

Project Documentation

By Vincent Carrancho, Robin Obregon, Stephanie Torres, Nicholas Belschner, Garvee Valcourt.

Table of Contents

Table of Contents.....	1
[Frontend] Introduction.....	2
Technology Stack.....	2
[Frontend] Running Application.....	3
Local Development.....	3
Deployment.....	3
Quick Notes.....	4
Note about Data-Fetching.....	4
Note about CSS.....	4
Routing.....	5
Protected Routes.....	5
Route Details.....	5
[Frontend] Account Creation/Login Pages.....	6
Overview.....	6
Form Submission.....	6
Error Handling.....	7
Notes on Forgot Password Page.....	7
[Frontend] Dashboard Page (Auth Route).....	8
Overview.....	8
Driving Functions.....	8
Fetching Conversations on Page Load.....	8
[Frontend] Chat Messaging Interface.....	9
Overview.....	9
Key Components.....	9
State Variables.....	9
Hooks.....	10
Driving Functions.....	10
Message Sending Function Documentation.....	10
Message Block.....	12
[Frontend] Notes Page (Sub-Page of Chat Interface).....	12
Overview.....	12
Key Components.....	12
State Variables.....	13
Hooks.....	13

[Frontend] Utility Functions.....	13
Theme Provider.....	13
Overview.....	13
Key Components.....	13
State Variables.....	14
Hooks.....	14
Functions.....	15
Error Handling.....	15
Storage (Cookies).....	15
Form Schemas.....	15
Overview.....	15
Key Schemas.....	15
Error Handling.....	16
Auth Providers.....	16
Overview.....	16
Key Components.....	16
State Variables.....	17
Hooks.....	17
Functions.....	17
Error Handling.....	17
Usage.....	17
[Frontend] Planned Features for Frontend.....	17
Dashboard.....	18
Chat Conversation Page.....	18
Login/Logout/Forgot Password Pages.....	18
[Backend] Introduction.....	19
Technology Stack.....	19
[Backend] Running Application.....	19
[Backend] Core Singleton.....	19
[Backend] Endpoints.....	20
POST /api/response.....	20
GET /testing/langchain/chain.....	21
[Backend] Planned Features.....	21

[Frontend] Introduction

This is STARS GPT, a tutor chatbot designed to facilitate learning specific to the Knight Foundation School of Computing and Information Sciences (FIU KFSCIS). This documentation

goes over the technical details for the frontend portion of this application. This documentation will be going over the most technically complex portions of the application.

Technology Stack

[React.js](#) Library for building UI Interfaces.

[ShadCN](#) Prebuilt-UI components that are accessible. Make sure to use these components when pages are being made. Avoid using HTML primitives and use ShadCN Components. They have great documentation on how to use this.

[Supabase](#) A comprehensive Backend-as-a-Service (BAAS) that includes features such as Authentication, PostgreSQL, GraphQL, Edge Functions, etc.

[Tailwind](#) Used for making styling much easier and faster.

[BlockNote](#) Notion-Like text editor.

[TanStack Query](#) Asynchronous State Management Solution for Javascript Frameworks.

[Render.com](#) Deployment Service for our test servers.

[Frontend] Running Application

Local Development

This application is a Node.js–React Application. This assumes that you have the .env variables for the frontend and have the backend running.

1. Clone the repository
2. CD into the repository you just downloaded.
3. Drop the .env file into the root of the repository directory.
4. Run `npm install`
5. After installation, run `npm run dev`.
 - a. This will launch the development server locally on localhost:5173.
 - b. Every time you make a change to any file while it is running locally, it'll reload the frontend. Depending on the changes, it will reload in under 2000 ms—on average 200 ms depending on the size of changes (thanks Vite!).

Deployment

This can be done with any website hosting provider. I recommend that since this is an FIU application, it is hosted on Azure's App Service. Here are the deployment commands—this is

going to be different depending on each service provider but this is the general deployment pattern:

1. Follow steps 1-3 from the local development steps.
2. Run ``npm run build``. This will build the applications into a static site. The output of this build step and compilation will be in a new directory called `/dist`.
3. This step will vary depending on what you use as your hosting service. For example on Render.com, this step is not necessary because once the `/dist` file is in the system, then they can deploy it. However, here is the command if otherwise run ``npm run preview``.

