# WEB PROJECT

```
const typingForm = document.querySelector(".typing-form");  // Form select k
const chatContainer = document.querySelector(".chat-list"); // Chat ka contai
const suggestions = document.querySelectorAll(".suggestion"); // Suggested
const toggleThemeButton = document.querySelector("#theme-toggle-button"
const deleteChatButton = document.querySelector("#delete-chat-button"); //
```

```
let userMessage = null;  // User ke message ko store karne ke liye
let isResponseGenerating = false;  // Track karne ke liye ki chatbot response g
```

**userMessage**: Yeh user ka **input message** store karega.
i
**sResponseGenerating**: Yeh **track karega** ki chatbot response generate kar raha hai ya nahi (taaki  duplicate requests na ho).

```
const API_KEY = "AIzaSyCcOk42FCX_z8b7bRTCtR3qEa--eXTFNT8"; // Your A
const API_URL = `https://generativelanguage.googleapis.com/v1beta/models/
```

**API_KEY** → Google Gemini API ka **authentication key** hai.

**API_URL** → API ka URL hai **jo AI responses fetch karega**.

```
const loadDataFromLocalstorage = () => {
  const savedChats = localStorage.getItem("saved-chats");
  const isLightMode = (localStorage.getItem("themeColor") === "light_mode")
```

```
  document.body.classList.toggle("light_mode", isLightMode);
  toggleThemeButton.innerText = isLightMode ? "dark_mode" : "light_mode";

  chatContainer.innerHTML = savedChats || '';
  document.body.classList.toggle("hide-header", savedChats);
  chatContainer.scrollTo(0, chatContainer.scrollHeight);
}
```

Certainly! Here's a formatted and easy-to-read version of your explanation:

---

## `loadDataFromLocalstorage()`

This function retrieves data from local storage and applies it to the webpage. Below is a breakdown of its components:

1. **Retrieve Chat History**
   - `localStorage.getItem("saved-chats")`: Retrieves the chat history stored under the key `"saved-chats"`.
     - Returns `null` if the key does not exist.
2. **Check Theme Mode**
   - `localStorage.getItem("themeColor") === "light_mode"`: Verifies if the stored theme is set to `"light_mode"`.
3. **Toggle Light Mode**
   - `document.body.classList.toggle("light_mode", isLightMode)`: Adds or removes the `"light_mode"` class from the `<body>` element depending on the `isLightMode` value.
     - Likely controls the theme's appearance.
4. **Update Theme Toggle Button**
   - `toggleThemeButton.innerText = isLightMode ? "dark_mode" : "light_mode"`: Updates the text of a theme toggle button to reflect the current theme.
     - Displays `"dark_mode"` for light mode and `"light_mode"` for dark mode.
5. **Set Chat History**

- `chatContainer.innerHTML = savedChats || ''`: Sets the content of the `chatContainer` element to the loaded chat history.
    - Uses `|| ''` as a fallback to prevent errors when `savedChats` is `null`.

6. **Toggle Header Visibility**
    - `document.body.classList.toggle("hide-header", savedChats)`: Toggles the `"hide-header"` class on the `<body>` based on the existence of `savedChats`.
        - If chat data exists, the header is hidden.
        - Logic assumes a falsy `savedChats` indicates no chats are present. This may require further review.
7. **Scroll Chat to Bottom**
    - `chatContainer.scrollTo(0, chatContainer.scrollHeight)`: Ensures the `chatContainer` scrolls to the bottom, showing the latest messages

```
const createMessageElement = (content, ...classes) => {
  const div = document.createElement("div");
  div.classList.add("message", ...classes);
  div.innerHTML = content;
  return div;
}
```

## `createMessageElement` Function

This function creates a dynamic `<div>` element representing a chat message. Here's the breakdown:

1. **Function Declaration**
    - `const createMessageElement = (content, ...classes) => { ... }`: Declares a constant function named `createMessageElement` with two arguments:
        - `content`: A string containing the chat message text (e.g., "Hello!", "How are you?"). This becomes the inner content of the `<div>`.

- `...classes`: Uses the rest parameter syntax (`...`) to allow multiple additional arguments, treated as an array of CSS class names. Useful for dynamically applying styles (e.g., message type).

