FRONTEND: ChatContext

domenica 3 dicembre 2023 12:42

First of all, let's create a service get request function that we can use for any occasion (such as retrieving data from the database)

```
export const getRequest = async (url) => {
  try {
    const response = await fetch(url);

    if (!response.ok) {
      throw new Error(`HTTP error! status: ${response.status}`);
    }

    const data = await response.json();

    return data;
  } catch (error) {
    console.error(`Fetch failed from ${url}. ${error}`);
  }
};
```

The **getRequest** function is a GET request that takes a single parameter: the URL.
This URL is the endpoint of the server from which we want to retrieve data.

Now let's create a file in the **context** folder name **ChatContext.jsx**

Now let's import our utility functions:

```
import { baseUrl, getRequest, postRequest } from "../utils/service";
```

STEP 1:
We want to use **createContext** to make the **Chat** be available across the app.
So what we do next is create a context by simply doing this:

```
// CREATE CONTEXT TO SHARE CHAT STATE ACROSS THE APPLICATION.
export const ChatContext = createContext();
```

What now? Once we've created our **ChatContext** state, we need to set up the data that we want to share across the whole app, so our next step is create another component called **ChatContextProvider**. This component will return **ChatContext.Provider** and have as **value** all the things we want to share to our **{ children }** (In this case, **ChatContextProvider** will wrap all the children components in **App.jsx file.**

STEP 2:
Let's now create the **ChatContextProvider** we talked about before.
PS: we're going to pass it a **user** prop so that we can do the operations for the chat based on the current user.

```
export const ChatContextProvider = ({ children, user }) => {···
};
```

STEP 3:
Let's now use the **ChatContextProvider** component that we created to wrap the children of **App.jsx.** This way the chat will be available for all routes.
PS: we're now importing the info of the **current logged in user** by deconstructing **AuthContext**, and pass the actual user to the **ChatContextProvider** as props.

```
import { ChatContextProvider } from "./context/ChatContext";
import { AuthContext } from "./context/AuthContext";

const App = () => {
  const { user } = useContext(AuthContext);

  return (
    <ChatContextProvider user={user}>
      <Routes>···
      </Routes>
    </ChatContextProvider>
  );
};

export default App;
```

IT'S TIME TO MAKE THIS CHAT WORK!

1. FETCHING ALL THE AVAIABLE USER CHATS - with useEffect

How are we going to fetch all the available user chats for the current logged-in user? (the chats he already has with an user)
First of all, let's initialize all the essential states that we need in order to make this operation possible:
These are the states that we need to create with the **useState** hook:

1. **userChats**, to store the chat data
2. **isUserChatsLoading**, to indicate whether the data is currently being fetched
3. **userChatsError**, to hold any errors that occur during the fetch

```
const [userChats, setUserChats] = useState(null);
const [isUserChatsLoading, setIsUserChatsLoading] = useState(false);
const [userChatsError, setUserChatsError] = useState(null);
```

Now that we have all the states set up, we can create a **useEffect** hook to defined a function that will run after the component renders (ChatContextProvider).
This function, **getUserChats,** is an asynchronous function that sends a GET request to the **/chats/{userId}** endpoint of the API to fetch the chat data of the currently logged-

in user.
This function runs whenever the **user** change, as indicated by the dependency array passed to **useEffect.**
If the API response contains an error, it is stored in **userChatsError.** Otherwise, the chat data is stored in **userChats.** The **isUserChatsLoading** state is used to track when the data is being fetched and when it has finished loading.

```
// FETCH USER CHATS FOR THE CURRENT LOGGED IN USER
useEffect(() => {
  const getUserChats = async () => {
    setIsUserChatsLoading(true);
    setUserChatsError(null);

    if (user?._id) {
      const userId = user?._id;

      const response = await getRequest(`${baseUrl}/chats/${userId}`);

      if (response.error) {
        return setUserChatsError(response);
      }

      setUserChats(response);
    }

    setIsUserChatsLoading(false);
  };

  getUserChats();
}, [user]);
```