Quick Notes

Note about Data-Fetching

Data fetching in this frontend is handled entirely by Tanstack Query and Axios instead using `useEffect` and the `fetch` API. When developing on this repository, **avoid using `useEffect` and `fetch`**.

Note about CSS

This application is built on `tailwind`. Instead of using CSS files use `tailwind`—only there are 2 usable CSS files you'll find in this project. The first one is the `globals.css` in the `src/` directory. If you want to change the color scheme, borders, roundness, etc, you would change it in that specific file (use this tool to copy and paste new themes into that CSS file:

<https://zippystarter.com/tools/shadcn-ui-theme-generator>). The reason is because the `ThemeProvider` relies on the CSS defined in this file. Here are some things to know.

The ``globals.css`` file is a global stylesheet that applies styles to the entire application. It uses the `Tailwind CSS` library for utility classes and custom properties for theming.

1. **Tailwind CSS Directives:** The `@tailwind` base, `@tailwind` components, and `@tailwind` utilities directives are used to include `Tailwind's` base, component, and utility styles.
 - a. **Root and Dark Theme Variables:** Two sets of CSS custom properties are defined for light and dark themes respectively. These properties include colors for various UI elements (background, foreground, card, popover, primary, secondary, muted, accent, destructive, border, input, ring) and a radius property for border radius.
2. **Color Variables:** Custom properties (CSS variables) are defined for various colors, such as ``background``, ``foreground``, ``card``, ``primary``, ``secondary``, ``muted``, ``accent``, ``destructive``, ``border``, ``input``, and ``ring``. These properties are defined in the ``:root`` selector and in the ``.dark`` class for light and dark themes, respectively.
3. **Border Radius Variable:** A custom property is defined for the border radius (``radius``).

4. Other Global Styles: Global styles are applied to all elements (`\*`), the `body`, headings (`h1` to `h6`), and list items (`li`). These styles include border color, background color, text color, padding, margin, font size, font weight, and list style.
-

Routing

Routing is done by [React Router](#). The routes are defined in the App component located at src/App.tsx.

Protected Routes

The ProtectedRoutes component is used to protect the routes under "/chat". If the user is not authenticated, they are redirected to the login page. If the user is authenticated, the child routes are rendered. For more information about protected routes, please visit the auth providers section of this.

```
<Routes>
  <Route path="/" element={<Landing />} />
  <Route path="/login" element={<Login />} />
  <Route path="/create" element={<CreateAccount />} />
  <Route path="/forgot" element={<ForgotPassword />}></Route>
  <Route path="/forgot/update" element={<UpdateForgotPassword />} />
  <Route element={<ProtectedRoutes />} path="chat">
    <Route element={<Dashboard />} path="" />
    <Route element={<MessageWindow />} path=":conversationid" />
  </Route>
  <Route path="*" element={<h1>404 Not Found</h1>} />
</Routes>
```

Route Details

- **"/"**: The base route that renders the Landing component located at src/pages/landing.tsx.
- **"/login"**: This route renders the Login component located at src/pages/login/login.tsx.
- **"/create"**: This route renders the CreateAccount component located at src/pages/login/CreateAccount.tsx.
- **"/forgot"**: This route renders the ForgotPassword component located at src/pages/login/forgotPassword.tsx.

- `"/forgot/update"`: This route renders the `UpdateForgotPassword` component located at `src/pages/login/updateForgotPassword.tsx`.
 - `"/chat"`: This route is protected by the `ProtectedRoutes` component located at `src/pages/protectedRoute.tsx`. If the user is authenticated, it renders the `Dashboard` component by default located at `src/pages/dashboard/dashboard.tsx`. If a `conversationid` is provided in the URL (like `"/chat/123"`), it renders the `MessageWindow` component for that conversation located at `src/pages/conversation/messageWindow.tsx`.
 - `"*"`: If the current URL does not match any of the above routes, a 404 page is rendered.
-

[Frontend] Account Creation/Login Pages

Overview

The Login page (`login.tsx`) is where users can enter their email and password to log into the application. It uses the `react-hook-form` library for form handling and validation, and the `zod` library for defining the form schema.

The form is defined using the `useForm` hook from `react-hook-form`. The form has two fields: email and password, both of which are initially empty strings.

