Step 5: Working with Open AI

ChatGPT API

We handle sessions:

```
sessions = {
  sessionId: UID,
  conversations: [conversation]
}
conversation = {
  id: UID,
  messages: [message, aiMessage],
}
message = {
 id: UID,
 content: string,
 aiMessage: false,
}
aiMessage = {
  id: UID
  content: string
  aiMessage: true,
};
```

From ChatGPT, we can get the aiMessage; we store the aiMessage to the conversation and update the current session.

Server

src/ai.js

We need to use the OpenAI API.

```
// server/src/ai.js
const { Configuration, OpenAIApi } = require("openai");
require("dotenv").config(); // no path needed

const configuration = new Configuration({
  apiKey: process.env.OPENAI_API_KEY,
});

const openai = new OpenAIApi(configuration);
module.exports = openai;
```

- We create the OpenAIApi Object `openai` and export it.

src/socketServer.js

sessionHistoryHandler

```
socket.on("session-history", (data) => {
  sessionHistoryHandler(socket, data);
});
```

This function receives an argument data (`sessionId`) and emits session: `{sessionId, conversations}`

When the session with `sessionId` exists:

```
if (sessions[sessionId]) {
```

It returns sesson JSON.

```
socket.emit("session-details", {
  sessionId,
  conversations: sessions[sessionId],
});
```

We use `session-details` API.

When the session with `sessionId` does not exist:

We generate a session JSON - new session Id, and empty conversations list and emit the sesson JSON.

```
const newSessionId = uuid();
sessions[newSessionId] = [];
const sessionDetails = {
  sessionId: newSessionId,
  conversations: [],
};
socket.emit("session-details", sessionDetails);
```

conversationMessageHandler

```
socket.on("conversation-message", (data) => {
  conversationMessageHandler(socket, data);
});
```

This function receives an argument data (`sessionId, message, conversationId`) and emits session: `{sessionId, conversations}`

Get `sessionId, message, conversationId` from the argument data and make an array to store previous messages.

If the conversations doesn't exist, exit.

```
const { sessionId, message, conversationId } = data;
if (!sessions[sessionId]) return;
```

Get the converation with the `conversationId` in current session.

```
const existingConversation = sessions[sessionId].find(
  (c) => c.id === conversationId
  );
```

If the conversation exists, transform the messages ([message, aiMessage]) into [{content, role}] to communicate with ChatGPT AI.

```
const previousConversationMessages = [];
if (existingConversation) {
  previousConversationMessages.push(
    ...existingConversation.messages.map((m) => ({
      content: m.content,
      role: m.aiMessage ? "assistant" : "user",
    }))
  );
}
```

This is an example:

```
existingConversation
{
  messages: [
    { content: 'Hello', aiMessage: false },
    { content: "What's up", aiMessage: true }
  ]
}

=>

previousConversationMessages
[
  { content: 'Hello', role: 'user' },
  { content: "What's up", role: 'assistant' }
]
```

The `...` is the spread operator used to insert each element into previousConversationMessages one by one.

Without the spread operator, we would add the array itself, resulting in an array inside an array.

```
const a = [1, 2, 3];
b.push(a);
// b becomes: [ [1, 2, 1,2,3] ]   ← array inside array

const a = [1, 2, 3];
b.push(...a);
// b becomes: [1, 2, 3]    ← elements added individually
```

We give exisiting conversation and new message as an argument and make a request to ChatGPT.

```
try {
  const response = await openai.createChatCompletion({
    model: "gpt-3.5-turbo",
    messages: [
      ...previousConversationMessages,
      { role: "user", content: message.content },
    ],
  });
  aiMessageContent = response?.data?.choices?.[0]?.message?.content || aiMessageContent;
} catch (err) {
  console.error("OpenAI error:", err.response?.data || err.message);
}
```

Making a Request and Getting Answer using ChatGPT API

To make a request to ChatGPT, we should make a list of JSON objects including history of question and answer: "assistant" role means the content from ChatGPT and "users" role means the content that we make.

```
[
  {
    content: "What is 1+1", // old question
    role: "users"
  },
  {
    content: "What is 2", // old answer
    role: "assistant"
  },
  {
    content: "What is 1*1", // new question
    role: "users"
  }
]
```

Making Request

We make the `Chat Completion` request to OpenAI:

```
const response = await openai.createChatCompletion({
  model: "gpt-3.5-turbo",
  messages: [
    ...previousConversationMessages,
    { role: "user", content: message.content },
  ],
});
```

- model: "gpt-3.5-turbo" specifies which OpenAI model to use.
- The full conversation history
- `{ role: "user", content: message.content }` gives the user input "message" so it can respond naturally.