2. HOW TO MAP ALL THE AVAIABLE USER CHATS + READ RECIPIENT USER DETAILS

Now that we have created a function that fetches all the **available user chats** that the current logged-in user has, let's now map every available chat.
First of all, let's create a page component called **Chat.jsx**

```
{userChats?.map((chat, index) => {
  return (
    <div
      id={`User_Card_${index + 1}`}
      key={index}
      onClick={() => updateCurrentChat(chat)}
    >
      <UserCard chat={chat} user={user} />
    </div>
  );
})}
```

Afterward, we use the array method .**map** that will create as many "**UserCard"** component as the quantity of the chat that the current logged-in user has.
We're also passing the **chat** and the current logged-in **user** info (from AuthContext) as props to the component.

We're now going to create a custom hook to get the **UserCard** details.
Let's go to the **hooks** folder and create a custom hook called **useFetchRecipientUserInfo,** it'll receive as props the **chat** and the **current logged in user.**
Next, we're going to create two states:

1. **recipientUser** - to save the recipient user details for its UserCard
2. **recipientUserError** - to catch errors during the fetch

```
const [recipientUser, setRecipientUser] = useState(null);
const [recipientUserFetchError, setRecipientUserFetchError] = useState(null);
```

How are we going to do this?
If you remember we've passed as props **chat** and **user** to the **useFetchRecipientUserInfo.**
So what we need to do next, is to find the correct **recipientId** (the prop **chat** has both the **current logged user id** and the **recipient id** that we're trying to extract).
So in order to do this, we need to find it. How? By creating a variable called **recipientId** that will find the **chat.members** data object the **ID** of the **recipient user** that is not equal to the **current logged in user**.

```
const recipientId = chat?.members.find((id) => id !== currentUser?._id);
```

We're going to make use of the **recipient id** that we have found, to fetch its details (name, surname etc...)
So we're going to use **useEffect** to fetch the data of the **recipient user.**

```
// FETCH THE RECIPIENT USER DETAILS.
useEffect(() => {
  const getRecipientUser = async () => {
    try {
      if (!recipientId) return null;

      const response = await getRequest(
        `${baseUrl}/users/find/${recipientId}`
      );

      setRecipientUser(response);
    } catch (error) {
      setRecipientUserFetchError(error);
      console.log(recipientUserFetchError);
    }
  };

  getRecipientUser();
}, [recipientId]);

return { recipientUser };
```

This is possible because when making a get request to this path **/users/find/** and we put the recipient id next to it, we're going to get this user's data.

We've talked about the UserCard component, so let's now take a closer look of how it works:

```
const UserCard = ({ chat, user }) => {
  const { recipientUser } = useFetchRecipientUserInfo(chat, user);
```

As you remember, we've passed the props **chat** and **user** to the **UserCard** component, so we're now going to pass it to the **useFetchRecipientUserInfo** in order to get their details such as name and surname.

We can finally set the **recipient name** and **surname** in the **UserCard** component.
Remember that the **UserCard** component is being mapped for **each chat** that the current logged in user has, so it will automatically set the right name and surname of the recipient user.

```
{/* SINGLE USER CARD - LEFT (AVATAR, NAME, SURNAME & LAST MESSAGE) */}
<div className="flex">
  {/* AVATAR */}
  <div className="mr-2">
    <img src={avatar} alt="person-circle" width="35px" />
  </div>
  {/* NAME, SURNAME & LAST MESSAGE */}
  <div className="UserDataInfo">
    {/* NAME, SURNAME */}
    <div className="font-bold">
      {recipientUser?.name} {recipientUser?.surname}
    </div>
    {/* LAST MESSAGE TIME STAMP */}
    <div className="text-sm text-gray-500">
      {latestMessage?.text && (
        <span>{truncateText(latestMessage?.text)}</span>
      )}
    </div>
  </div>
```

3. CREATE CHATS

Initially, when you first set your user account, you have no potential user chats. In order to create them you need to match with other users.
When a new match is found, you'll receive a notification at the top of your chat saying that you have a match and you're ready to start a chat with the user just by clicking on the name.
This will create a new chat. Once the new chat is created, it'll be part of all the chats that the current logged in user can have.