The form schema is defined using `zod` and passed to the `resolver` option of `useForm` to enable form validation.

```
const form = useForm<z.infer<typeof login>>({
  resolver: zodResolver(login),
  defaultValues: {
    email: "",
    password: "",
  },
  values: {
    email: username,
    password: password,
  },
});
```

Additionally, the create account page uses the same validation as the login/logout pages.

Form Submission

The `handleSubmit` function is called when the form is submitted. It uses the `signInWithPassword` method from the Supabase client to authenticate the user.

If the authentication is successful, the user is logged in and the function does nothing. If there's an error, the error message is logged to the console and set as a form error.

```
const handleSubmit = async (values: z.infer<typeof login>) => {
  await supabase?.auth
    .signInWithPassword({
      email: values.email,
      password: values.password,
    })
    .then((res) => {
      if (res.error) {
        console.log(res.error);
        form.setError("root", {
          message: res.error.message,
        });
        throw new Error(res.error.message);
      } else if (res.data) {
        navigate("/chat");
      }
    })
    .catch((e) => {
      toast({
        title: "❌ Error",
        description: e.message,
      });
    });
};
```

Error Handling

If there's an error during form submission, the error message is set as a form error using the `setError` method from `react-hook-form`. The error message is associated with the "root" field, which can be used to display a general error message above or below the form. Also, a error toast will pop up on the screen

Notes on Forgot Password Page

As of Monday April 8th 2024, the Forgot Password page is front-end-functional. There are plans to connect an SMTP so we can send emails.

[Frontend] Dashboard Page (Auth Route)

Overview

The Dashboard page (dashboard.tsx) is the main interface for the application where users can interact with their conversations. It uses the react library for building the user interface and the react-router-dom library for navigation.

Once a user is logged in, the user will be able to do many things. If they click on an arrow leading in one of the dashboard card components, they will be redirected to that specific conversation.

If a user wants to create a conversation, then they can. It follows the same form validation system that the login/create account pages use.

Driving Functions

Fetching Conversations on Page Load

```
const { data, error, isFetching } = useQuery({
  queryKey: ["conversations"],
  queryFn: async () => {
    await new Promise((r) => setTimeout(r, 1000));
    const res = await supabase
      ?.from("conversations") // THROW ERROR HERE
      .select("*")
      .eq("owner_id", auth.user?.id ?? "");
    if (res?.error) {
      throw res.error;
    }
    return res?.data as Tables<"conversations">[];
  },
  retry: 3,
});
```

Filtering conversations based on the search bar:

```
useEffect(() => {
  const filterConversations = () => {
    // Implement this
    const filteredConvo = conversations?.filter(
```

```
        (conversation) =>
            conversation.title?.toLowerCase().includes(search.toLowerCase())
    ||

conversation.description?.toLowerCase().includes(search.toLowerCase())
    );
    setFilteredConversations(filteredConvo);
};
filterConversations();
}, [search]);
```

[Frontend] Chat Messaging Interface

Overview

The Message Window page (messageWindow.tsx) is where users can view and interact with a specific conversation. It includes a chat interface, chat settings, and a notes section.

Key Components

- Message Display: Messages for the current conversation are fetched from a server and stored in the messages state variable. These messages are then rendered in a list, with each message represented by a MessageComponent.
- Message Input: The Textarea component is used to create a text area where users can type their messages. The userInput state variable stores the current user input.
- Send Button: The Button component is used to create a send button. When this button is clicked, the sendMessage mutation function is called.
- Chat Settings Dialog (Disabled): The Dialog component is used to display the chat settings. The openSettings state variable determines whether the dialog is open or not. The ChatSettings component is passed the conversationid prop.
- Notes Sheet: The Sheet component is used to display the notes for the current conversation. The openSheet state variable determines whether the sheet is open or not. This opens the notes sheet so the user can take notes.

State Variables

- messages: An array that stores the messages for the current conversation.
- userInput: A string that stores the current user input.
- openSettings: A boolean that determines whether the chat settings dialog is open or not.
- openSheet: A boolean that determines whether the notes sheet is open or not.

