React/Typescript NLIP Frontend Documentation

Overview

The React/Typescript application serves as a frontend that uses NLIP to interact with the NLIP backend. It supports both textual and binary input (e.g., images) and converts user inputs into a structured NLIP message format before forwarding them to the backend and subsequently receiving a response.

Get Started

Prerequisites

- **Node/npm**: Ensure you have Node/npm installed.

1. **Clone the Repository**
   `git clone https://github.com/nlip-project/nlip_client_vite_ts.git`

2. **Install Dependencies**
   Installs into node-modules from package.json:
   `npm i`

3. **Run the Application**
   Launch the frontend:
   `npm run dev`

   a. **Vite**
   `npm run dev` builds the application and launches the frontend through a Vite development server running on port 5173.

4. **Access the Interface**
   Open your browser and navigate to the URL displayed in the terminal (e.g., http://localhost:5173).

Application Workflow

1. **Input Parsing**
   - Users interact with the frontend to provide inputs.
   - The application retrieves textual content and, if present, base64 encodings for any uploaded binary files (e.g., images).

2. **Message Creation**
   - User inputs are processed into a NLIP structured format defined under the `Message` interface in `message.ts`.
   - Text inputs populate the `format,subformat,content` fields
   - Binary inputs are encoded in base64 and added to the `submessages` field of a text NLIP message.

3. **Serialization**
   - The structured message is serialized into a JSON-compatible format using the `JSON.stringify()` function.

Text Request	Binary Request

{
"format": “text”,
"subformat": “english”,
"content": "Tell me a fun fact",
}
	{
	"format": “text”,
	"subformat": “english”,
	"content": "Describe this picture",
	"submessages": [
	{
	"format": "binary",
	"subformat": "jpeg",
	"content": <base-64-encoding>
	}
	]
	}

4. **Communication with NLIP Backend**
   - The serialized message is sent to the NLIP backend as a secure HTTPS POST request to endpoint `https://druid.eecs.umich.edu/nlip.`
   - A rootCA certificate is used for SSL certification.

5. **Response Handling**
   - The backend processes the message and returns a response.
   - The application parses the response and displays it on the frontend.

Code Structure

Main.tsx

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App.tsx";
import { RecoilRoot } from "recoil";
import "./index.css";

ReactDOM.createRoot(document.getElementById("root")!).render(
 <React.StrictMode>
     <RecoilRoot>
       <App />
     </RecoilRoot>
 </React.StrictMode>
);
```

- **Root File**: It initializes and renders the React app into the DOM.
- **Framework Setup**: It wraps the main `App` component in higher-order components.
- **RecoilRoot:** Sets up `Recoil`, a state management library for React.
- **Global Styles**: It imports global styles via `index.css`.

App.tsx

```
import { Playground } from "./components/playground";

function App() {
 return (
   <>
     <div>
       <Playground />
     </div>
   </>
 );
}

export default App;
```

Serves as the main component of the application, rendering the child component of `Playground`, where most of the frontend functionality resides.

NLIP `Message` Structure

```
export interface Message {
   control?: boolean;
   format: Format;
   subformat: SubFormat | string;
   content: string;
   submessages?: Message[];
 }
```

- `control`: Optional field that determines if the message is for control or data purposes.
- `format`: Specifies the message format (`text` or `binary`).
- `subformat`: Specifies additional format details (e.g., `english`, `jpeg`).
- `content`: The message content (text or base64-encoded binary data) that is being sent between the client and server.
- `submessages`: an optional field whose value containing one+ NLIP messages

---

`PostMessage`

```
import { Message } from "@/components/message";

export async function postMessage(msg: Message): Promise<any> {

 try {

   const response = await fetch("https://druid.eecs.umich.edu/nlip", {

     method: "POST",

     headers: {

       "Content-Type": "application/json",

     },

     body: JSON.stringify(msg),

   });

   if (!response.ok) {

     throw new Error(`HTTP error! Status: ${response.status}`);

   }

   const responseData = await response.json();

   return responseData;

 } catch (error) {
```

```
    console.error("Error:", error);
  }
}
```

- **Asynchronous:** Uses `async/await` to handle asynchronous code for cleaner syntax.
- **Structure**:
  - Parameters: `msg`, an NLIP `Message` structure
  - Returns a `Promise` that resolves to `any`
- **Serializer**: Converts NLIP `Message` to JSON structure through `JSON.stringify()`
- **Fetch API**: Sends POST request to `https://druid.eecs.umich.edu/nlip`, with JSON data as request payload

---

`Playground` - Main Handler

Workflow

1. Users can input text in the text box and select an image from their machines.
2. If there is text or image (or both), the application stores as a `user` message and renders on the frontend.
3. Constructs a NLIP `Message` object, which is a text format message, with a binary format `submessage` if an image is uploaded.
4. The message is sent to the backend through the `postMessage` function.
5. Once the backend responds, parses the reply as a `chatbot` message and displays it to the frontend.