Extracting the AI’s Reply

The answer is stored in `response`:

```
aiMessageContent = response?.data?.choices?.[0]?.message?.content || aiMessageContent;
```

- `response?.data?.choices?.[0]?.message?.content`
  This is the actual AI-generated text.
- If, for some reason this value is undefined,
  we keep the previous aiMessageContent instead of crashing.
- This is safe extraction using optional chaining (?.) to avoid runtime errors.

We make a new AI message from the returned content from ChatGPT.

```
const aiMessage = {
  content: aiMessageContent,
  id: uuid(),
  aiMessage: true,
};
```

We find the conversation ID

```
const conversation = sessions[sessionId].find((c) => c.id === conversationId);
```

Then, if there is no converation create a new conversationId with messages.

```
if (!conversation) {
  sessions[sessionId].push({
    id: conversationId,
    messages: [message, aiMessage],
  });
```

If there is an existing conversation, push (append) message and AI generated message to the message list.

```
} else {
  conversation.messages.push(message, aiMessage);
}
```

Get the conversation and emit using the `conversation-details` API.

```
const updatedConversation = sessions[sessionId]
.find((c) => c.id === conversationId);

socket.emit("conversation-details", updatedConversation);
```

conversationDeleteHandler

```
socket.on("conversation-delete", (data) => {
  conversationDeleteHandler(socket, data);
});
```

We delete the session with `sessionId`.

```
const conversationDeleteHandler = (_, data) => {
  const { sessionId } = data;

  if (sessions[sessionId]) {
    sessions[sessionId] = [];
  }
};
```

Client

Dashboard/Chat/Messages.js

Automatic scrolling

In this example, we implement the feature of automatic scrolling down to the latest message.

```
  return (
    <div className="chat_messages_container">
      {conversation?.messages.map((m, index) => (
        <Message
          key={m.id}
          content={m.content}
          aiMessage={m.aiMessage}
          animate={index === conversation.messages.length - 1 && m.aiMessage}
        />
      ))}
      <div ref={scrollRef} />
    </div>
  );
};
```

This code sets only the last message as animate = true.

```
animate={index === conversation.messages.length - 1 && m.aiMessage}
```

We use `scrollRef` reference to a real DOM element that React keeps for you so you can scroll to it programmatically.

```
const scrollRef = useRef();

// In React
...
  <div ref={scrollRef} />
...
```

- `useRef()` creates an object with a .current property.
- `React` assigns .current to the actual DOM node of whatever element you attach it to:

We make scrollToButton is activated when message are updated.

```
useEffect(scrollToButton, [conversation?.messages]);
const scrollToButton = () => {
  scrollRef.current.scrollIntoView({ behavior: smooth" });
};
```

As a result, the chat window automatically scrolls down to the latest message.

Dashboard/Chat/NewMessageInput.js

Set state fields

We can get `conversations` and set `selectedConversation` with `selectedConversationId` with new state field.

```
const conversations = useSelector((state) => state.dashboard.conversations);

const selectedConversation = conversations.find(
  (c) => c.id === selectedConversationId
);
```

Disable Input

```
  return (
    <div className="new_message_input_container">
      <input
        ...
        disabled={ // <---
          selectedConversation &&
          !selectedConversation.messages[
            selectedConversation.messages.length - 1
          ].aiMessage
        }
      />
      <div className="new_message_icon_container" onClick={handleSendMessage}>
        <BsSend color="grey" />
      </div>
    </div>
  );
};
```

The disabled `={ ... }` logic controls when the user is allowed to type a new message.

```
disabled={
  selectedConversation &&
  !selectedConversation.messages[
    selectedConversation.messages.length - 1
  ].aiMessage
}
```

This expression becomes true or false, which then disables or enables the input.

Using state field, we get the conversations and selectedConversationId from stored states.

```
const dispatch = useDispatch();
const selectedConversationId = useSelector(
  (state) => state.dashboard.selectedConversationId
);
const conversations = useSelector((state) => state.dashboard.conversations);
```

Find the conversation from the conversations.

```
const selectedConversation = conversations.find(
  (c) => c.id === selectedConversationId
);
```