Hooks

- `useEffect`: This hook is used to automatically scroll to the latest message whenever the messages array changes, to enable or disable the send button based on whether the user input is empty, and to set the initial messages when the component mounts or when the conversationid changes.
- `useMutation`: This hook is used to create the `sendMessage` mutation function that sends a message to the server and updates the local state to include the new message.
- `useQuery`: This hook is used to create the `getInitialMessage` and `getConversationData` query functions that fetch the initial messages and conversation data from the server, respectively.
- `useToast`: This hook is used to display toast notifications.
- `useForm`: This hook is used to manage the form state for the message input.
- `useParams`: This hook is used to get the conversationid from the URL parameters.
- `useNavigate`: This hook is used to navigate to different pages.

Driving Functions

Message Sending Function Documentation

The `sendMessage` function is a mutation function created using the `useMutation` hook from `react-query`. It's responsible for sending a user's message to the server, updating the local state to include the new message, and handling the server's response.

It follows this pattern: **User Input Message Creation > State Update > 1st Database Insertion > OpenAI Response > 2nd Database Insertion > State Update**

Here is the code all together:

```
const sendMessage = useMutation({
  mutationKey: ["sendMessage"],
  mutationFn: async () => {
    const newMessage: Tables<"Messages"> = {
      content: userInput,
      conversation_id: conversationid ?? "",
      role: "user",
      id: uuidv4(),
      created_at: new Date(
        Date.now() + 1000 * 60 * -new Date().getTimezoneOffset()
      )
        .toISOString()
        .replace("T", " ")
        .replace("Z", ""),
    };
  }
});
```

```

messages.push(newMessage);
setMessages(messages);

const newMessagesInsertion = await supabase
  ?.from("Messages")
  .insert([newMessage]);

if (newMessagesInsertion.error) {
  throw new Error("Failed to insert into database");
}

const result = await axios({
  method: "post",
  url: `${import.meta.env.VITE_BACKEND_LINK}/api/response`,
  data: messages,
  params: {
    conversation_id: conversationid,
    model: getConversationData.data?.[0].model,
  },
}).then(async (res) => {
  if (res.status !== 200) {
    throw new Error("Bad Request, Try Again later");
  }
  const openaiMetadata: Tables<"OpenAI-Responses"> =
res.data.metadata;
  const openAIResponseMessage: Tables<"Messages"> = res.data.message;

  const openAIMessagesInsertion = await supabase
    ?.from("Messages")
    .insert([openAIResponseMessage]);

  const metadataInsertion = await supabase
    ?.from("OpenAI-Responses")
    .insert([openaiMetadata]);

  if (openAIMessagesInsertion.error || metadataInsertion.error) {
    throw new Error("Failed to insert into database");
  }

  return {
    metadata: openaiMetadata,
    message: openAIResponseMessage,
  };
});

```

```
});  
  
    return result;  
  },  
  onSuccess(data) {  
    setUserInput("");  
    data;  
    setMessages((prev) => [...prev, data.message]);  
  },  
});
```

Message Block

The MessageComponent (messageComponent.tsx) is a component that displays a single message in a conversation. It supports both text and code messages, and it uses the react-markdown library to render markdown. It supports LaTeX/Katex rendering so that the bot can display math equations if required by the user.

One of the limitations of this message block is that it does not support the creation of tables. But more importantly, the biggest limitation to this is performance. Everytime a state gets updated inside of this conversation messaging interface page, every block gets rerendered causing wasteful recalculations (this is one of the tasks that the next development team will tackle).

[Frontend] Notes Page (Sub-Page of Chat Interface)

Overview

The Notes page (Notes.tsx) is where users can view and edit notes for a specific conversation. It uses the @blocknote/react library for the notes editor and the react-query library for data fetching and mutation.

Key Components

Notes Editor: The BlockNoteView component is used to display the notes editor. The editor prop is passed the BlockNoteEditor instance created by the useBlockNote hook.

- Save Notes Mutation: The useMutation hook is used to create a mutation function (saveNotes) that saves the current notes to the server. The onSuccess callback is called after a successful mutation and displays a toast notification.

- Fetch Initial Notes Query: The `useQuery` hook is used to create a query function (`fetchInitialNotes`) that fetches the initial notes from the server when the editor is ready.
- Autosave: The `useAutosave` hook is used to automatically save the notes every 10 seconds.
- Keyboard Shortcuts: A `useEffect` hook is used to listen for when the user presses `CTRL+S` or `⌘+S` and save the notes.

State Variables

`data`: An array of blocks that represents the current notes.

`visualTheme`: A string that represents the current theme ("dark" or "light").

