Step 7: Standalone ChatGPT Clone

Making Standalone ChatGPT Clone

Recaptures (Optional)

This section is to summarize the ideas from previous sections, you can skip any part of it if you are familiar with the ideas.

Recapture 1 - session, conversation, messages

When we use the ChatGPT clone, we have a session that has multiple question & answer set.

- question is the message and answer is the aiMessage, they become messages
    - messages: [message, aiMessage]
- one conversation is messages (question & answer) with ID
- one session is the converations with ID

Sessions, conversations, 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,
};
```

Recapture 2 - IndexedDB (localStorage)

We use IndexedDB to store local information; we can retrieve the information as it is saved in local DB.

We store the sessionId to store and get the current session.

```
localStorage.setItem("sessionId", sessionId);
localStorage.getItem("sessionId");
```

It is used for WebSocket APIs between the client and server.

WebSocket APIs used in functions

For WebSocket API "conversation-message" and "conversation-delete".

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

WebSocket API used for successful connection

These Websocket APIs "session-history" and "session-details" are used when the connection is made.

```
socket.on("connect", () => {
  socket.emit("session-history", {
    sessionId: localStorage.getItem("sessionId"),
  });

  socket.on("session-details", (data) => {
    const { sessionId, conversations } = data;

    localStorage.setItem("sessionId", sessionId);
    store.dispatch(setConversations(conversations));
  });
});
```

The conversations are stored in Slice states.

Recapture 3 - Redux Slice State Fields

The conversations are stored in the slice states; they are the storage in memory and can be accessed anywhere in the app.

Defining slice states

We need to define `initialState` and `reducers` to use slice states:

The initialState has state fields.

```
const initialState = {
  sessionEstablished: false,
  conversations: [],
  selectedConversationId: null,
};
const dashboardSlice = createSlice({
  name: "dashboard",
  initialState,
  reducers: { ... }
});
```

Reducers and Actions

The Reducers define actions that update the state fields:

They are used together with `dispatch` to update the slice state fields.

```
reducers: {
  setSelectedConversationId: (id) => ...
  addMessage: ...
  setConversations: ...
  ...
}
```

Using slice state fields

To update information in the slice state, we use dispatch:

In this example, we update the `selectedConversationId` with `id` using the action `setSelectedConversationId`.

```
const dispatch = useDispatch();

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

To get information from the slice state, we use `useSelector`:

In this example, we can access the conversations state field.

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

Recapture 4 - Asking Questions to ChatGPT API

1. We need all the previous conversations.
2. We get a new question (message) from the user.

1. We need all the previous conversations.
2. We get a new question (message) from the user.

- The second argument `message` has users' input.

```
const conversationMessageHandler = async (socket, data) => {
  const { sessionId, message, conversationId } = data;
  const previousConversationMessages = [];
  if (!sessions[sessionId]) return;
```

3. We combine all the previous conversaions and new question (message) and send it to ChatGPT API.

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

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

- This is the messages that are sent to ChatGPT.

```
messages: [
  ...previousConversationMessages,
  { role: "user", content: message.content },
],
```

4. We receive ChatGPT answer (aiMessage).

- This is the code that asks question and gets an answer.

```
    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;
```

5. We group (message, aiMessage, ID) to make a new conversation.

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

- get conversation with conversationId
    - If not found, make a new one

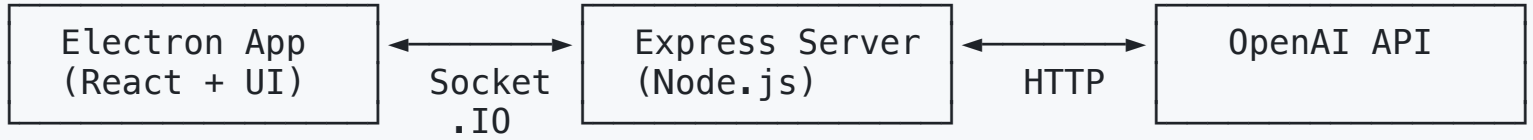
```
const conversation = sessions[sessionId].find((c) => c.id === conversationId);

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

- sessions[sessonId] already has the updated conversation from the previous actions, so find it and emit to the client.

```
  const updatedConversation
    = sessions[sessionId].find(
        (c) => c.id === conversationId);
  socket.emit("conversation-details",
    updatedConversation);
};
```

Old Architecture & Communication

Previous Architecture (with Express Server)



Components:

- Client: React app in Electron
- Server: Express server with Socket.IO
- Communication: WebSocket (socket.io-client)
  - localStorage is used to store `sessionId`
- Session Storage: Slice state fields (conversations)

Previous Communication

- We used sendConversationMessage function for the client)
- We used conversationMessageHandler for the server

Client function (socketConnection/scoketConn.js)

- It gets the sessionId from localStorage.
- Message is given from users.
- It gets the current conversationId.

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

Server function (socketServer.js)

This socketServer.js (server) connects to ChatGPT API to get the answer (refer to Recapture 4).

