# Step 4: Sending Messages from Server

## Server

Now, we implement socketServer's conversationMessageHandler function; this function gets the `message` and `Ids` from client, gets the `aiMessag` and return a new conversation to the client.

## src/socketServer.js

### conversationMessageHandler function

In this function, `sessionId, message, conversationId` are given as arguments.

```
const conversationMessageHandler = (socket, data) => {
  const { sessionId, message, conversationId } = data;
```

Then, from the `sessionId`, we find the conversation with `conversationId`.

```
  if (sessions[sessionId]) {
    const aiMessage = {
      content: "Hello here is AI",
      id: uuid(),
      aiMessage: true,
    };

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

For now, we don't use ChatGPT API, so we use "Hello here is AI" to mimic ChatGTP output.

If the conversation is found, we push the conversation to existing conversations with new messages.

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

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

Then emit the updated conversation using the `conversation-details` API to the client.

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

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

### conversationDeleteHandler function

This function deletes the conversations that are attached to `sessionId`.

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

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

module.exports = { registerSocketServer };
```

# Client

We need to add two reducers and actions:

## Dashboard/dashboardSlice.js

The `setConversationHistory` reducer finds the conversation with `id` and update the messages with the received messages, if not found, we create a new one.

```
setConversationHistory: (state, action) => {
  const { id, messages } = action.payload;
  const conversation = state.conversations.find((c) => c.id === id);
  if (conversation) {
    conversation.messages = messages;
  } else {
    state.conversations.push({ id, messages,});
  }
},
```

Using this reducer, we can update the messages of a

The `deleteConversations` reducer removes the conversations and set `selectedConversationId` null.

```
    deleteConversations: (state) => {
      state.conversations = [];
      state.selectedConversationId = null;
    },
```

## socketConnection/socketConn.js

The `conversation-details` WebSocket API is processed in this module.

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

The received `conversation` is stored in the state field using the `setConversationHistory` action.

We need to update the `sendConversationMessage` and `deleteConversations`, as we get the current `sessinId` from localStorage.

```
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"),
  });
};
```