Hooks

- `useEffect`: This hook is used to listen for keyboard shortcuts and to set the visual theme based on the current theme.
 - `useMutation`: This hook is used to create the `saveNotes` mutation function.
 - `useQuery`: This hook is used to create the `fetchInitialNotes` query function.
 - `useBlockNote`: This hook is used to create the `BlockNoteEditor` instance.
 - `useAutosave`: This hook is used to automatically save the notes every 10 seconds.
 - `useToast`: This hook is used to display toast notifications.
-

[Frontend] Utility Functions

Theme Provider

Overview

The `ThemeProvider` component (`themeprovider.tsx`) is a context provider that allows child components to access and modify the current theme. It supports "dark", "light", and "system" themes. The reason we need this is to give users the option to switch between light mode and dark mode for accessibility purposes. This is switched via a UI component.

Key Components

- `ThemeProviderContext`: The `ThemeProviderContext` is a context created with `createContext`. It provides a theme state variable and a `setTheme` function to all child components.

```
const ThemeProviderContext =
```

```
createContext<ThemeProviderState>(initialState);
```

- ThemeProvider Component: The ThemeProvider component is a context provider that provides the theme state variable and setTheme function to all child components. It also saves the current theme to localStorage and updates the theme class on the root HTML element.

```
export function ThemeProvider({
  children,
  defaultTheme = "system",
  storageKey = "vite-ui-theme",
  ...props
}: ThemeProviderProps) {
  // ... code to manage theme state ...

  return (
    <ThemeProviderContext.Provider {...props} value={value}>
      {children}
    </ThemeProviderContext.Provider>
  );
}
```

- useTheme Hook: The useTheme hook is a custom hook that allows any component to access the theme state variable and setTheme function from the ThemeProviderContext.

```
export const useTheme = () => {
  const context = useContext(ThemeProviderContext);

  if (context === undefined)
    throw new Error("useTheme must be used within a ThemeProvider");

  return context;
};
```

State Variables

- theme: A string that represents the current theme ("dark", "light", or "system").

Hooks

- useState: This hook is used to manage the theme state variable.
- useEffect: This hook is used to update the theme class on the root HTML element whenever the theme state variable changes.
- useContext: This hook is used in the useTheme hook to access the ThemeProviderContext.

Functions

- `setTheme`: This function is used to update the theme state variable and save the current theme to `localStorage`.

Error Handling

- If the `useTheme` hook is used outside of a `ThemeProvider`, an error is thrown.

Storage (Cookies)

- The current theme is saved to `localStorage` using the `storageKey` prop. This allows the theme to persist across page reloads.

Form Schemas

Overview

The `zodtypes.ts` file defines various Zod schemas for validating and parsing data in the application. Zod is a TypeScript-first schema declaration and validation library.

Key Schemas

- `newConversation`: This schema is used for creating new conversations. It expects an object with a `title` (a string of 10 to 100 characters), a `description` (a string of 10 to 500 characters), and a `model` (a string that must not be empty).
- `newMessage`: This schema is used for creating new messages. It expects an object with a `message` (a string of at least 1 character).
- `login`: This schema is used for user login. It expects an object with an `email` (a valid email address) and a `password` (a string of at least 6 characters).
- `register`: This schema is used for user registration. It expects an object with an `email` (a valid email address that ends with ".fiu.edu"), a `confirmEmail` (a string that must match the email), a `password` (a string of at least 6 characters), a `confirmPassword` (a string that must match the password), and a `confirmed` (a boolean that must be true).
- `conversationMessageInput`: This schema is used for validating conversation message inputs. It expects an object with a `message` (a string of at least 1 character).
- `conversationSettingsChange`: This schema is used for changing conversation settings. It expects an object with a `title` (a string of 10 to 100 characters), a `description` (a string of 10 to 500 characters), a `model` (a string that must not be empty), and a `tone` (an array of numbers with a length of 1).
- `forgotPasswordInitializer`: This schema is used for initializing the password reset process. It expects an object with an `email` (a valid email address).
- `forgotPasswordUpdate`: This schema is used for updating the password during the password reset process. It expects an object with a `passwordReset` (a string of at least 6 characters) and a `confirmPasswordReset` (a string that must match the `passwordReset`).