Let's precede doing this by creating this state:

```
// POTENTIAL CHATS STATE || MATCHED USERS
const [potentialChats, setPotentialChats] = useState(null);
```

and this (**we'll take a better look at this later)**

```
// ALL REGISTERED USERS STATE
const [allUsers, setAllUsers] = useState([]);
```

```
// FETCH ALL USERS AND POTENTIAL CHATS || MATCHED USERS
useEffect(() => {
  const getUsers = async () => {
    const response = await getRequest(`${baseUrl}/users`);

    if (response.error) {
      return console.log("Error fetching users:", response);
    }

    if (userChats) {
      const pChats = response?.filter((user) => {
        let isChatCreated = false;

        if (user._id === user._id) return false;

        isChatCreated = userChats?.some(
          (chat) =>
            chat.members[0] === user._id || chat.members[1] === user._id
        );

        return !isChatCreated;
      });

      setPotentialChats(pChats);
    }

    setAllUsers(response);
  };

  getUsers();
}, [userChats]);
```

Let's break down this code:

1. **Initialization:** The **useEffect** hook is called when the components mounts or when the **userChats** dependency array changes.
2. **Fetching Users:** The **getUsers** async function sends a GET request to the server to retrieve all the users.
3. **Error handling:** If there is an error in the response, it's logged to the console and the function exits early.
4. **Filtering Potential Chats:** if **userChats** exists (meaning the current logged in user already has some chats), the code filters the list of all users to find potential new chat partners.
   This is done by:
   a. Excluding the current logged in user from the list to prevent creating a chat with oneself.
   b. Using the **some** method to check if a chat already exist with each user. If a chat exist, the user is excluded from the potential chats list.
   We intentionally check in both **chat.members[0]** and **chat.members[1]** if there is a user that has the same id as the logged in user, to avoid mistakes.
5. **Setting State:** The **setPotentialChats** function updates the component's state with the filtered list of users (**pChats)** who the current user does not have a chat with yet.
   The **setAllUsers** function updates the state with the list of all users received from the server.
6. **Clean-up:** The **getUsers** function is called to execute the above steps.

Once we have our **potentialChats** user list, we can use it in another component called **MatchedUsers.jsx**
This component will be visible whenever it will recognize a new potential chat (whenever we get a match) and it will be displayed on top of our chat.

So after we create the **MatchedUsers.jsx** component, we can now import the **potentialChats** from **ChatContext.js**

```
const { potentialChats} = useContext(ChatContext);
```

Now the next step is to map all the potential users we can have a chat with, and show them at the top of our chat.

```
<div id="MatchedUsers_List" className="flex flex-row">
  {potentialChats &&
    potentialChats.map((potentialUser, index) => (···
    ))}
</div>
```

This will map each **potentialUser** in this div.

Now, what's missing is create a chat when we click on these **potential users** that we have retrieved.
So let's go back again in **ChatContext.js** and write this snippet of code:

```
const createChat = useCallback(async (senderId, receiverId) => {
  const response = await postRequest(
    `${baseUrl}/chats`,
    JSON.stringify({ senderId, receiverId })
  );

  if (response.error) {
    return console.log("Error creating chat:", response);
  }

  setUserChats((prev) => [...prev, response]);
}, []);
```

This **createChat** function will make a **postRequest** to the **API URL** provided from the backend, and we'll pass the **senderId** and the **receiverId** to it in order to create a new chat with these two users. The backend will check if we already have a chat with this user, if not it will create it.
Then code checks if the response returns an error, if yes, we'll console.log it.
Then, if we don't have an error, we can finally set the user chats list that the current logged-in user can have, by simply doing:

```
setUserChats((prev) => [...prev, response]);
```
This will keep the existing user chats list that the user already have, and also add the new one we just created (the response from the server).

