Step 3: Sessions

Define Sessions and use it for Communication & Storage

You should understand the Redux Toolkit (RTK), specifically Slice, States, and Store.

- We use Redux Slice and States to store session and messages.
- Using Redux, we can access the information anywhere in the app.

Using Redux Slice States and Actions

Get the state fields

We use `useSelector`: this is an example to get the `selectedConversationId` state field.

```
import { useSelector } from "react-redux";
const selectedConversationId = useSelector(
  (state) => state.dashboard.selectedConversationId
);
```

Trigger action

We use `dispatch` : this is an example to trigger the addMessage action with a JSON object.

```
const dispatch = useDispatch();
dispatch(
  addMessage({
    conversationId,
    message,
  })
);
```

session, conversation, and message

`sessons` is conversations with ID.

```
sessions = {
  sessionId: UID,
  conversations: [conversation]
}
```

`conversation` is a pair (list) of message and aiMessage with ID.

```
conversation = {
  id: UID,
  messages: [message, aiMessage],
}
```

`message` is content with ID; it can be aiMessage or normal message.

- content is the user input or AI generated output.

```
message = {
 id: UID,
 content: string,
 aiMessage: false,
}
aiMessage = {
  id: UID
  content: string
  aiMessage: true,
};
```

Server

src/socketServer.js

In this server, we process two WebSocket APIs:

1. session-history: to returns `conversations` from `sessionID`.
2. conversation-message: nothing in this step

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

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

1. session-history: sessionHistoryHandler

If we have `sessions[sessionId]`, return the `session` with `sessionId` and `conversations`.

```
const sessionHistoryHandler = (socket, data) => {
  const { sessionId } = data;

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

If not, create a new session (sessionId and conversations) with a generatedID and return empty list (conversations), and emit the session using the `session-details` API.

```
  } else {
    const newSessionId = uuid();

    sessions[newSessionId] = [];

    const sessionDetails = {
      sessionId: newSessionId,
      conversations: [],
    };

    socket.emit("session-details", sessionDetails);
  }
};
```

2. conversation-message:conversationMessageHandler

We display the data in this step.

```
const conversationMessageHandler = (socket, data) => {
  console.log("message came from client side");
  console.log(data);
};
```

Client

store.js

We use Redux store, and its reducer `dashboard` to share states in the app.

```
import { configureStore } from "@reduxjs/toolkit";
import dashboardReducer from "./Dashboard/dashboardSlice";

export const store = configureStore({
  reducer: {
    dashboard: dashboardReducer,
  },
});
```

`dashboard` has state fields to store messages and related Ids that can be used anywhere in the app.

main.js

The main app (App) uses the React Redux store.

```
import React from "react";
import ReactDOM from "react-dom/client";
import { Provider } from "react-redux";
import App from "./App";
import { store } from "./store";

import "./index.css";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <Provider store={store}>
    <App />
  </Provider>
);
```

Dashboard/dashboardSlice.js

Dashboard has one Redux slice ('dashboard') with an initial state containing three fields.

```
const dashboardSlice = createSlice({
  name: "dashboard",
  initialState,
  reducers: ...
```

```
const initialState = {
  sessionEstablished: false,
  conversations: [],
  selectedConversationId: null,
};
```

Recucers and Actions

The dashboard slice has three reducers and related actions:

1. setSelectedConversationId
2. setConversations
3. addMessage

We use the actions to trigger action, and the reducers to act upon it.

setSelectedConversationId:

1. sets the state field `selectedConversationId` to the argument (action.payload)

```
    setSelectedConversationId: (state, action) => {
      state.selectedConversationId = action.payload;
    },
```

setConversations:

1. sets sessionEstablished to true to notify the start of communication between the server and client.
2. sets the state field `conversations` stores to the argument (action.payload).

```
    setConversations: (state, action) => {
      state.conversations = action.payload;
      state.sessionEstablished = true;
    },
```

addMessage:

1. retrieves message and coversationId from argument (action.payload)
2. finds the conversation in the state field `conversations` with the conversationId.

```
    addMessage: (state, action) => {
      const { message, conversationId } = action.payload;

      const conversation = state.conversations.find(
        (c) => c.id === conversationId
      );
```

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

3. If the coversation is found, add (push) the message to the `messages` state field.

```
      if (conversation) {
        conversation.messages.push(message);
      }
```

4. If not, make a new conversation with the `convesationId` and `[message]`.

```
      else {
        state.conversations.push({
          id: conversationId,
          messages: [message],
        });
      }
    },
```

Dashboard/Dashboard.js

To access the state fields, we need to use `useSelector`.

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

Dashboard component

If the state field `sessionEstablished` is false, we load the `<LoadingSpinner />` component to show the React app is waiting for the server connection.