Error Handling

Each schema uses the `refine` method to add custom validation rules. If a rule is not met, a custom error message is returned. For example, the `register` schema checks if the email ends with "fiu.edu", if the password matches the `confirmPassword`, if the email matches the `confirmEmail`, and if `confirmed` is true. If any of these checks fail, a custom error message is returned.

Auth Providers

Overview

The `AuthProvider` component (`authprovider.tsx`) is a context provider that allows child components to access and modify the current user's session and user data. It uses the `@supabase/supabase-js` library for authentication.

Key Components

- `AuthContext`: The `AuthContext` is a context created with `createContext`. It provides a session state variable and a user state variable to all child components.

```
export const AuthContext = createContext<{
  session: Session | null | undefined;
  user: User | null | undefined;
}>({ session: null, user: null });
```

- `AuthProvider` Component: The `AuthProvider` component is a context provider that provides the session and user state variables to all child components. It also fetches the current session and user data when the component mounts and updates the state variables whenever the auth state changes.

```
export default function AuthProvider({ children }: any) {
  // ... code to manage session and user state ...

  return (
    <AuthContext.Provider value={value}>
      {!loading && children}
    </AuthContext.Provider>
  );
}
```

- `useAuth` Hook: The `useAuth` hook is a custom hook that allows any component to access the session and user state variables from the `AuthContext`.

```
export const useAuth = () => {
```

```
    return useContext(AuthContext);  
  };
```

State Variables

- session: An object that represents the current user's session.
- user: An object that represents the current user's data.
- loading: A boolean that represents whether the session and user data are still loading.

Hooks

- useState: These hooks are used to manage the session, user, and loading state variables.
- useEffect: This hook is used to fetch the current session and user data when the component mounts and to update the state variables whenever the auth state changes.
- useContext: This hook is used in the useAuth hook to access the AuthContext.

Functions

- setData: This function is called when the component mounts. It fetches the current session and user data from Supabase and updates the state variables.
- onAuthStateChange: This function is called whenever the auth state changes. It updates the session and user state variables and sets loading to false.

Error Handling

If fetching the current session fails, an error is thrown.

Usage

The AuthProvider component should be used to wrap any components that need to access or modify the current user's session or user data. The useAuth hook can be used within these child components to access the session and user state variables.

[Frontend] Planned Features for Frontend

Most of these features are planned to be implemented in future development cycles.

- ☐ Migration to Next.js 14

Dashboard

- ☐ Sorting and Filtering notes by various different features.

- ☐ Implement settings tab so that users can set account preferences and/or change email and password.
- ☐ Enable deletion of conversation (this would not remove the relationship between the user though inside of the database).
- ☐ Use of tags for organization.
- ☐ Implement Collections of conversations for further organization.

Chat Conversation Page

- ☐ Chat setting page that allows users to change various things about the model.
 - ☐ Change the tone of the conversation (ie, simpler to professional)
 - ☐ Change the model that the conversation is using.
 - ☐ Change name and description of conversation. These values are read by the AI.
- ☐ Introduce a setting in which if the last message is from a user, they can resend it (used for handling edge cases in which the user exits from the conversation before bot responds).
- ☐ **Performance improvements to the conversation page. Most likely using Memoization to prevent unnecessary re-renders (performance is acceptable in current version as of April 3rd, 2024).** However, in longer conversations, performance is expected to have an impact.
- ☐ Change layout of the entire page so that you can look at notes at the same time as interacting with a bot. <https://ui.shadcn.com/docs/components/resizable> this would be quite helpful in doing this. Left side would be a chatbot while the right side would be the notes. The notes portion can be collapsible.
- ☐ Implement a WYSIWYG message input so users can format messages in markdown—think Notion or Discord with realtime markdown.
- ☐ Better Error handling.
- ☐ Notes component supporting LaTeX equations and rendering.

Login/Logout/Forgot Password Pages

- ☐ Password-less Sign-in (i.e emailing a special link that authenticates the user on click.)
- ☐ OTP Login Page variant
- ☐ Phone Number Auth.
- ☐ Connect SMTP service so Forgot Password Pages work.

[Backend] Introduction

This is documentation for the backend. This backend only has 2 available endpoints right now: 1 for testing and one for production use. The purpose of this backend is to act as an intermediary between OpenAI and the client while also injecting Langchain in the API. On your browser, you can go to this link: <https://<API-IP-ADDRESS>/docs> or wherever you are hosting the API>/docs. This link will take you to the automatically generated API docs based on the endpoint.

