React Native!</Text>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#fff',
  },
});
```

App will look like this:



Now, install "*socket.io-client*" dependency in the main app directory:

npm install socket.io-client

This is used to connect the "*socket.io server*" with the app. Now, in "*App.tsx*" file, first import "*io*" from above dependency and add a "*useEffect()*" hook:

```
import React, {useEffect} from 'react';
import io from 'socket.io-client';

export default function App() {
  useEffect(() => {
    io('http://192.168.1.4:3001');
  }, []);
```

"*UseEffect*" will connect the server to the app when it loads to the provided server link i.e. "*http://192.168.1.4:3001*". "*3001*" is the port we've provided in the "*server.ts*" file. "*192.168.1.4*" is the IP of the machine. To find the IP, in the terminal, write:

ipconfig /all

Here, we'll get the IP of the machine. Now, when we run the app, in the console, we get:

[nodemon] starting `ts-node server.ts`
User Connected!

This means the function is triggered when the app loads, hence, the app and the server are now connected.

Now, I've changed the structure of the project by making a file named "*ChatScreen.tsx*" inside the "*ChatScreen*" folder inside the "*screens*" folder. Then, I've moved the code in the "*App.tsx*" file to the "*ChatScreen.tsx*" file and imported it in the "*App.tsx*" file.

Adding TextInput and saving entered message:

Now, we'll add a TextInput in which we'll type a message and save it using the "*useState()*" hook so that we'll use the entered message to send it to the server. So, in "*ChatScreen.tsx*" file, add:

```
import React, {useEffect, useState} from 'react';
import {View, Text, TextInput} from 'react-native';
import {styles} from '../../styles/styles';

export default function ChatScreen() {
  const [messageToSend, setMessageToSend] =
useState('');

  useEffect(() => {
```

```
    io('http://192.168.1.4:3001');
  }, []);

  return (
    <View style={styles.mainContainer}>
      <Text style={styles.textStyle}>Hello React
Native!</Text>
      <TextInput
        value={messageToSend}
        placeholder="Enter chat message..."
        placeholderTextColor="#999"
        onChangeText={(text: any) =>
setMessageToSend(text)}
        style={styles.textInputStyle}
      />
    </View>
  );
}
```

Then, in "*styles.ts*" file, add:

```
import {StyleSheet} from 'react-native';

export const styles = StyleSheet.create({
  mainContainer: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#fff',
  },
  textStyle: {
    color: '#000',
```

```
    fontSize: 16,
    fontWeight: '700',
  },
  textInputStyle: {
    color: '#000',
  },
});
```

Now, the app will look like this:

Sending Chat Message to Socket.io Backend from React Native:

Now, we'll send the entered message in TextInput to the "*socket.io server*" from the React Native App. So, we can use a TextInput prop called "*onSubmitEditing*" which is called when we click on the submit button. So, add it in the TextInput:

```
<TextInput onSubmitEditing={sendMessageToServer} />
```

Now, create the function "*sendMessageToServer*" which will send the message to the server. So we need to use the "*socket*" that "*io*" has created. So, the "*useEffect*" is returning a socket client which can be used to emit some events like this:

```
useEffect(() => {
    const socket = io('http://192.168.1.4:3001');
    socket.emit('message', 'hello world!');
  }, []);
```

"*socket.emit()*" is used to emit a message "*hello world!*" to either side of the connection. But, we can't access this "*socket*" variable outside the "*useEffect*" hook so, we'll use "*useRef*" hook. So add:

```
import {useRef} from 'react';
  const socket = useRef<any>(null);

  useEffect(() => {
    socket.current = io('http://192.168.1.4:3001');
  }, []);

  function sendMessageToServer() {
    socket.current.emit('message', messageToSend);
    setMessageToSend('');
  }
```

Now, "*sendMessageToServer()*" will send the entered message to the server by emitting the "*message*" event and passing the entered message as the data. "*setMessageToSend('')*" will clear the message after sending it to the server. Now, to listen to the message, we need to set up an event which will listen to the message that we're emitting from the client in the "*server.ts*" file in the "*server.io-backend*" folder. So, add in "*server.ts*" file:

```
io.on('connection', socket => {
```

```
    console.log('User Connected!');
    socket.on("message", message => {
        console.log(message);
    });
});
```

Here, "*socket*" is the socket connection to the client and "*socket.on*" is used to create an event which will listen to the "*message*" event created in "*ChatScreen.tsx*" file and the callback function will return the message that we've sent to the server. Now, if we check the server console, we can see the entered message in the server:

[nodemon] starting `ts-node server.ts`
User Connected!
Hello from client

Now, we're able to send messages to the server.

Receiving Chat Message from Socket.io Backend to React Native:

Now, we'll send a message from the socket.io backend to our app. So, we'll use the method "*io.emit()*" which is used to emit messages to all the connected clients including the sender. So, in "*server.ts*" file, add after receiving the message from client:

```
io.emit("message", message);
```

Now, in the "*ChatScreen.tsx*" we'll add a listener which will listen to the received messages from the server just like we did in "*server.ts*" for messages from the client. But first, we need to add a "*useState()*" for storing all the chat messages so, first add:

```
const [recvMessagesFromServer,
setRecvMessagesFromServer] = useState<any>([]);
```

Then, add the listener in the "*useEffect()*" hook:

```
useEffect(() => {
    socket.current = io('http://192.168.1.4:3001');
    socket.current.on('message', (message: any) => {
      setRecvMessagesFromServer((prevState: any) =>
[...prevState, message]);
    });
  }, []);
```

"*192.168.1.4*" is the IP address of the device which can be found using the command:

ipconfig /all

"*socket.current.on()*" will listen to any message received from the server and update the state using "*setRecvMessagesFromServer()*" function. Now, we need to display these messages so add:

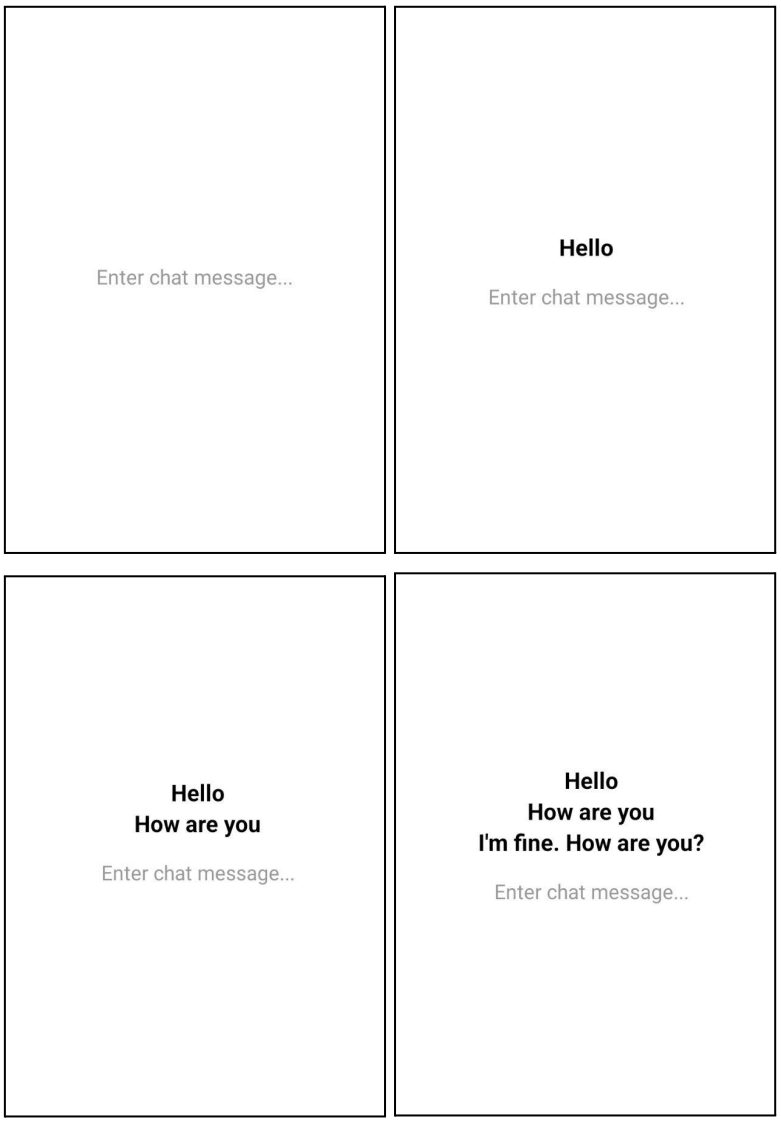
```
const displayChatMessages =
recvMessagesFromServer.map((msg: any) => (
    <Text style={styles.textStyle} key={msg}>
      {msg}
    </Text>
  ));
```

Now, it will return the received messages as a text component. Now, use it to display the messages in the app:

```
return (
    <View style={styles.mainContainer}>
      {displayChatMessages}
      <TextInput
        value={messageToSend}
```

```
        placeholder="Enter chat message..."
        placeholderTextColor="#999"
        onChangeText={(text: any) =>
setMessageToSend(text)}
        style={styles.textInputStyle}
        onSubmitEditing={sendMessageToServer}
      />
    </View>
  );
```

Now, run the app and when we write a text in the TextInput and press enter or submit button, we'll see the message on top of the TextInput.



Improving the UI:

Now, the basic functionality is done, so we can improve the UI of the app. For that, we'll use a library called "*react-native-gifted-chat*" which is used to display chat messages effectively. "*react-native-safe-area-context*" is required to effectively run the gifted chat UI. So, install it:

npm i react-native-gifted-chat
npm i react-native-safe-area-context

Now, use this in "*ChatScreen.tsx*" file:

```
import {GiftedChat} from 'react-native-gifted-chat';

export default function ChatScreen() {
  const [recvMessagesFromServer,
setRecvMessagesFromServer] = useState<any>([]);

  useEffect(() => {
    setRecvMessagesFromServer([
      {
        _id: 1,
        text: 'Hello Developer',
        createdAt: new Date(),
        user: {
          _id: 2,
          name: 'React Native',
          avatar:
'https://placeimg.com/140/140/any',
        },
      },
    ]);
  }, []);
```

```
  return <GiftedChat
messages={recvMessagesFromServer} user={{_id: 1}}
/>;
}
```

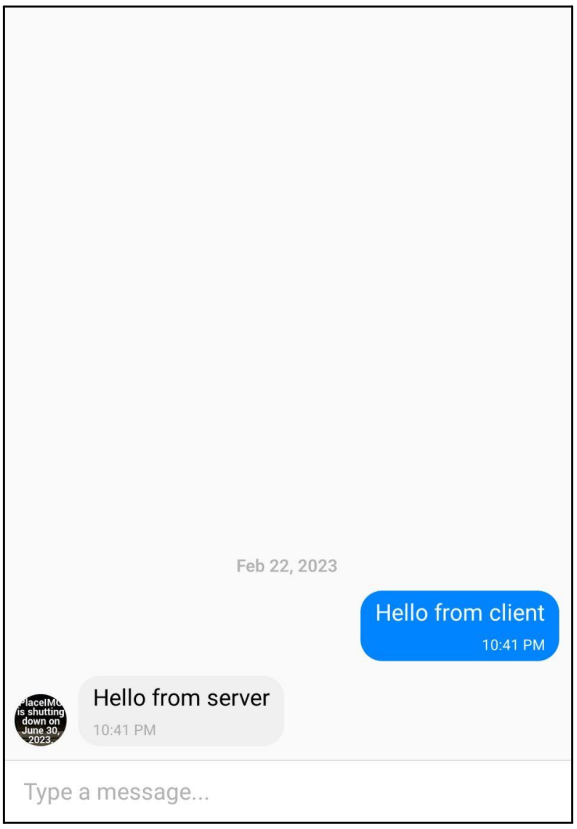
In "*setRecvMessagesFromServer()*" function, we're setting up a static message with text, created at, user name and user avatar. Then, in the "*GiftedChat*" component, we're displaying that message. Now, the app will look like this:

Displaying received messages from Socket.io in Gifted Chat UI:

Now, we need to display the messages from the server to the added UI. Now, in above "*GiftedChat*" component, "*user={{_id: 1}}*" means the sender id is 1, so if we add another object in the array in

“*setRecvMessagesFromServer()*” function, we’ll get a blue bubble message which indicates that the sender had written that message:

```
{
        _id: 2,
        text: 'Hello from client',
        createdAt: new Date(),
        user: {
          _id: 1,
          name: 'React Native',
          avatar:
'https://placeimg.com/140/140/any',
        },
      },
```



Now, in Gifted Chat UI, we’ve a method called “*onSend()*” which is used to send messages. So add it in the component:

```
onSend={messages => sendMessageToServer(messages)}
```

Now, we've also created a function "*sendMessageToServer()*" which will send messages to the server from the client. So, modify it to:

```
function sendMessageToServer(messages: any) {
    console.log(messages);
  }
```

Now, when we write a message in the app, in the console, we get:

[{"_id": "abcca0da-149e-4d0c-ad22-74adb2dae6fa", "createdAt": 2023-02-22T17:26:22.649Z, "text": "Hello", "user": {"_id": 1}}]

We get message id, created at, text, and user object with user id in it. It is similar to what we're adding in the "*setRecvMessagesFromServer()*" function in "*useEffect()*" hook. Now, we need to send this message to the server and receive a message back. So, update the "*sendMessageToServer()*" to:

```
function sendMessageToServer(messages: any) {
    socket.current.emit('message',
messages[0].text);
  }
```

In this, we're just sending the "*text*" value of the first element of the array that we're getting in the console. Now, if we send a message, we get an error that:

Cannot read property '_id' of undefined

This is because we do not have an "*user_id*" when setting it in the "*useEffect()*" hook inside "*setRecvMessagesFromServer()*". We're just setting a text string only that we're getting from the server.

```
socket.current.on('message', (message: any) => {
      setRecvMessagesFromServer((prevState: any) =>
[...prevState, message]);
    });
```

So, we need to add the "*user_id*", so add in the event listener in "*useEffect()*" hook:

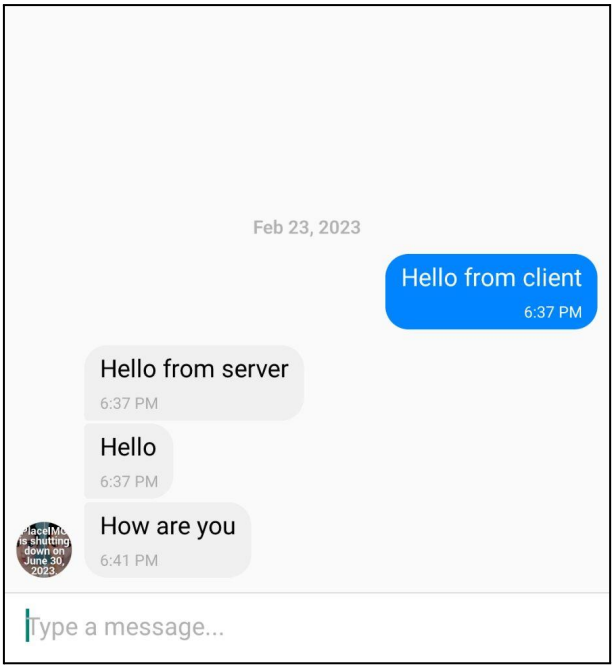
```
socket.current.on('message', (message: any) => {
      const testMessage = [{
        _id: 3,
        text: 'Hello from server',
        createdAt: new Date(),
        user: {
          _id: 2,
          name: 'React Native',
          avatar:
'https://placeimg.com/140/140/any',
        },
      }];
      testMessage[0].text = message;
      setRecvMessagesFromServer((prevState: any) =>
[
        ...prevState,
        testMessage,
      ]);
    });
```

Now, when we write a message, we get:

Now, the entered message is not showing properly i.e. showing on top of other messages, so we'll use a method called "*append()*" of "*GiftedChat*" component which is used to add messages and will add messages at the bottom of other messages so, in the "*useEffect()*" hook, set the state i.e. "*setRecvMessagesFromServer()*" to:

```
setRecvMessagesFromServer((prevState: any) =>
        GiftedChat.append(prevState, testMessage),
      );
```

Now, when we type a message, it'll shown at the bottom:



Avoiding Security issues and mapping userIds:

Now, we can send messages but there's an issue of "*user_id*" which will be done from the backend even though we're sending messages from the client. So we could do something like this:

```
function sendMessage(messages: any) {
    socket.current.emit('message', messages);
  }
```

In this, we're sending the whole message object to "*socket.io server*" and it could send the whole object to their clients. This approach is not good as anyone can send bogus messages to other clients even if they are not a part of the connection. So, someone could create a message object with a different "*user_id*" than they actually have. So, for other people, it will look like that other user has sent that message.

So we can't trust what the client is sending and who the client is based on the above scene.So, the idea is **we've different sockets connected to "*socket.io server*" and each of the sockets has a unique "*socket_id*", which is only known between the server and the client**. So, clients don't know about each other's "*socket_ids*". So, we can create a directory of which "*socket_id*" is connected to which "*user_id*" on the backend.

For example, when "*user_id = 1*" or its corresponding socket connects to the server and sends a message, the server will sense the message object sent by the client and set the "*user_id*" based on what we have inside the directory. So, what we want to get from the client is the "*text*" message and not its "*user_id*" so that the "*user_id*" could not be shared with other clients.

Another thing is, we've a blue bubble in "*GiftedChat*" as "*user_id = 1*" i.e. "*user={{_id: 1}}*".

```
<GiftedChat
        messages={recvMessages}
```

```
        user={{_id: 1}}
        onSend={messages => sendMessage(messages)}
      />
```

So, when we send messages, the first message object in the array always has "*user_id = 1*". So other clients will also see it as a blue bubble or they've written it themselves. So, what we can do is append a new message if you're sending a message from a client having "*user_id = 1*". Then the server will send this message as "*user_id = 2*". So, other clients will see it as a grey bubble instead of blue bubble. So, it means the actual "*user_ids*" will start from 2 and so on but we'll append the messages in the array starting from "*user_id = 1*".

Refactoring Code:

Now, first of all, remove the dummy text message from the "*ChatScreen.tsx*" file inside the "*socket.current.on()*" function. It'll look like this:

```
socket.current.on('message', (message: any) => {
      setRecvMessagesFromServer((prevState: any) =>
        GiftedChat.append(prevState, message),
      );
    });
```

So, we're receiving the whole object and appending that to GiftedChat. Next, remove the dummy data inside "*setRecvMessagesFromServer()*" function. The "*useEffect()*" will look like this:

```
useEffect(() => {
    socket.current =
io('http://192.168.0.109:3001');
    socket.current.on('message', (message: any) => {
      setRecvMessages((prevState: any) =>
        GiftedChat.append(prevState, message),
```

```
      );
    });
  }, []);
```

Implementing userIds and sending full message object from the server:

Now, we need to send full message object with "*user_id = 1*" instead of just the text to the server so that the client can see a blue bubble when sending a message so, update "*sendMessageToServer()*" function to:

```
function sendMessageToServer(messages: any) {
    socket.current.emit('message',
messages[0].text);
    setRecvMessagesFromServer((prevState: IMessage[]
| undefined) =>
      GiftedChat.append(prevState, messages),
    );
  }
```

Now, when we send a message to the server as a client, we see a blue bubble with the text in it.

Now we have different "*socket_id*" for different clients so if we write inside "*io.on()*" function:

```
console.log(socket.id);
```

We get "*socket_id*" of different clients connected to the server:

User Connected!
YRJSB1t6B4Wm_9yrAAAC
User Connected!
7YebVt6wBDCYaXtYAAAD
User Connected!
MvxsbxzL7h4gLO7RAAAF

Now, we need to create a directory of different "*user_id*" and map them to their respective "*socket_id*" so add:

```
let currentUserId = 2;
const userIds = {};

io.on("connection", (socket) => {
  console.log("User Connected!");
  ...
  userIds[socket.id] = currentUserId++;
  ...
});
```

"*currentUserId*" must start with 2 as discussed above and "*userIds*" is the directory in which we store all "*user_id*" with respective "*socket_id*" and we're adding them after every socket connection and incrementing "*user_id*" by 1 after each socket connection.

Now, we need to send the message back when we receive a message on the server from the client. So, instead of emitting the message, we can use a method called "*broadcast()*". This method is used to send messages to all the clients "***except the sender.***" So, we can write inside "*socket.on()*" function:

```
socket.on("message", (textMessage) => {
    console.log(textMessage);
    socket.broadcast.emit("message", textMessage);
  });
```

Now, we're sending just the text message but we need to send the whole message object so create a function which will create this message object and a variable for assigning the message ids which will be unique for each message:

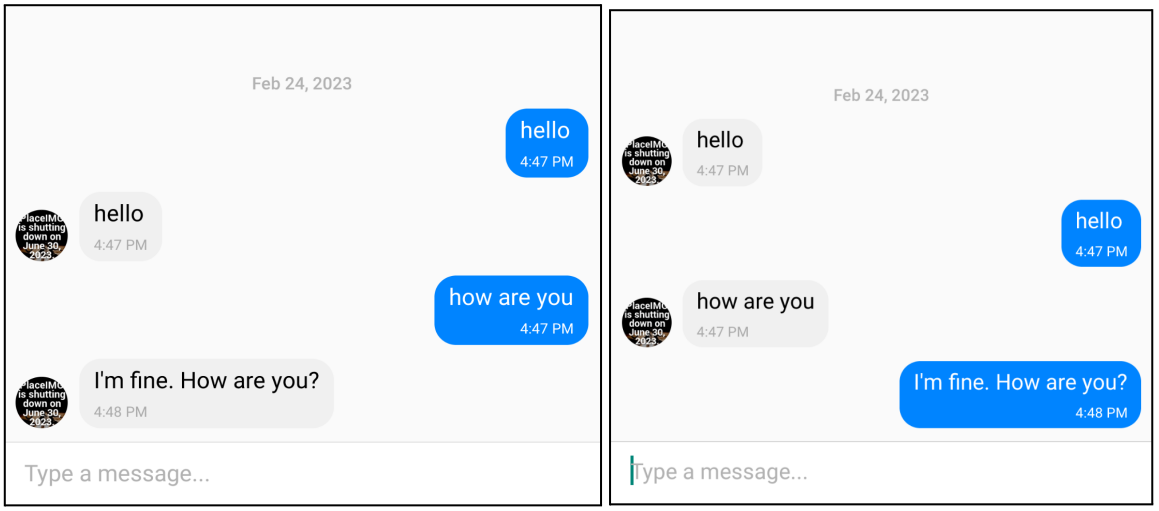
```
let currentMessageId = 1;

function createMessage(userId: any, textMessage: any)
{
  return {
    _id: currentMessageId++,
    text: textMessage,
    createdAt: new Date(),
    user: {
      _id: userId,
      name: "Test User",
      avatar: "https://placeimg.com/140/140/any",
    },
  };
}
```

Then, call it inside "*io.on*" method:

```
io.on("connection", (socket) => {
  ...
  socket.on("message", (textMessage) => {
    const userId = userIds[socket.id];
    const message =
createMessage(userId,textMessage);
    console.log(message);
    socket.broadcast.emit("message", message);
  });
});
```

"*userId*" is used to get the user id hence we're not dependent on the client to send the "*user_id*" which resolves a major security issue. The "*message*" will contain the message object. Now, when we send a

message, we'll see a blue bubble on the client side and a grey bubble on the receiver side.



Now, our basic public chat room is created and anyone can connect to socket.io server and start chatting. But we have not implemented "*user_name*" so we don't know who is writing the messages.

Creating Separate Message Event Handler:

Now, we want to separate the message event handler so that the code will look a lot cleaner. So, create a folder inside "*socket.io-backend*" and name it as "*handlers*" and inside it, create a file named "*message.handler.ts*" and add:

```
let currentMessageId = 1;

function createMessage(userId: any, textMessage: any)
{
  return {
    _id: currentMessageId++,
    text: textMessage,
    createdAt: new Date(),
    user: {
      _id: userId,
```

```
      name: "Test User",
      avatar: "https://placeimg.com/140/140/any",
    },
  };
}

export default function handleMessage(socket: any,
userIds: any) {
  socket.on("message", (textMessage: any) => {
    const userId = userIds[socket.id];
    const message = createMessage(userId,
textMessage);
    console.log(message);
    socket.broadcast.emit("message", message);
  });
}
```

Then update "*server.ts*" file to:

```
import { Server } from "socket.io";
import handleMessage from
"./handlers/message.handler";

const io = new Server({});

let currentUserId = 2;
const userIds: any = {};

io.on("connection", (socket) => {
  console.log("User Connected!");
  console.log(socket.id);
  userIds[socket.id] = currentUserId++;
```

```
  handleMessage(socket, userIds);
});

io.listen(3001);
```

Adding usernames to Socket.io Backend:

Now, we'll add the usernames to the server. So, in "*server.ts*" file, inside "*io.on()*" function, add:

```
const users: any = {};

io.on("connection", (socket) => {
  ...
  users[socket.id] = { userId: currentUserId++ };
  socket.on("join", (username) => {
    users[socket.id].username = username;
  });
  handleMessage(socket, users);
});
```

Here, first we've changed the directory name as "*users*". Since we're storing more than 1 value in it, we're storing them in the form of objects. Now, we also need to change the "*messageHandler*" because we're sending "*users*" as an object. So update "*handleMessage()*" to:

```
export default function handleMessage(socket: any,
users: any) {
  socket.on("message", (textMessage: any) => {
    const user = users[socket.id];
    const message = createMessage(user,
textMessage);
    ...
  });
```

```
}
```

Now, update "*createMessage()*" function to:

```
function createMessage(user: any, textMessage: any) {
    ...
    user: {
      _id: user.userId,
      name: user.username,
      ...
}
```

Now, we've also added username functionality to the socket.io backend.

Creating Join Screen:

Now, we want to update our UI so that the user first enters a username and then they can chat. First, in the "*server.ts*" file, move the "*handleMessage()*" handler inside the "*join*" handler so that the messages can be sent only when someone joins the chat.

```
socket.on("join", (username) => {
    users[socket.id].username = username;
    handleMessage(socket, users);
  });
```

This type of approach means we can execute another handler only when the parent handler executes so we can't send messages without joining in the chat room.

Now, create a file named "*JoinScreen.tsx*" inside the "*JoinScreen*" folder in which we will take the user's username and then he/she can join the chat room. Add in it:

```
import React from 'react';
```

```
import {View, Text} from 'react-native';

export default function JoinScreen() {
  return (
    <View>
      <Text>Join Screen</Text>
    </View>
  );
}
```

Then import it in "*ChatScreen.tsx*":

```
import JoinScreen from '../JoinScreen/JoinScreen';
```

Then, create a "*useState*" for checking if the user has joined the chat room or not:

```
const [isJoined, setIsJoined] = useState(false);
```

Then, in "*return*" statement, add:

```
return (
    <>
      {isJoined ? (
        <GiftedChat
          messages={recvMessagesFromServer}
          user={{_id: 1}}
          onSend={messages =>
sendMessageToServer(messages)}
        />
      ) : (
        <JoinScreen />
      )}
```

```
    </>
  );
```

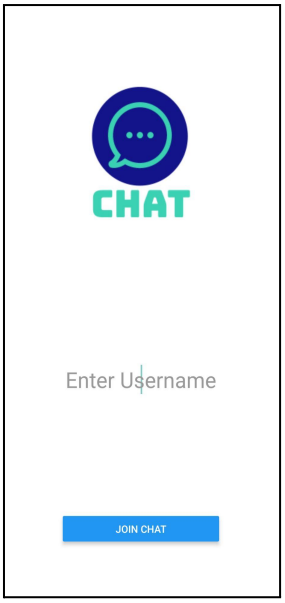
Now, based on whether the user has joined the chat or not, we're displaying respective screens. Now, update the "*JoinScreen.tsx*" file to:

```
import React from 'react';
import {View, TextInput, Image, Button} from
'react-native';
import styles from '../../styles/style';

export default function JoinScreen() {
  return (
    <View style={styles.container}>
      <Image

source={require('../../assets/chat-icon.png')}
        style={styles.imageStyle}
        resizeMode="contain"
      />
      <View style={styles.inputContainerStyle}>
        <TextInput
          placeholder="Enter Username"
          placeholderTextColor="#999"
          style={styles.textInputStyle}
        />
        <Button title="Join Chat" />
      </View>
    </View>
  );
}
```

And "*style.ts*" file to:

```
import {StyleSheet} from 'react-native';

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#FFF',
  },
  inputContainerStyle: {flex: 1, justifyContent:
'space-around'},
  textInputStyle: {
    color: '#000',
    fontSize: 30,
    textAlign: 'center',
  },
  imageStyle: {
    flex: 1,
  },
});

export default styles;
```

Now, the “*JoinScreen*” will look like this:



Creating JoinChat function and joining chat with usernames:

Now, after typing the username, we must be able to join chat room so we need to create a function inside "*ChatScreen.tsx*" file so add:

```
function joinChat(username: any) {
    socket.current.emit('join', username);
    setIsJoined(true);
  }
```

It'll emit the "*join*" handler and after that "*isJoined*" is set to true so that we see the "*GiftedChat*" component. Then, pass this to "*JoinScreen*" component and in "*JoinScreen.tsx*" file, first create a "*useState*" which will store the typed username:

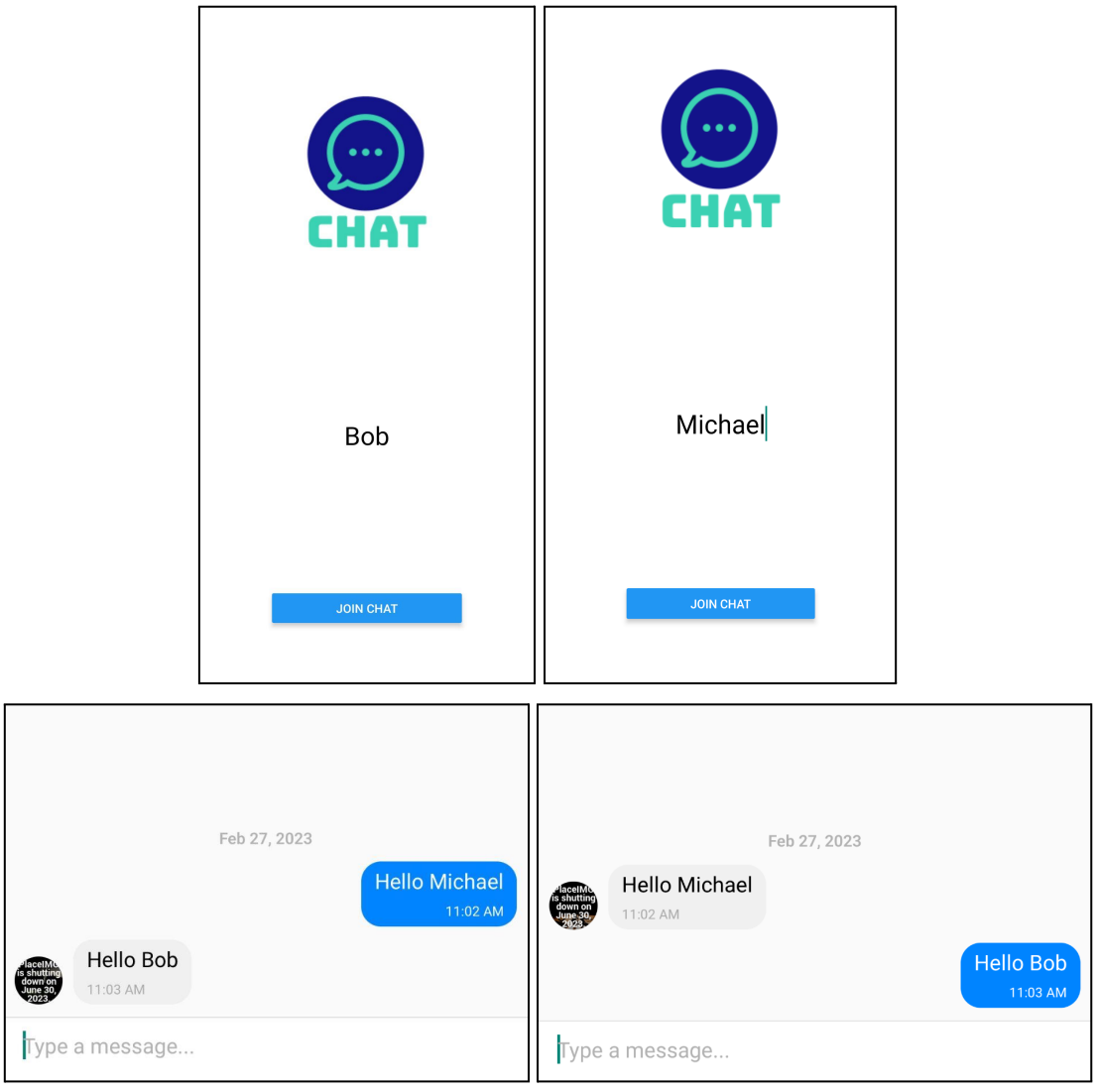
```
const [username, setUsername] = useState('');
```

Then, modify the "*TextInput*" and call the "*joinChat()*" function inside the "*Button*" component:

```
<TextInput
    value={username}
```

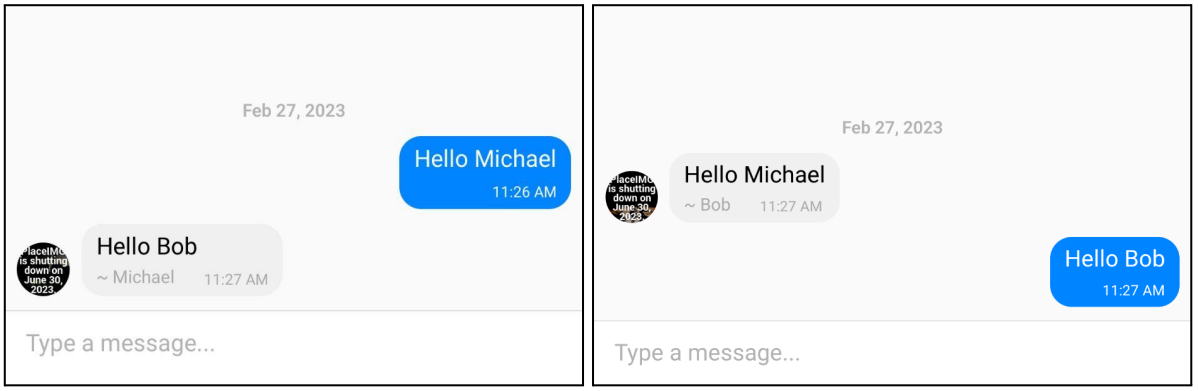
```
    placeholder="Enter Username"
    placeholderTextColor="#999"
    style={styles.textInputStyle}
    onChangeText={text => setUsername(text)} />
<Button title="Join Chat" onPress={() =>
joinChat(username)} />
```

Now, when we enter a username, we see our "*GiftedChat*" component, and we can now chat with other users.



But, we're not getting the usernames in "*GiftedChat*". So there's a property of "*GiftedChat*" called "*renderUsernameOnMessage*" which will

render the username on the message. Now, if we write the message again we see the username on the grey message bubble:



Creating unique avatars for different users:

Now, we have same avatar for all the users, so we need to display different avatars for different users so in "*server.ts*" file, create a function:

```
function createUserAvatarUrl() {
  const rand1 = Math.round(Math.random() * 200 +
100);
  const rand2 = Math.round(Math.random() * 200 +
100);
  return `https://picsum.photos/${rand1}/${rand2}`;
}
```

"*rand1*" and "*rand2*" are for image "*width*" and "*height*" respectively. Then it'll return the image url with random width and height. Now, when user joins the chat, set the avatar to the directory:

```
socket.on("join", (username) => {
    users[socket.id].username = username;
    users[socket.id].avatar = createUserAvatarUrl();
    handleMessage(socket, users);
  });
```

Then inside “*message.handler.ts*” file, inside “*createMessage()*” function, set the avatar using “*user*” object:

```
function createMessage(user: any, textMessage: any) {
  return {
    ...
    user: {
      ...
      avatar: user.avatar,
    },
  };
}
```

Now, when we chat , we have different avatars for different users.

Adding Navigations:

Now, we need to display a list of different users in the chat and then we can tap on one of the users and chat with them in private. So, we need to add some navigations for that. So add some navigation dependencies inside ChatApp folder:

npm install @react-navigation/native react-native-screens react-native-safe-area-context @react-navigation/stack react-native-gesture-handler

Then, create a folder named “*navigations*” and inside it, create a file named “*StackNavigator.tsx*” where we create our stack navigation. Then create another file named “*RootNavigator.tsx*” where we use the stack navigation. First, in “*StackNavigator.tsx*” add:

```
import React from 'react';
import {createStackNavigator} from
'@react-navigation/stack';
import JoinScreen from
'../screens/JoinScreen/JoinScreen';
```

```
import ChatScreen from
'../screens/ChatScreen/ChatScreen';

const Stack = createStackNavigator();

export default function StackNavigator() {
  return (
    <Stack.Navigator
      screenOptions={{headerShown: false}}
      initialRouteName="JoinScreen">
      <Stack.Screen name="JoinScreen"
component={JoinScreen} />
      <Stack.Screen name="ChatScreen"
component={ChatScreen} />
    </Stack.Navigator>
  );
}
```

Then, in "*RootNavigator.tsx*" file, add:

```
import React from 'react';
import {NavigationContainer} from
'@react-navigation/native';
import StackNavigator from './StackNavigator';

export default function RootNavigator() {
  return (
    <NavigationContainer>
      <StackNavigator />
    </NavigationContainer>
  );
}
```

Then, in the "*ChatSceen.tsx*" file, only return the "*GiftedChat*" component and remove the condition for the "*isJoined*" variable. Lastly, in the "*App.tsx*" file, return "*RootNavigator*" instead of "*ChatScreen*".

Adding Redux:

Now, the navigation is added but for sending socket.io requests and other state management things, we'll use "***Redux***" for state management and other requests. So install some dependencies for it:

npm install redux react-redux redux-socket.io

"*redux-socket.io*" is used to connect redux and socket.io server.

Create a folder named "*redux*" and then create a file named "*action.ts*". In there create an action which will send "*data*" to server:

```
export const action = {type: 'server/hello', data:
'Hello!'};
```

Then, create a new file named "*reducer.ts*" which will take care of updating the state of the app. Add:

```
function reducer(state = {}, action: any) {
  switch (action.type) {
    case 'message':
      return {...state, message: action.data};
    default:
      return state;
  }
}

export default reducer;
```

Reducer will take an initial empty object and an action which is used to emit events or change the state of the app. So, in the "*message*" case, we've returned the initial state with a message property which has the action's data.

Then, create another file named "*store.ts*" which will store the state of the app and add:

```
import {createStore, applyMiddleware} from 'redux';
import createSocketIOMiddleWare from
'redux-socket.io';
import io from 'socket.io-client';
import reducer from './reducer';

const socket = io('http://192.168.0.130:3002');
const socketIOMiddleWare =
createSocketIOMiddleWare(socket, 'server/');

export const store =
applyMiddleware(socketIOMiddleWare)(createStore)(red
ucer);
```

In this, we've created a socket and then passed it to "*socketIOMiddleWare*" and we've also passed a prefix "*server/*" which is used everytime we want to emit a socket.io event. Then, we've created a store by passing middleware and reducer which will store all the states of the app.

Then, in "*App.tsx*" file, add:

```
...
import {action} from './source/redux/action';
import {store} from './source/redux/store';
```

```
export default function App() {
  store.subscribe(() => {
    console.log('message', store.getState());
  });
  store.dispatch(action);
...
```

"*store.subscribe()*" is used to notify every time when the state changes. "*store.getState()*" to get the updated state of the app. Then we've used "*store.dispatch()*" to emit a server event i.e. the "*action*" that we've created above which will send data to the server. So when the app starts, we're sending a "*Hello!*" message to the server.

Now, we're sending an event "*server/hello*" and also we've a prefix "*server/*" which means every time we dispatch an action that starts with "*server/*", it'll go through "*socketIOMiddleWare*" and emit this as an event to the server. So, in "*server.ts*" file, add another event for listening to "*action*" or "*server/*" event after "*join*"event:

```
socket.on("action", (action) => {
    switch (action.type) {
      case "server/hello":
        console.log("Got hello event", action.data);
        socket.emit("action", { type: "message",
data: "Good day!" });
    } });
```

In here, we're checking for "*server/hello*" case which we're receiving from the action and after that, we're printing the "*data*" action is sending and emitting an action with type "*message*" which will trigger the reducer and update the state with the "*data: "Good day!"*". So, in client side console, we get:

message {"message": "Good day!"}

which we're getting from the server. And in server side console, we get:

Got hello event Hello!

which we're getting from the client. So, the basic idea is, the client dispatches an action to the server with a type and data. Server will check for the right case and print the action data. Then the server emits or dispatches an action to the client, client checks for the right case in the reducer and updates the state and prints the updated state.

Dispatching JoinChat Event from JoinScreen using Redux:

Now, we need to dispatch a socket.io event i.e. "*JoinChat*" event using Redux, so in "*JoinScreen.tsx*" file, add:

```
...
import {useDispatch} from 'react-redux';

export default function JoinScreen() {
  ...
  const dispatch: any = useDispatch();

  return (
    ...
        <Button
          title="Join Chat"
          onPress={() => dispatch({type:
'server/join', data: username})}
        />
      ...
  );
}
```

“*useDispatch()*” is used to dispatch actions from any point in the app. So, we’re dispatching a server join event with “*username*” as the data. Now, we need to wrap our app inside a “*Provider*” so in “*index.js*” file, add:

```
...
import {Provider} from 'react-redux';
import {store} from './source/redux/store';
...

const ReduxProvider = () => {
  return (
    <Provider store={store}>
      <App />
    </Provider>
  );
};

AppRegistry.registerComponent(appName, () =>
ReduxProvider);
```

This “*Provider*” helps us to access redux actions and state from anywhere in the app. Now, we need to add a case in “*server.ts*” file for “*server/join*” event:

```
case "server/join":
        console.log("Got join event", action.data);
        users[socket.id].username = action.data;
        users[socket.id].avatar =
createUserAvatarUrl();
        break;
```

Now, when we write a username and click on join chat button, we get:

Got join event Dfrfrvvtg

Hence, we've added the join event using Redux to the server. Lastly ,we need to navigate to "*ChatScreen*" after the join event so in "*JoinScreen.tsx*", in the "*Button*" component and inside "*onPress*" property, add:

```
<Button
          title="Join Chat"
          onPress={() => {
            dispatch({type: 'server/join', data:
username});
            navigation.navigate('ChatScreen');
          }}
        />
```

Now, after the join event we're navigated to "*ChatScreen*".

Navigating to FriendListScreen:

Now, after entering the username, we need to navigate to a new screen named "*FriendListScreen*" so create that inside the "*screens*" folder and add:

```
import React from 'react';
import {View, Text} from 'react-native';
import styles from '../../styles/style';

export default function FriendListScreen() {
  return (
    <View style={styles.container}>
      <Text>Friend List Screen</Text>
    </View>
  );
}
```

Then, add this screen in "*StackNavigator.tsx*" file:

```
<Stack.Screen name="FriendListScreen"
component={FriendListScreen} />
```

Lastly, in "*JoinScreen.tsx*", change the "*onPress*" of the join button to:

```
<Button
  title="Join Chat"
  onPress={() => {
    dispatch({type: 'server/join', data: username});
    navigation.navigate('FriendListScreen');
  }}
/>
```

Now, when we click on the join button, we navigate to "*FriendListScreen*".

Getting online users from Socket.io into Redux:

Now, in the "*FriendListScreen*", we want to display all the users that are online or connected to the socket.io server. So in "*server.ts*" file, after join event, we want to emit joined users to the client, so in it, add:

```
socket.on("action", (action) => {
    ...
      case "server/join":
        ...
        const values = Object.values(users);
        socket.emit('action', {type: 'users_online',
data: values});
        ...
  });
```

Now, in here, we don't want to send the "*socket_id*" to the client hence we're getting only the values of the users and then emitting the object to the client. Then, in "*reducer.ts*" file, add a listener of this action type i.e. "*users_online*":

```
case 'users_online':
      return {...state, usersOnline: action.data};
```

Now when a user is joined, in the console, we get:

{"message": "Good day!", "usersOnline": [{"avatar": "https://picsum.photos/253/125", "userId": 2, "username": "111"}]}

This is the user that joined the chat after entering the username. Now, filter this data because there may be a case where there are users who haven't joined without entering the usernames so we only want users that have their usernames. So, in "*server.ts*", inside "*join*" case, add:

```
const onlyWithUserNames = values.filter(
      (user: any) => user.username !== undefined);
socket.emit("action", { type: "users_online", data:
onlyWithUserNames });
```

Now, we'll receive only users with their usernames. Now, if another user joins the chat, in the console, we see two user details but previous user doesn't get updated when new user joins so instead of using "*socket.emit()*" we need to use "*io.emit()*" which will emit to all the users connected to the server including sender.