2. **Create `<div>` Element**
   - `const div = document.createElement("div");`:
     Creates a new `<div>` element using `document.createElement()`.
3. **Add CSS Classes**
   - `div.classList.add("message", ...classes);`:
     Adds the `"message"` class (base styling) and any additional classes passed through `...classes`.
     - Example: `createMessageElement("Hello", "incoming")` will add `"message"` and `"incoming"` classes to the `<div>`.
     - Example: `createMessageElement("Hello", "incoming", "unread")` will add `"message"`, `"incoming"`, and `"unread"` classes.
4. **Set Message Content**
   - `div.innerHTML = content;`:
     Sets the inner HTML of the `<div>` to the `content` string, inserting the chat message text.
5. **Return the `<div>`**
   - `return div;`:
     Returns the fully created and styled `<div>` element, ready to be inserted into the chat interface.

The `createMessageElement` function is an efficient way to create reusable chat message elements. Its use of the rest parameter (`...classes`) provides flexibility, allowing for easy customization of message appearance and behavior based on attributes like sender or read status.

```
const showTypingEffect = (text, textElement, incomingMessageDiv) => {
  const words = text.split(' ');
  let currentWordIndex = 0;

  const typingInterval = setInterval(() => {
    textElement.innerText += (currentWordIndex === 0 ? '' : ' ') + words[current
    incomingMessageDiv.querySelector(".icon").classList.add("hide");

    if (currentWordIndex === words.length) {
      clearInterval(typingInterval);
      isResponseGenerating = false;
      incomingMessageDiv.querySelector(".icon").classList.remove("hide");
      localStorage.setItem("saved-chats", chatContainer.innerHTML);
    }

    chatContainer.scrollTo(0, chatContainer.scrollHeight);
  }, 75);
}
```

## `showTypingEffect` Function

This function creates an animation effect simulating typing, where each word of a given text appears at a specified time interval. Below is the breakdown:

1. **Function Declaration**
   - `const showTypingEffect = (text, textElement, incomingMessageDiv) => { ... }`: Declares a constant function `showTypingEffect` with the following parameters:
     - `text`: The full text to be displayed word by word.
     - `textElement`: The target HTML element where the text is to be displayed.
     - `incomingMessageDiv`: The HTML element containing the message, used to control animations and effects.

2. **Split Text into Words**
   - `const words = text.split(' ');`:
     Splits the `text` into an array of words using a space (`' '`) as the delimiter.
3. **Initialize Word Index**
   - `let currentWordIndex = 0;`:
     Keeps track of the current word being displayed, starting from the first word.
4. **Set Typing Interval**
   - `const typingInterval = setInterval(() => { ... }, 75);`:
     Sets up a repeating interval to display each word every 75 milliseconds.
5. **Update Text Element**
   - `textElement.innerText += (currentWordIndex === 0 ? '' : ' ') + words[currentWordIndex++];`:
     Appends the next word in the array to the `textElement`, with a space unless it's the first word.
6. **Hide Typing Icon**
   - `incomingMessageDiv.querySelector(".icon").classList.add("hide");`:
     Hides the typing indicator (e.g., a loading icon) during the animation.
7. **Check Typing Completion**
   - `if (currentWordIndex === words.length) { ... }`:
     Checks if all words have been displayed:
     - `clearInterval(typingInterval);`: Stops the typing animation once complete.
     - `isResponseGenerating = false;`: Updates a flag indicating that the response is ready.
     - `incomingMessageDiv.querySelector(".icon").classList.remove("hide");`: Shows the typing icon again.
     - `localStorage.setItem("saved-chats", chatContainer.innerHTML);`: Saves the chat content to local storage.
8. **Scroll to Bottom**
   - `chatContainer.scrollTo(0, chatContainer.scrollHeight);`:
     Ensures the chat interface scrolls to display the latest message.

The `showTypingEffect` function creates an engaging "typing" animation by displaying text word by word at a set interval. It dynamically updates the content of a target element and manages animations like showing or hiding a typing icon. Additionally, it ensures the chat is scrolled to the latest message and saves the chat history to local storage for persistence.

```
const generateAPIResponse = async (incomingMessageDiv) => {
  const textElement = incomingMessageDiv.querySelector(".text");

  try {
    const response = await fetch(API_URL, {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({
        contents: [{ role: "user", parts: [{ text: userMessage }] }]
      }),
    });

    const data = await response.json();
    if (!response.ok) throw new Error(data.error.message);

    const apiResponse = data.candidates[0].content.parts[0].text.replace(/\*\*(
    showTypingEffect(apiResponse, textElement, incomingMessageDiv);
  } catch (error) {
    isResponseGenerating = false;
    textElement.innerText = error.message;
    textElement.parentElement.closest(".message").classList.add("error");
  } finally {
    incomingMessageDiv.classList.remove("loading");
  }
}
```

## `generateAPIResponse` Function

This asynchronous function sends a user's message to an API, fetches the response, and handles errors if they occur. Here's a detailed breakdown:

1. **Function Declaration**
   - `const generateAPIResponse = async (incomingMessageDiv) => { ... }`: Declares an asynchronous function that accepts `incomingMessageDiv` as its parameter:
     - `incomingMessageDiv`: The HTML element representing the incoming chat message.
2. **Identify the Text Element**
   - `const textElement = incomingMessageDiv.querySelector(".text");`: Retrieves the `.text` element within `incomingMessageDiv` where the response will be displayed.
3. **Send API Request**
   - The `fetch()` method is used to send a POST request to the `API_URL`:
     - `method: "POST"`: Specifies the HTTP method.
     - `headers: { "Content-Type": "application/json" }`: Ensures the request body is in JSON format.
     - `body`: Sends the user's message as a JSON object.
4. **Process the API Response**
   - `const data = await response.json();`: Converts the API's response into a JSON object.
   - `if (!response.ok) throw new Error(data.error.message);`: Checks if the response is not successful and throws an error if there's an issue.
5. **Extract and Format the API Response**
   - `data.candidates[0].content.parts[0].text`: Accesses the response text from the API's JSON structure.
   - `.replace(/\*\*(.*?)\*\*/g, '$1');`: Removes Markdown-style bold formatting (`*text**`) from the response.

6. **Display the Typing Effect**
    - `showTypingEffect(apiResponse, textElement, incomingMessageDiv);`: Calls the `showTypingEffect` function to display the API's response with a typing animation.
7. **Error Handling**
    - `catch (error) { ... }`: Handles any errors that occur during the request or response processing:
        - `isResponseGenerating = false;`: Updates the flag indicating the response is no longer being generated.
        - `textElement.innerText = error.message;`: Displays the error message in the `textElement`.
        - `textElement.parentElement.closest(".message").classList.add("error");`: Adds an `error` class to style the error message.
8. **Finalize the Process**
    - `finally { ... }`: Ensures that, regardless of success or error:
        - `incomingMessageDiv.classList.remove("loading");`: Removes the "loading" state from the message.

The `generateAPIResponse` function sends a user's message to an API, processes the response, and displays it dynamically with a typing effect. It also manages errors gracefully by showing an error message and updating the UI accordingly. This function ensures a smooth and interactive chat experience.

```
const showLoadingAnimation = () => {
  const html = `<div class="message-content">
            <img class="avatar" src="images/gemini.svg" alt="Gemini avatar">
            <p class="text"></p>
            <div class="loading-indicator">
              <div class="loading-bar"></div>
              <div class="loading-bar"></div>
```

```
            <div class="loading-bar"></div>
          </div>
        </div>`;
  const incomingMessageDiv = createMessageElement(html, "incoming", "loa
  chatContainer.appendChild(incomingMessageDiv);
  chatContainer.scrollTo(0, chatContainer.scrollHeight);
  generateAPIResponse(incomingMessageDiv);
}
```

## `showLoadingAnimation` Function

This function creates a loading animation in the chat interface to indicate that a response is being generated. Here's how it works:

1. **Define HTML for the Loading Animation**
   - `const html = ...`:
     Creates a string containing the HTML structure for the loading animation. The structure includes:
     - A `<div>` with the class `message-content` to group the elements.
     - An `<img>` element with the class `avatar` to display the "Gemini" avatar image.
     - A `<p>` element with the class `text`, acting as a placeholder for the chat message text.
     - A `<div>` with the class `loading-indicator`, containing three child `<div>` elements representing loading bars.
2. **Create Incoming Message Element**
   - `const incomingMessageDiv = createMessageElement(html, "incoming", "loading");`:
     Calls the `createMessageElement` function to create a message element with the specified HTML and classes:
     - `"incoming"`: Indicates it's an incoming message.
     - `"loading"`: Adds a loading state to the message.

3. **Add Message to Chat Container**
    - `chatContainer.appendChild(incomingMessageDiv);`:
      Appends the newly created `incomingMessageDiv` to the `chatContainer`, which holds all the chat messages.
4. **Scroll to Latest Message**
    - `chatContainer.scrollTo(0, chatContainer.scrollHeight);`:
      Scrolls the chat container to the bottom, ensuring the loading animation is visible.
5. **Trigger API Response**
    - `generateAPIResponse(incomingMessageDiv);`:
      Calls the `generateAPIResponse` function, passing `incomingMessageDiv` as an argument, to initiate the API request and handle the response.