```
const Dashboard = () => {
  const sessionEstablished = useSelector(...);
  return (
    <div className="dashboard_container">
      <Sidebar />
      <Chat />
      {!sessionEstablished && <LoadingSpinner />}
    </div>
  );
};

export default Dashboard;
```

Dashboard/Sidebar

Dashboard/Sidebar/NewChatButton.js

The `newChatButton` component draws '+' icon and a string "New Chat".

- When clicked the button, the function given as the prop is exectued with the "new" argument.

```
const NewChatButton = ({ handleSetSelectedChat }) => {
  const handleChooseNewChat = () => {
    handleSetSelectedChat("new");
  };

  return (
    <div className="new_chat_button"
      onClick={handleChooseNewChat}>
```

Usage of NewChatButton component

in the Dashboard/Sidebar/Sidebar.js, the handleSetSelectedChat function is given as the prop.

```
  const handleSetSelectedChat = (id) => {
    dispatch(setSelectedConversationId(id));
  };

  return (
    <div className="sidebar_container">
      <NewChatButton handleSetSelectedChat={handleSetSelectedChat} />
```

The `setSelectedConversationId` becomes "new" from the `handleSetSelectedChat("new")` in the NewChatButton component.

Dashboard/Sidebar/Sidebar.js

Now, the Sidebar component has three components:

1. NewChatButton

```
<NewChatButton handleSetSelectedChat={handleSetSelectedChat} />
```

The handleSetSelectedChat function updates the `setSelectedConversationId` state field.

```
const handleSetSelectedChat = (id) => {
  dispatch(setSelectedConversationId(id));
};
```

2. ListItem (conversations)

We get conversations from state field.

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

We display the conversation information (ID and message content) using ListItem component.

```
{conversations.map((c) => (
  <ListItem key={c.id} title={c.messages[0].content} chatId={c.id} />
))}
```

3. DeleteConversationsButton

```
<DeleteConversationsButton />
```

Dashboard/Chat

Dashboard/Chat/Chat.js

Chat component

The Chat component uses the `selectedConversationId` state field.

```
const Chat = () => {
  const selectedConversationId = useSelector(
    (state) => state.dashboard.selectedConversationId
  );
```

If there is no `selectedCoversionId`, it means we don't have any conversation yet, so display ChagGPT logo.

```
  return (
    <div className="chat_container">
      {!selectedConversationId ? (
        <ChatLogo />
      ) : (
```

When we find the `selectedCoversionId`, we display messages and new message input.

```
  <div className="chat_selected_container">
    <Messages />
    <NewMessageInput />
  </div>
```

Dashboard/Chat/NewMessageInput.js

This is the component to get an input from users.

```
const NewMessageInput = () => {
  const [content, setContent] = useState("");

  const dispatch = useDispatch();

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

We get `selectedConversationId` from the state field.

proceedMessage function

This is invoked when users pressed the "enter" key.

```
const [content, setContent] = useState("");
<input
  onKeyDown={handleKeyPressed}
  ...
>
const handleKeyPressed = (event) => {
  if (event.code === "Enter" && content.length > 0) {
    proceedMessage();
  }
};
```

From `onChange={(e) => setContent(e.target.value)}`, the state content stores the user input.

As a first step, we make a `message` object: notice the content (user input) is part of the message.

```
    const message = {
      aiMessage: false,
      content,
      id: uuid(),
      animate: false,
    };
```

The `selectedConversationId` state field is "new" when this is a new conversation, in this case, we create a new one, otherwise use existing one.

```
    const conversationId =
      selectedConversationId === "new" ? uuid() : selectedConversationId;
```

Then, trigger the `addMessag` action using the ID and message to store this conversation into the state field.

```
    dispatch(
      addMessage({
        conversationId,
        message,
      })
    );
```

Also, trigger the `setSelectedConversationId` to set the `conversationId` state field.

```
    dispatch(setSelectedConversationId(conversationId));
```

Finally, send the message using WebSocket to the server.

```
export const sendConversationMessage = (message, conversationId) => {
  socket.emit("conversation-message", {
    sessionId: localStorage.getItem("sessionId"),
    message,
    conversationId,
  });
```

```
  // send message to the server
    sendConversationMessage(message, conversationId);
    // reset value of the input
    setContent("");
  };
```

Dashboard/Chat/Messages.js

We get `selectedConversationId` and `conversations` state field from the dashboard slice state.

```
const { selectedConversationId, conversations } = useSelector(
  (state) => state.dashboard
);
```

Then, find the conversation with the `selectedConversationId` from `conversations`.

From the `conversation`, we return messages using the Message component.

```
  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>
  );
};
```