```
io.emit("action", { type: "users_online", data:
onlyWithUserNames });
```

Displaying online users and handling disconnect event:

Now, we need to display users who are connected to the server so in "*FriendListScreen.tsx*" add:

```
...
import {useSelector} from 'react-redux';

export default function FriendListScreen() {
  const usersOnline = useSelector((state: any) =>
state.usersOnline);

  return (
    <View style={styles.container}>
      <FlatList
        data={usersOnline}
        renderItem={({item}) => {
          return <Text>{item.username}</Text>;
        }}
        keyExtractor={item => item.username}
      />
    </View>
  );
}
```

"*useSelector()*" is used to get Redux state in functional components wherever in the app. "*usersOnline*" will have all the connected users data. Now, for the item's key, we're setting the "*username*" which is not efficient as usernames can be the same. So, we need to install a package named "*uuid*" inside the backend folder so write in the console:

npm install uuid

This will create a unique id for every different user. Then, in "*server.ts*" file, import it:

```
import { v4 as uuidv4 } from "uuid";
```

Now, set the uuid when we connect to the server so, inside "*io.on()*" set the "*userId*" to "*uuidv4*":

```
io.on("connection", (socket) => {
  ...
  users[socket.id] = { userId: uuidv4() };
  ...
});
```

Now, we want to remove the user when it gets disconnected so in the same file, create a handler for it after "*join*" event and before "*action*" event:

```
socket.on("disconnect", () => {
    delete users[socket.id];
    io.emit("action", {
      type: "users_online",
      data: createUsersOnline(),
    });
  });
```

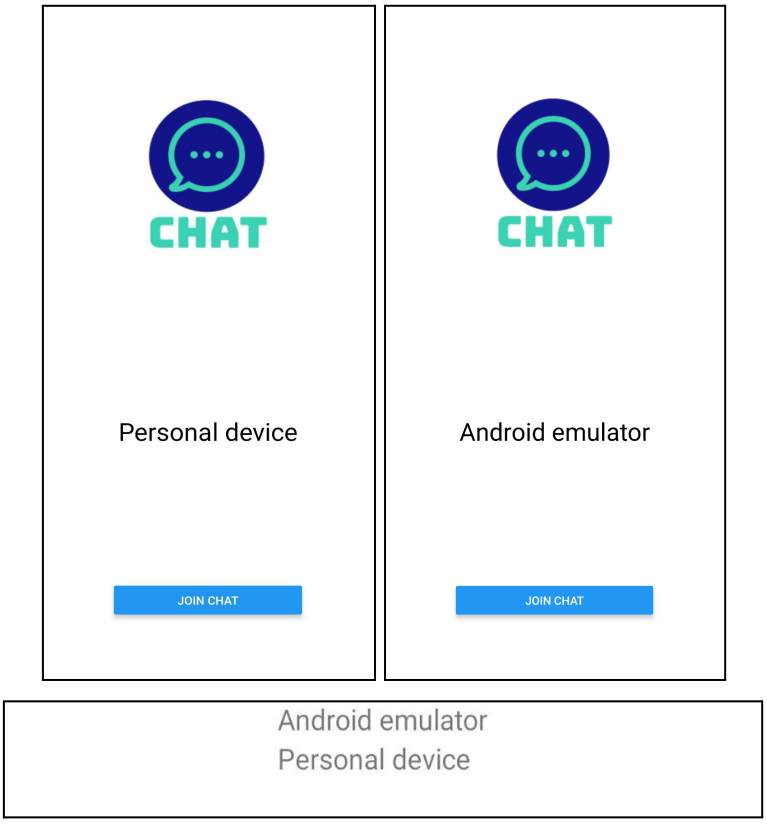
"*delete*" will remove the user from the "*users*" from the directory and then we'll emit the updated users object using "*createUsersOnline()*". Also, create this function:

```
function createUsersOnline() {
  const values = Object.values(users);
  const onlyWithUserNames = values.filter(
    (user: any) => user.username !== undefined
  );
  return onlyWithUserNames;
}
```

Also, call it inside “*join*” case inside “*action*” event:

```
socket.on("action", (action) => {
    ...
    case "server/join":
      ...
      io.emit("action", {
        type: "users_online",
        data: createUsersOnline(),
      ...
});
```

Now, when we create users, we get:



In the “*FriendListScreen*”, we see the above connected users. Now, if we close the app from one device, we get:

So, the user is removed when it closes the app or disconnects from the server.

Adding user avatar and improving UI:

Now, update "*FriendListScreen.tsx*" file to:

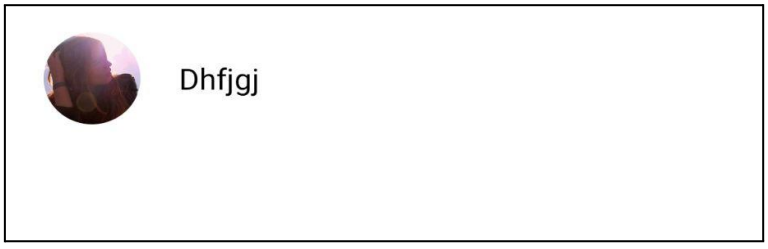
```
import React from 'react';
import {View, FlatList, Text, Image,
TouchableOpacity} from 'react-native';
import styles from '../../styles/style';
import {useSelector} from 'react-redux';

export default function FriendListScreen() {
  const usersOnline = useSelector((state: any) =>
state.usersOnline);

  return (
    <View style={styles.secondaryContainer}>
     <FlatList
      data={usersOnline}
      renderItem={({item}) => {
       return (
        <TouchableOpacity>
         <View style={styles.userListItemContainer}>
          <Image
           source={{uri: item.avatar}}
           style={styles.userListItemImageStyle}
          />
          <Text
style={styles.userListItemTextStyle}>
           {item.username}
```

```
            </Text>
          </View>
        </TouchableOpacity>
      );
    }}
    keyExtractor={item => item.userId}
   />
  </View>
 );
}
```

Then in "*styles.ts*" file, add:

```
secondaryContainer: {
    flex: 1,
    backgroundColor: '#FFF',
    padding: 20,
  },
userListItemContainer: {flex: 1, flexDirection:
'row', alignItems: 'center'},
  userListItemImageStyle: {width: 50, height: 50,
borderRadius: 50},
  userListItemTextStyle: {color: '#000', fontSize:
16, marginLeft: 20},
```

Now, the screen will look like this:



Navigating to Chat Screen after pressing on user in Friend List Screen:

Now, we need to navigate to "*ChatScreen*" when clicking on a user so in "*FriendListScreen.tsx*" file, add an "*onPress*" on "*TouchableOpacity*" component:

```
onPress={() => navigation.navigate('ChatScreen')}
```

Now, when we click on a user, we navigate to the Chat Screen. Now, in the "*ChatScreen.tsx*" file, remove the "*useState*", "*socket*" variable, "*useEffect*" and "*sendMessageToServer()*". Then, update the "*GiftedChat*" component to:

```
<GiftedChat
  renderUsernameOnMessage
  messages={[]}
  user={{_id: 1}}
  //onSend={messages =>
sendMessageToServer(messages)}
/>
```

Now, in this, we're only remaining with the "*GiftedChat*" component.

Sending private messages to Socket.io backend:

Now, we need to send private messages to the backend server. So, in "*ChatScreen.tsx*" file, add:

```
...
import {useDispatch} from 'react-redux';

export default function ChatScreen() {
  const dispatch: any = useDispatch();
```

```
  return (
    <GiftedChat
      ...
      onSend={messages =>
        dispatch({type: 'server/private-message',
data: messages})
      ...
```

In here, we're dispatching another action for private messaging to the server so now in "*server.ts*" file, add another case for this action type:

```
socket.on("action", (action) => {
  ...
  case "server/private-message":
    console.log("Got a private message",
action.data);
  break;
}
```

Now, when we send a message, in the server console, we get:

Got a private message [
{
text: 'Hxhxhch',
user: { _id: 1 },
createdAt: '2023-03-09T17:00:18.240Z',
_id: 'b996f9da-3d4d-4e73-a363-cc6e45e05151'
}
]

Here, we get the text message that we've typed and createdAt and message's user_id. Now, we want the text message and "*to*" property i.e., to which user we've sent the message so that the server will look up all the users and send the message to the right receiver. So, in "*GiftedChat*" component, update the "*dispatch*" to:

```
onSend={(messages: any) =>
        dispatch({
          type: 'server/private-message',
          data: {text: messages[0].text, to:
userId},
        })
      }
```

Now, we're only storing the text message and "*to*" property. Now, to get the "*userId*", inside "*FriendListScreen.tsx*" file, update the "*onPress()*" to:

```
onPress={() =>
            navigation.navigate('ChatScreen',
            {userId: item.userId})
        }
```

Now, in "*ChatScreen.tsx*" file, add a prop named "*route*" and access the "*userId*":

```
...
export default function ChatScreen({route}: any) {
    ...
    onSend={(messages: any) =>
    dispatch({type: 'server/private-message',
              data: {text: messages[0].text,
                     to: route.params.userId},
    ...
```

Now, when we send the message, in the server console, we get:

Got a private message { text: 'Hello', to: '8108e117-32c5-48a9-850a-ce4e44c4ced9' }

Here, we get the text message and "*userId*" to whom we're sending the message. Now, in "*server.ts*", we can match this "*userId*" to all the users and send the message to the right receiver.

Creating selfUser State property:

First, in "*server.ts*" file, remove the "*join*" event because now, we're using the "*action*" event:

```
socket.on("join", (username) => {
    users[socket.id].username = username;
    users[socket.id].avatar = createUserAvatarUrl();
    handleMessage(socket, users);
  });
```

Then, remove "*server/hello*" event as it is just a demo event:

```
case "server/hello":
        console.log("Got hello event", action.data);
        socket.emit("action", { type: "message",
data: "Good day!" });
        break;
```

Then, in "*reducer.ts*" file, remove "*message*" listener:

```
case 'message':
      return {...state, message: action.data};
```

Then remove the "*action.ts*" file. Then, in "*App.tsx*" file, remove the "*dispatch()*" function:

```
store.dispatch(action);
```

Now, in "*ChatScreen.tsx*" file, inside "*GiftedChat*" component, we've set the userId to 1:

```
user={{_id: 1}}
```

We need to set it to be unique that we actually have in the socket.io server. So, in "*server.ts*" file, inside "*server/join*" event, we'll send the socket that is connected and what the userId it has so add:

```
case "server/join":
    ...
    socket.emit("action", { type: "self_user", data:
    users[socket.id] });
break;
```

Now, "*users[socket.id]*" contains the client's userId that we can use in the "*GiftedChat*" component. Now, add a listener to it so in "*reducer.ts*" file, add:

```
case 'self_user':
      return {...state, selfUser: action.data};
```

Now ,when the click on join chat button, we get:

> *"selfUser": {"avatar": "https://picsum.photos/161/168", "userId": "f3eee0fd-bcde-4b93-9a6b-89d0e8f5d1aa", "username": "Bob"}*

Now, we get the client's details in the console. Now to use it, in "*ChatScreen.tsx*" file, add:

```
import {useSelector} from 'react-redux';
...
  const selfUser = useSelector((state: any) =>
state.selfUser);

  return (
    <GiftedChat
      ...
```

```
      user={{_id: selfUser.userId}}
      ...
    />
  );
}
```

Now, we've set the client's userId to be unique that we're getting from the socket.io server.

Creating conversations State Property:

Now, we need to maintain some kind of data structure for all the conversations that we have inside of the app. So, in "*reducer.ts*" file, add:

```
function reducer(state = {conversations: {}},
action: any) {
 switch (action.type) {
  case 'users_online':
   const conversations:any =
{...state.conversations};
   const usersOnline = action.data;
   for (let i = 0; i < usersOnline.length; i++) {
    const userId = usersOnline[i].userId;
    if (conversations[userId] === undefined) {
     conversations[userId] = {
      messages: [],
      userName: usersOnline[i].username,
     };
    }
   }
   return {...state, usersOnline, conversations};
   ...
```

In this, first we've created an initial state for "*conversations*" which is just an empty object. Then in the "*users_online*" listener, we'll add new conversations with the userIds as the users join the server. So, first we've made a copy of the initial state:

```
const conversations: any = {...state.conversations};
```

Then, we've retrieve all the online users connected to the server:

```
const usersOnline = action.data;
```

Then, we've added a loop where we're checking for each user and adding the "*messages*" and "*userName*" respective of their userId to the "*conversations copy*" created above.

```
for (let i = 0; i < usersOnline.length; i++) {
  const userId = usersOnline[i].userId;
  if (conversations[userId] === undefined) {
    conversations[userId] = {
      messages: [],
      userName: usersOnline[i].username,
    };
  }
}
```

Lastly, we've returned the updated state with online users and conversations:

```
return {...state, usersOnline, conversations};
```

Now, we join the chat, in the console, we get:

"conversations": {"2b42bf49-b104-4d9e-9026-b828de3cbcd7": {"messages": [], "userName": "Bob"}

Here, we get the messages and username based on their userId respectively.

Appending sent messages locally:

Now, we need to append the messages that we're sending in the app so in "*ChatScreen.tsx*", update the "*dispatch()*" function to:

```
dispatch({
          type: 'private_message',
          data: {message: messages[0],
conversationId: route.params.userId},
        })
```

Then in "*reducer.ts*" file, create a listener for above action:

```
case 'private_message':
  const conversationId = action.data.conversationId;
    return {
      ...state,
      conversations: {
        ...state.conversations,
        [conversationId]: {
          ...state.conversations[conversationId],
          messages: [
            action.data.message,
...state.conversations[conversationId].messages,
          ],
        },
      },
    };
```

In here, we ‘re just appending the previous messages with the newly added message as we type and storing all these in the “*conversations*” state.

Now, in “*ChatScreen.tsx*” file, fetch the messages using “*useSelector*”:

```
const conversations = useSelector((state: any) =>
state.conversations);
const messages =
conversations[route.params.userId].messages;
```

Then, use these messages inside “*GiftedChat*” component:

```
<GiftedChat
      ...
      messages={messages}
      ...
/>
```

Now, when we type messages, we’ll see the messages are appending in the correct order locally.

Sending messages across Socket.io:

Now, we’re appending the messages locally but we need to send these messages to receiver end also so in “*ChatScreen.tsx*” file, update the “*onSend()*” function to:

```
onSend={(message: any) => {
        dispatch({
          type: 'private_message',
          data: {message: message[0],
conversationId: route.params.userId},
        });
        dispatch({
```

```
          type: 'server/private_message',
          data: {message: message[0],
conversationId: route.params.userId},
        });
      }}
```

We've just added another "*dispatch()*" for "*server/private_message*" event. Then, in "*server.ts*" file, update the "*server/private-message*" listener to:

```
case "server/private_message":
  const conversationId = action.data.conversationId;
  const from = users[socket.id].userId;
  const userValues: any = Object.values(users);
  const socketIds = Object.keys(users);
  for (let i = 0; i < userValues.length; i++) {
    if (userValues[i].userId === conversationId) {
      const socketId = socketIds[i];
      io.sockets.sockets[socketId].emit("action", {
       type: "private_message",
       data: { ...action.data, conversationId: from
},
      });
      break;
    }
  }
break;
```

In this, we've retrieved conversationId, client's userId i.e. "*from*", all the users and socketIds. Then, we've checked for all the users that if "*userId*" is equal to the "*conversationId*". If it's true, then we retrieve the corresponding "*socketId*" and emit the messages to that particular socketId. Now, when we send messages we see the receiver also gets the message that is sent by the client.