The `showLoadingAnimation` function introduces a loading animation into the chat interface while the system prepares a response. It dynamically generates and appends a message element with an avatar, text placeholder, and animated loading bars. Once the response is generated, the loading animation is replaced with the actual chat message.

```
const copyMessage = (copyButton) => {
  const messageText = copyButton.parentElement.querySelector(".text").inner
  navigator.clipboard.writeText(messageText);
  copyButton.innerText = "done";
  setTimeout(() => copyButton.innerText = "content_copy", 1000);
}
```

## `copyMessage` Function

This function allows users to copy a message's text to the clipboard and provides a visual confirmation via a button label. Here's how it works:

1. **Retrieve Text Content**
   - `const messageText = copyButton.parentElement.querySelector(".text").innerText;`:
     Fetches the inner text from the `.text` element, which is assumed to be a sibling of the `copyButton` inside a parent container.
2. **Write to Clipboard**
   - `navigator.clipboard.writeText(messageText);`:
     Uses the `Clipboard API` to copy the retrieved message text to the clipboard.
3. **Provide Feedback**
   - `copyButton.innerText = "done";`:
     Updates the button's text to `"done"` to indicate that the copy action was successful.
4. **Reset Button Text**
   - `setTimeout(() => copyButton.innerText = "content_copy", 1000);`:
     Sets a timeout of 1 second to change the button text back to `"content_copy"`, restoring its original label.

---

The `copyMessage` function simplifies text copying by enabling users to quickly copy content with a button click and receive immediate visual feedback. The `Clipboard API` ensures seamless integration, while the timeout restores the button's state for consistent user interaction.

```
const handleOutgoingChat = () => {
  userMessage = typingForm.querySelector(".typing-input").value.trim() ||
  if(!userMessage || isResponseGenerating) return;

  isResponseGenerating = true;
  const html = `<div class="message-content">
                <img class="avatar" src="images/user.jpg" alt="User avatar">
                <p class="text"></p>
              </div>`;
```

```
  const outgoingMessageDiv = createMessageElement(html, "outgoing");
  outgoingMessageDiv.querySelector(".text").innerText = userMessage;
  chatContainer.appendChild(outgoingMessageDiv);

  typingForm.reset();
  document.body.classList.add("hide-header");
  chatContainer.scrollTo(0, chatContainer.scrollHeight);
  setTimeout(showLoadingAnimation, 500);
}
```

## `handleOutgoingChat` Function

This function handles the outgoing chat message submitted by the user, updates the UI, and prepares for the response generation. Here's how it works:

1. **Retrieve and Validate User Input**
   - `userMessage = typingForm.querySelector(".typing-input").value.trim() || userMessage;`:
     Retrieves the value of the input field (`.typing-input`), trims unnecessary whitespace, and assigns it to `userMessage`.
     - If the input is empty, it uses the existing `userMessage` value.
   - `if (!userMessage || isResponseGenerating) return;`:
     Prevents further execution if the message is empty or if a response is already being generated.
2. **Set Response State**
   - `isResponseGenerating = true;`:
     Updates the flag to indicate that a response is being processed.
3. **Create HTML for Outgoing Message**
   - `const html = ...`:
     Defines the HTML structure for an outgoing message:
     - A `<div>` with the class `message-content` containing:
       - An `<img>` element with a `user` avatar.
       - A `<p>` element with the class `text` for the message text.

4. **Create Outgoing Message Element**
   - `const outgoingMessageDiv = createMessageElement(html, "outgoing");`:
     Calls the `createMessageElement` function to generate an HTML element for the outgoing message with the `"outgoing"` class.
   - `outgoingMessageDiv.querySelector(".text").innerText = userMessage;`:
     Sets the message text inside the `outgoingMessageDiv`.
5. **Append Message to Chat Container**
   - `chatContainer.appendChild(outgoingMessageDiv);`:
     Adds the outgoing message element to the chat container.
6. **Reset Input Form**
   - `typingForm.reset();`:
     Clears the input field (`.typing-input`) for the next message.
7. **Hide Header**
   - `document.body.classList.add("hide-header");`:
     Adds the `"hide-header"` class to the `<body>`, possibly to declutter the UI.
8. **Scroll to Latest Message**
   - `chatContainer.scrollTo(0, chatContainer.scrollHeight);`:
     Scrolls the chat container to the bottom to display the newly added message.
9. **Show Loading Animation**
   - `setTimeout(showLoadingAnimation, 500);`:
     Delays the execution of the `showLoadingAnimation` function by 500 milliseconds, simulating a realistic typing delay before the system prepares a response.

The `handleOutgoingChat` function processes the user's message, dynamically updates the chat interface with an outgoing message element, and triggers a loading animation while awaiting the system's response. It ensures the chat feels interactive and provides a smooth user experience.