```
const conversationMessageHandler = async (socket, data) => { ...
  const aiMessage = {
    content: aiMessageContent,
    id: uuid(),
    aiMessage: true,
  };
  ...
  const updatedConversation =
    sessions[sessionId].find((c) => c.id === conversationId);
  socket.emit("onversation-details", updatedConversation);
};
```

The "conversation-details" WebSocket API is processed in the client side.

- The conversation is stored in the `conversations` state field.

```
socket.on("conversation-details", (conversation) => {
  store.dispatch(setConversationHistory(conversation));
});
```

New Architecture & Communication

New Architecture (Serverless)

Components:

- Client: React app in Electron (calls API directly)
- Communication: Direct HTTP calls to OpenAI
- Session Storage: localStorage
    - We use all the conversations in the localStorage

New Communication: Direct Connection to ChatGPT

Async Communication to the ChatGPT server

As we don't use the server for ChatGPT server communication, the client directly connects to the ChatGPT API asynchrously.

- We need to update `Dashboard/dashboard.jsx` extensively to make this change.

Update in createSlice: createAsyncThunk

To use async function in the Redux Slice, we should use `createAsyncThunk` function.

```
export const sendConversationMessage = createAsyncThunk(
  'dashboard/sendMessage',
  async ({ message, conversationId, conversationMessages }) => { ... }
);
```

In this function, we get the previous conversations, add new question from the user and get AI response - the same actions previously in the server.

```
// Build message history for OpenAI
const messages = conversationMessages.map(m => ({
  role: m.aiMessage ? 'assistant' : 'user',
  content: m.content
}));

// Add new user message
messages.push({ role: 'user', content: message.ontent });

// Get AI response
const aiContent = await sendMessageToAI(messages);

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

return { message, aiMessage, conversationId };
```

Update in createSlice: Added State fields

We need to add two state fields to indicate the communication status:

- loading shows that we are waiting for the ChatGPT answers.
- error shows the error in the communication with the ChatGPT.

```
const initialState = {
  conversations: [],
  selectedConversationId: null,
  loading: false, // <--
  error: null, // <--
};
```

Update in createSlice: extraReducers

The `createAsyncThunk` function should have the matching functions to process the cases when the action is waiting (pending), success (fulfilled), and failure (rejected).

waiting (pending)

In this case, we set the loading filed true.

```
.addCase(sendConversationMessage.pending, (state) > {
  state.loading = true;
  state.error = null;
})
```

success (fulfilled)

In this case, we get (message, aiMessage, conversationId) from the `createAsyncThunk`.

```
.addCase(sendConversationMessage.fulfilled, state, action) => {
  const { message, aiMessage, conversationId } = action.payload;
```

We find the conversation from the conversationId, update the conversation, and store the conversations in the localStorage.

```
  const conversation = state.conversations.find(c => c.id === conversationId);
  if (conversation) {
    conversation.messages.push(aiMessage);
  }

  state.loading = false;

  // Save to localStorage
  localStorage.setItem('conversations', JSON.stringify(state.conversations));
})
```

failure (rejected).

In this case, we set the error message, and add error message to the conversation.

```
.addCase(sendConversationMessage.rejected, (state, ction) => {
  state.loading = false;
  state.error = action.error.message;

  // Add error message to conversation
  const { conversationId } = action.meta.arg;
  const conversation = state.conversations.find(c => c.id === conversationId);

  if (conversation) {
    conversation.messages.push({
      content: "Sorry, I couldn't process your message. Please try again.",
      id: uuid(),
      aiMessage: true,
      error: true,
    });
  }
});
```

Update Dashboard/Sidebar

DeleteConversationsButton.jsx

Before, we need to remove the state filed (we rename it to avoid name confliction):

```
import { deleteConversations as deleteConversationsFromStore } from "../dashboardSlice";
dispatch(deleteConversationsFromStore([]));
```

Then, use deleteConversations() in the `socketConnection/socketConn.js` to remove conversations in the server.

```
  const handleDeleteConversations = () => {
    dispatch(deleteConversationsFromStore([]));
    deleteConversations();
  };
```

Now, we don't need the server communication:

We remove the state filed (no need to name change due to the conflict), and remove localStorage.

```
const handleDeleteConversations = () => {
  // Clear from localStorage
  localStorage.removeItem('conversations');

  // Clear from Redux store
  dispatch(deleteConversations());
};
```

Update Dashboard/Chat

Chat/NewMessageInput.js

loading state field

We need the loading state field:

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

This is used in many functions to process only when loading is finished.

```
  const handleSendMessage = () => {
    if (content.length > 0 && !loading) {
      proceedMessage();
    }
  };
```

```
  const handleKeyPressed = (event) => {
    if (event.code === "Enter" && content.length > 0 && !loading) {
      proceedMessage();
    }
  };
```

For the input, code uses the loading state filed to change its behavior.