Now let's use this function in the **MatchedUsers.jsx** component.
When we finally get a new match, and the user name will show, we can add a **onClick event** on this user that will create a chat between the current logged in user and the user we have just matched.

We can do this by importing the current logged-in user info
```
const { user } = useContext(AuthContext);
```
and also by importing the function we just created
```
const { potentialChats, createChat } = useContext(ChatContext);
```

After we've done this, we can pass to the **createChat** function the **current logged-in user id** as the first parameter and **the potential user id** as the second parameter.
```
<div id="MatchedUsers_List" className="flex flex-row">
  {potentialChats &&
    potentialChats.map((potentialUser, index) => (
      <div
        id={`Matched_User_${index + 1}`}
        onClick={() => createChat(user._id, potentialUser._id)}
      >
        {potentialUser.name} {potentialUser.surname}
        <span
        ></span>
      </div>
    ))}
</div>
```

4. GET MESSAGES

Now that we know how to create a chat with a potential user, we may want to log in our app the messages we send with this potential user.
In way to do this, we need to set the current chat with the user.
Let's go back in our **ChatContext.jsx** component and create a new state:
```
const [currentChat, setCurrentChat] = useState(null);
```

Now we need a function that will update this state.
```
const updateCurrentChat = useCallback(async (chat) => {
  setCurrentChat(chat);
}, []);
```

We're going to pass the **updateCurrentChat** function in our **Chat.jsx page.**
Let's start by importing this function:
```
const { userChats, isUserChatsLoading, userChatsError, updateCurrentChat } =
  useContext(ChatContext);
```

Now that we have imported it, we need to add a **onClick** event on the div that contains the **UserCard** (which represents the current user we've clicked on and we want to chat with), and pass the **chat** element as a parameter (the **chat** element is ab object that contains the **ID** of the ongoing chat, and the members that are participating to this chat)
```
{userChats?.map((chat, index) => {
  return (
    <div
      id={`User_Card_${index + 1}`}
      key={index}
      onClick={() => updateCurrentChat(chat)}
    >
      <UserCard chat={chat} user={user} />
    </div>
  );
})}
```

Now that we've set the current chat, we need to retrieve the messages for the current chat.
Let's start by going back to the **ChatContext.jsx** and add these states:
```
const [messages, setMessages] = useState(null);
const [messagesError, setMessagesError] = useState(null);
const [isMessagesLoading, setIsMessagesLoading] = useState(false);
```

Once we're done with that, we need to create this function:

```
useEffect(() => {
  const getMessages = async () => {
    setIsMessagesLoading(true);

    const response = await getRequest(
      `${baseUrl}/messages/${currentChat?._id}`
    );

    setIsMessagesLoading(false);

    if (response.error) {
      return setMessagesError(error);
    }

    setMessages(response);
  };
  getMessages();
}, [currentChat]);
```

Let's break down this code:

1. **Fetching Messages:** The useEffect hook is used to perform side effects, in this case, fetching messages. It runs when the **currentChat** object changes, which indicates a new chat selection by the user.
2. **Asynchronous Request:** Inside the **useEffect**, an asynchronous function **getMessages** is defined and immediately invoked. This function sets **isMessagesLoading** to **true** to indicate the start of the fetch operation.
3. **GET Request:** A GET request is made to the server endpoint constructed with the **baseUrl** and the **ID** of the **currentChat.** This request is meant to retrieve messages specific to the selected chat.
4. **Response Handling:** After the request, **isMessagesLoading** is set back to **false.**
   If the response contains an error, **setMessagesError** is called to update the **messagesError** state.
   If there is no error, **setMessages** is called to update the messages state with the fetched messages for the current selected chat id.

