This document provides a detailed breakdown of the `index.html` file, explaining how its HTML, CSS, and JavaScript work together to create a functional chat interface that meets all your assignment requirements.

1. HTML Structure (The Skeleton)

The HTML provides the basic layout and elements of the chat application.

- `<head>` :
  - Imports the "Inter" font from Google Fonts to match the modern UI style.
  - Contains all the CSS within a `<style>` tag, as requested instead of SCSS.
- `<body>` :
  - Uses Flexbox (`display: flex`) to center the main chat window vertically and horizontally on the page.
- `.chat-container` :
  - This is the main white box that holds the entire chat UI. It uses Flexbox in a `column` direction to stack the header, message list, and input box on top of each other.
- `.chat-header` :
  - The top bar containing the title (`<h3>ZUS AI Agent</h3>`) and the `/reset` button.
- `.message-list` :
  - The main, scrollable area where messages appear.
  - `flex-grow: 1` is the key property that makes this element expand to fill all available space, pushing the composer to the bottom.
  - `overflow-y: auto` makes it scrollable only if the messages overflow.
- `.composer-container` :
  - A wrapper that holds both the chat input and the (initially hidden) autocomplete box.

    - `position: relative` is crucial for allowing the autocomplete box to be positioned *above* it.
- `.autocomplete-box`:
    - Positioned `absolute` to float on top of the UI, just above the input.
    - `display: none` hides it by default.
- `.chat-composer` **(the** `<form>` **):**
    - The form at the bottom containing the `textarea` and the "Send" `button`.
- `<textarea id="message-input">`:
    - The multiline input field. `rows="1"` sets its initial height to one line, and it grows automatically using JavaScript.
- **SVG Icon (** `<svg id="icon-tool">` **)**
    - An SVG "tool" icon is defined at the bottom. This is used in the JavaScript to visually represent agent steps without needing an external image file.

2. CSS Styling (The Appearance)

The CSS inside the `<style>` tag styles the HTML to look like your design.

- `:root`: A best-practice way to define CSS variables (like a theme). This makes it easy to change colors (e.g., `--brand-color`) in one place.
- `.message`:
    - This is the base class for a chat bubble wrapper. It uses `display: flex` to put the avatar and the message content side-by-side.
- `.message.user` & `.message.bot`:
    - These "modifier" classes change the styling.
    - `.message.user` uses `align-self: flex-end` to move the message to the **right**.
    - `.message.bot` uses `align-self: flex-start` to keep it on the **left**.

    - They also apply different background colors and `border-radius` to create the "chat bubble" tail effect.
- `.agent-steps` & `.step`:
    - This is the visualization for your agent's plan.
    - `.agent-steps` is a container with a `border-left` that creates the vertical line, connecting the steps.
    - `.step` is the box for a single tool call, using flexbox to align the icon, name, and (most importantly) the `<pre>` tag.
    - `<pre>` **tag:** This is used to display the tool's JSON arguments. It's styled to look like a code block and uses `white-space: pre-wrap` and `word-break: break-all` to ensure long JSON strings wrap neatly instead of breaking the UI.
- `.composer-container` & `.chat-composer`:
    - The `<textarea>` is styled to auto-resize. Its `max-height` is set to 6 lines to prevent it from growing and taking over the whole screen.
- `.autocomplete-box`:
    - Styled to appear as a floating box above the composer.
    - The `.selected` class, which is managed by JavaScript, adds a highlight to the currently selected autocomplete item.

3. JavaScript Logic (The Brain)

This is the most complex part. It's wrapped in a `DOMContentLoaded` listener to ensure it only runs after all the HTML is loaded.

Core State & Config

- `messages = []`: An array that holds the entire chat history as JavaScript objects. This is your "source of truth."
- `SESSION_ID`: A unique ID generated on page load, sent to the backend with each request.
- `BACKEND_URL`: The URL for your FastAPI server.
- `QUICK_ACTIONS`: An array of objects defining your `/` slash commands.

Key Functions

1. `loadChatHistory()` & `saveChatHistory()` **(Requirement Fulfilled)**
   - `loadChatHistory()` is called on page load. It checks `localStorage` for `"chatHistory"`.
   - If history exists, it parses the JSON and calls `renderMessage()` for each saved message.
   - If no history exists, it creates a new welcome message.
   - `saveChatHistory()` is called after *any* change to the `messages` array (adding a user message, receiving a bot response). It uses `JSON.stringify()` to save the array to `localStorage`.
2. `addMessage(content, role)` & `renderMessage(message)`
   - `addMessage` is a helper function. It creates a new message object (with an ID, timestamp, etc.), adds it to the `messages` array, and then calls `renderMessage`.
   - `renderMessage` is the workhorse. It creates the HTML elements for a message, including the avatar, author, and timestamp.
   - It correctly handles rendering messages from history, including recreating the `steps` and `finalAnswer` from the saved data.
   - It creates an empty `<p>` tag with a loading spinner for new bot messages, which will be filled in by the streaming function.
3. `handleSubmit(e)`
   - Triggered by pressing "Enter" or clicking the Send button.
   - It gets the text, adds the user's message (e.g., `addMessage("Hi", "user")`), and then saves the history.
   - It clears the input text.
   - It creates the empty "bot" message shell (e.g., `addMessage("", "bot")`).
   - Finally, it calls `streamAgentResponse()`, passing in the user's text and the bot message data.
4. `streamAgentResponse(text, botMessageData)` **(The Core Logic)**

- This function uses the modern `fetch` API to make the `POST` request to your backend.
- It gets a `ReadableStream` from the `response.body` and uses a `TextDecoder` to read the incoming data in chunks.
- **Stream Parsing:** This is the most complex part. Your backend sends a stream of `<step>...</step>` tags.
  - It uses a `buffer` to store incoming text.
  - It repeatedly checks the buffer for a *complete* pair of `<step>` and `</step>` tags.
  - When it finds a complete step, it parses the content *inside* that step using string matching (`.match(/<step_name>(.*?)<\/step_name>/)`) to find the tool name and the JSON arguments.
  - **Visualizing Steps:** If the `stepName` is **not** `final_answer`, it calls `createStepElement()` to build the visual step and appends it to the bot's message. It also saves this step to the `botMessageData.steps` array.
  - **Final Answer:** If the `stepName` **is** `final_answer`, it parses the JSON, extracts the `answer`, and puts that text into the bot's final `<p>` tag.
- **Error Handling:** A `try...catch` block wraps the entire fetch request. If the API fails (like your HTTP 500 test), it will catch the error and display a graceful error message in the chat.
- **Saving:** After the stream is complete (`done` is true), it calls `saveChatHistory()` one last time to save the *complete* bot response (with all its steps and the final answer).

5. **Autocomplete & Input Handlers (Requirement Fulfilled)**
   - `handleInput(e)`: Fires on every keystroke.
     - It auto-resizes the textarea's height based on its content (`scrollHeight`).
     - If the text starts with `/`, it filters the `QUICK_ACTIONS` list and calls `showAutocomplete()`.
   - `handleKeydown(e)`:
     - Handles "Enter-to-send" and "Shift+Enter-for-newline".
     - Handles "ArrowUp" and "ArrowDown" to navigate the autocomplete menu by changing `autocompleteIndex` and updating the `.selected` class.

    - Handles "Enter" to select an item from the autocomplete menu.
- `handleReset()` : Clears the `messages` array, clears `localStorage` , and clears the UI.