```
<div className="new_message_input_container">
  <input
    className="new_message_input"
    placeholder={loading ? "Waiting for response..." : "Send a message ..."} <--
    value={content}
    onChange={(e) => setContent(e.target.value)}
    onKeyDown={handleKeyPressed}
    disabled={loading} <--
  />
  <div className="new_message_icon_container" onClick={handleSendMessage}>
    <BsSend color={loading ? "lightgrey" : "grey"} /> <--
  </div>
</div>
```

Reading OPENAI_API_KEY from .env

Two versions of OPENAI_API_KEYs

With `npm run electron-dev`, we use Chrome as the frontend UI, and with `npm run electron-prod`, we use Electron:

- We should have two different KEY:
  - One for VITE (VITE_OPENAI_API_KEY.)
  - The other for Electron (OPENAI_API_KEY)

main.js (Electron)

We need the renderer to use the OPENAI_API_KEY, but we should not allow the renderer to access it directly.

- The main process reads the .env file and handle the API key request from the renderer.

```
require('dotenv').config(); // Load .env at startup

// Handle API key request from renderer
ipcMain.handle('get-api-key', () => {
  return process.env.OPENAI_API_KEY || null;
});
```

preload.js (Electron)

In proeload.js, we expose protected methods that allow the renderer process to use: the ipcRenderer without exposing the entire object

```
contextBridge.exposeInMainWorld('electron', {
  getApiKey: () => ipcRenderer.invoke('get-api-key')
});
```

services/openaiService.js

The functions for accessing ChatGPT are in the openaiServices.js.

initializeOpenAI

We create a `openaiClient` object from the configuration.

- We need to detect if we're running inside Electron (with preload exposing window.electron) and access OPENAI_API_KEY or VITE_OPENAI_API_KEY.

```
export const initializeOpenAI = async () => {
  try {

    const isElectron =
      typeof window !== "undefined" &&
      window.electron &&
      typeof window.electron.getApiKey === "function";
```

Then, we need to create `openaiClient` from the configuration.

```
    if (!apiKey) {
      console.error("Missing OPENAI_API_KEY / VITE_OPENAI_API_KEY in .env");
      return false;
    }

    // openai@3.3.0 uses Configuration + OpenAIApi (not `new OpenAI(...)`)
    const configuration = new Configuration({
      apiKey,
    });

    openaiClient = new OpenAIApi(configuration);
    return true;
  } catch (error) {
    console.error("Failed to initialize OpenAI:", error);
    return false;
  }
};
```

sendMessageToAI

We use OpenAI ver 3.3 functions:

```
export const sendMessageToAI = async (messages) => {
  if (!openaiClient) {
    throw new Error("OpenAI client not initialized");
  }

  try {
    const response = await openaiClient.createChatCompletion({
      model: "gpt-3.5-turbo",
      messages: messages,
    });

    console.log("OpenAI raw response choice:", response.data.choices[0]);

    const aiMessageContent =
      response?.data?.choices?.[0]?.message?.content ??
      "No response from AI";

    return aiMessageContent;
  } catch (error) {
    console.error("OpenAI API error:", error);
    throw new Error("Failed to get response from AI");
  }
};
```

Using the sendMessageToAI

Send message to ChatGPT

To send message to ChatGPT, we store conversationId to the state field.

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

> Before, we used this function to notify the coversationId to the server:
>
> ```
> sendConversationMessage(message, conversationId);
> ```

Then, we process the selection of current messages using Redux slice state reducer: sendConversationMessage.

1. we need to find conversations from the conversationId.

```
// Get conversation messages for context
const conversation = conversations.find(c => c.id == conversationId);
const conversationMessages = conversation ? onversation.messages : [];
// Send message to AI
dispatch(sendConversationMessage({
  message,
  conversationId,
  conversationMessages
}));
```

2. This function invokes the `sendMessageToAI` asynchronously.

```
export const sendConversationMessage = createAsyncThunk(
  async ({ message, conversationId, conversationMessages }) => {
    // Build message history for OpenAI
    ...
    // Get AI response
    const aiContent = await sendMessageToAI(messages);
```

Run

The same as step 6.

Key Changes Summary

Removed

- ❌ Express server (`server/` directory)
- ❌ Socket.IO server and client
- ❌ `socketConnection/` folder
- ❌ Server-side session management
- ❌ No Loading Spnner

Added

- ✅ `services/openaiService.js` - Direct OpenAI API calls
- ✅ Redux async thunks in `dashboardSlice.js`
- ✅ Secure API key handling through Electron IPC
- ✅ localStorage for conversation persistence

Modified

- 🔄 `main.js` - Added IPC handler for API key
- 🔄 `preload.js` - Exposed electron API to renderer
- 🔄 `App.js` - Initializes OpenAI client instead of socket
- 🔄 `dashboardSlice.js` - Added `createAsyncThunk` for API calls
- 🔄 `NewMessageInput.js` - Dispatches thunk instead of socket emit
- 🔄 `DeleteConversationsButton.js` - Only updates localStorage and Redux

Benefits

1. Simpler Architecture: No separate server to manage
2. Fewer Dependencies: Removed socket.io, Express
3. Direct Communication: Lower latency, simpler debugging
4. Client-side Storage: Conversations persist locally
5. Easier Deployment: Single Electron app to distribute

Security

- API key stored in `.env` file
- Accessed only by Electron's main process
- Passed to renderer via secure IPC channel
- Never exposed in client-side code