Now that we have this function ready, it's time to create the **ChatBox.jsx** component which will the actual chat where we will display messages between 2 users.
After we create it, we can import it in the **Chat.jsx** page.

```
  return (
    <div
      id="ChatContainer"
      className="flex flex-col justify-center mx-auto ml-10"
    >
      {/* SHOW MATCHED USERS IF THERE ARE ANY */}
      <MatchedUsers />

      {/* AVAILABLE USERS CHATS & ACTUAL CHAT */}
      {userChats?.length < 1 ? null : (
        <div id="UserChatList_and_Chat" className="flex flex-row space-
          {/* USER CHATS AVAIABLE */}
          <div id="UserChatList">···
          </div>

          {/* ACTUAL CHAT FOR CHATTING */}
          <ChatBox />
        </div>
      )}
    </div>
  );
};

export default Chat;
```

Let's now open the **ChatBox.jsx component** and import the current logged-in user by deconstructing it from AuthContext.

```
const { user } = useContext(AuthContext);
```

Now let's extract the **current chat id** (we set when before when clicking on the user chat) and the **current logged-in user id** so that we can get the **recipientUser** (the user we're chatting with).

Let's start my importing the **currentChat**

```
const { currentChat, messages, isMessagesLoading, sendTextMessage } =
  useContext(ChatContext);
```

Now that we have both the **currentChat** id and the **user** id, we can pass these values to the **useFetchRecipientUserInfo** in order to get the **recipientUser.**

```
const { recipientUser } = useFetchRecipientUserInfo(currentChat, user);
```

If we don't have the recipient user:

```
  if (!recipientUser)
    return (
      <div className="flex flex-col items-center justify-center w-full font-bold text-center">
        <h2 className="mb-8 text-2xl">What are you waiting for?</h2>
        <p className="text-4xl text-purple-500">Start chatting!</p>
      </div>
    );
```

If the messages are loading...

```
if (isMessagesLoading)
  return <p className="w-full text-center">Loading chat...</p>;
```

If none of the previous condition is true, we can finally set the header of the recipient user we're chatting with:

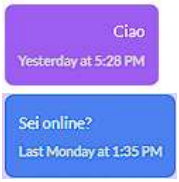
```
<div
  id="ChatBox_User_Info"
  className="justify-between p-3 text-center text-white bg-gray-900"
>
  <strong>
    {recipientUser?.name} {recipientUser?.surname}
  </strong>
</div>
```

Now that we've done that, we can also find the messages we exchanged with the current recipient user (we fetched them before with **getMessages** function that fetched the user messages based on the **chat id** we pass to it when we **click** on the user we want to chat with; this will trigger the **updateCurrentChat function**)

```
{messages &&
  messages?.map((message, index) => (
    <div
      id={`ChatBox_UserMessage_${index + 1}`}
      className={`${
        message?.senderId === user?._id
          ? "bg-purple-500 text-white p-3 rounded-lg self-end text-end max-w-[48%] break-words"
          : "bg-blue-500 text-white p-3 rounded-lg self-start text-start max-w-[48%] break-words"
      }`}
      key={index}
      ref={scroll}
    >
      <span className="text-sm text-wrap">{message.text}</span>
      <br></br>
      <span className="text-xs font-semibold text-gray-300">
        {moment(message.createdAt).calendar()}
      </span>
    </div>
```

In this code we're mapping the messages of the users.
We filter them by using the conditional operator: if in the message object, there is a **senderId** that is **equal** the same **current logged in user**, the color of the message will be **purple**, otherwise it will be **blue.**

About the timestamps for the message, we're using **moment.** It's a library for react. You can install it with: npm i moment**.**
What it does is: it accesses to the **message.createdAt** object and then, based on the timestamp, it will display a date under the message like this:



4. SEND MESSAGES

In the **ChatBox.jsx** page, we're also going to use a library called **react-input-emoji**
We are going to place this input field text component at the bottom of the ChatBox page.

```
{/* INPUT TEXT */}
<div
  id="ChatBox_TextArea"
  className="flex flex-row p-4 space-x-3 bg-gray-800 self-centerflex-wrap"
>
  <InputEmoji
    value={textMessage}
    onChange={setTextMessage}
    onEnter={() =>
      sendTextMessage(textMessage, user, currentChat._id, setTextMessage)
    }
    shouldReturn="true"
    theme="dark"
    maxLength="300"
    fontSize="1rem"
  />
</div>
```

Let's now save the messages we type in the input field box by adding them to a state.

```
const [textMessage, setTextMessage] = useState("");
```

Let's now go to **ChatContext.jsx** component and create a function that will help us send the message to the backend.
First of all, let's create the states that we're going to need to execute this function:

```
const [sendTextMessageError, setSendTextMessageError] = useState(null);
const [newMessage, setNewMessage] = useState(null);
```
newMessage state will hold the response from the server after a message is successfully sent.

Now let's create the function that will allow us to send the messages:

```
const sendTextMessage = useCallback(
  async (textMessage, sender, currentChatId, setTextMessage) => {
    if (!textMessage) return console.log("You must type something...");

    const response = await postRequest(
      `${baseUrl}/messages`,
      JSON.stringify({
        chatId: currentChatId,
        senderId: sender._id,
        text: textMessage,
      })
    );

    if (response.error) {
      return setSendTextMessageError(response);
    }

    setNewMessage(response);
    setMessages((prev) => [...prev, response]);
    setTextMessage("");
  },
  []
);
```
Let's break down this code:

1. **Function Definition:** The **sendTextMessage** function takes four parameters:
   -**textMessage:** the content of the message to be sent
   -**sender:** the user object representing the sender of the message
   -**currentChatId:** The ID of the chat to which the message is being sent
   -**setTextMessage:** A function to reset the message input field after sending
2. **Validation:** Before sending, the function checks if **textMessage** is not empty to prevent sending blank messages. If it's empty, a console.log is generated, and the function exits early.
3. **Sending Message:** A POST request is made to the server with the message details. The body of the request includes the **chatId, senderId,** and the **text** of the message.
4. **Error Handling:** if the server responds with an error, the **setSendTextMessageError** function updates the **sendTextMessageError** state with the error details.
5. **Updating State:** if the message is sent successfully (no error in response), the following updates occur:
   -**setNewMessage** updates the **newMessage** state with the server's response.
   -**setMessages** updates the messages list by appending the new message to the existing messages.
   -**setTextMessage** is called with an empty string to clear the message input field.

In summary, this function called **sendTextMessage** is called on a **onEnter event.**

Let's now scroll to the latest message whenever there is a new message.
First of all, let's create a ref to our scroll

```
const scroll = useRef();
```

and add it to our chat messages box:

```
{messages &&
  messages?.map((message, index) => (
    <div
      id={`ChatBox_UserMessage_${index + 1}`}
      className={`${
        message?.senderId === user?._id
          ? "bg-purple-500 text-white p-3 rounded-lg self-end text-end max-w-[48%] break-words"
          : "bg-blue-500 text-white p-3 rounded-lg self-start text-start max-w-[48%] break-words"
      }`}
      key={index}
      ref={scroll}
    >
      <span className="text-sm text-wrap">{message.text}</span>
      <br></br>
      <span className="text-xs font-semibold text-gray-300">
        {moment(message.createdAt).calendar()}
      </span>
    </div>
```

Now let's create this function:

```
useEffect(() => {
  scroll.current?.scrollIntoView({ behavior: "smooth" });
}, [messages]);
```
This **useEffect** function will be triggered each time a new **messages** arrive, and it'll scrollIntoView smoothly.

Now we need to implement the Socket.io service, so we can check if the current user is online. We'll see you there.