Technology Stack

[FastAPI](#) Fast and Easy to learn framework for web applications.

[LangChain](#) Helps us work with OpenAI's LLMs.

[Backend] Running Application

This assumes that you have Python 3.10+ installed (this has been tested to work with Python 3.12 too) and you have a virtual environment using Venv or Conda. Additionally, this assumes that you have the .env file to make this work. This will work for deployment too.

1. Clone the repository
 2. CD into the repository and make sure the python environment you are on is the environment you will use to run this application.
 3. Run `pip3 install -r requirements.txt`. This will install the dependencies
 4. Drop the .env file into the root directory of the repository.
 5. Run `python3 src/main.py`. This will launch the FastAPI Application.
-

[Backend] Core Singleton

We are using a singleton pattern in order to communicate with the LLM. It is located within `src/langchainAPI/ModelSingleton.py`. Using a singleton will allow us to avoid instantiating the chain over and over again. The reason is because instantiation introduces some latency and might cost us some tokens.

1. Class Variables: The class has three class variables: `__instance`, `chain`, and `model`. `__instance` is used to store the singleton instance of the class. `chain` is used to store

the chatbot model chain. `model` stores the name of the model being used, which is "gpt-4-turbo" by default.

2. `initialize_chain` Method: This static method initializes the chatbot model chain. It creates a vector store from a set of examples, sets up a semantic similarity example selector [1], defines a few-shot prompt template, assembles the final prompt template, and creates the chat model using OpenAI's GPT-4. It then creates a chain with the final prompt and the chat model, and returns this chain.
3. `change_model` Method: This static method allows you to change the model being used. It updates the `model` class variable and re-initializes the `chain` with the new model.
4. `get_model` Method: This static method returns the current model being used. This is mostly a test method

[1] Semantic Similarity Selector is used to determine how similar one piece of text is to another. In this application, text is converted into vectors and they are compared to each other to see if they "point" in roughly the same direction. Since it is converted into vectors, that is why we need a vector store.

[Backend] Endpoints

POST /api/response

Takes 2 parameters: `conversation_id` and `model`. Both are strings. Additionally, the call must provide a body that is an array of messages (which is the chat history).

```
[
  {
    "content": "string",
    "conversation_id": "string",
    "created_at": "string",
    "id": "string",
    "role": "string"
  }
  ...
]
```

This returns a response that is of this structure:

```
{
  "message": {
```

```

    "role": "assistant",
    "content": "### Understanding String Assignment in Python\n\nIt looks like you're working with string assignment in Python. Here's a brief overview to ensure you're on the right track:\n\n- Strings in Python are defined either with single quotes (`' '` ) or double quotes (`\" \"`).\n- When you assign a string to a variable, you ensure that any text you want to store is enclosed within these quotes.\n\nFor your example:\n\n```\npython\ncontent = 'string'\n```\n\n- `content` is the variable name.\n- `'string'` is the string assigned to the variable.\n\nIf you have any specific questions about this or need further clarification on how strings work in Python, feel free to ask!",
    "conversation_id": "123",
    "created_at": "2024-04-16T19:33:45",
    "id": "875f98e0-b54e-4514-9374-67af6b7d93bb"
  },
  "metadata": {
    "chat_completion_id": "c981662a-6c8c-4c6f-ae8b-5365afb4646f",
    "completion_tokens": 139,
    "created": "2024-04-16T19:33:45",
    "id": "c981662a-6c8c-4c6f-ae8b-5365afb4646f",
    "message": "875f98e0-b54e-4514-9374-67af6b7d93bb",
    "model": "gpt-4-turbo",
    "prompt_tokens": 513,
    "system_fingerprint": null,
    "total_tokens": 652
  }
}

```

GET /testing/langchain/chain

No parameters. This is a test endpoint to make sure the API server is up and running. This works more less the same as the endpoint above. However, the chat history and model the LLM is using is hard-coded into the function body.

[Backend] Planned Features

- ☐ Insertion message into Supabase after the response instead of the client.
- ☐ Implementation of authentication (this was already written but there was no time to test if it works). Essentially reject any request that is being made that does not include any auth credentials.

End of Documentation