Setup instructions

After creating the Node.js project using the command:

npm init -y

I followed the steps from the **Trigger.dev** documentation:

1. Initialized Trigger.dev

I ran the following command:

npx trigger.dev@latest init

This command:

- Logged me into the CLI
- Created the trigger.config.ts file in the root of my project.
- Created the trigger directory and added the task chatAI.ts

Installed Dependencies: To support the project, I installed these additional dependencies:

Axios for handling API requests:

- npm install axios

LowDB for managing a lightweight JSON-based database:

- npm install lowdb

2. Started the Development Server

Next, I ran:

npx trigger.dev@latest dev

This command:

- Started the server to run and watch tasks.
- Synced the project with the **Trigger.dev** platform to register tasks, execute runs, and exchange data.
- Displayed useful data in the terminal (for testing and viewing task results).

Server is now running

The server is active, monitoring changes in the /trigger directory.

Documentation of architecture decisions

The primary objective is to create a task-based workflow that:

- Interacts with Hugging Face's GPT-2 model to generate text responses.
- Processes the responses for further use in API integrations.
- Sends the processed responses back to a frontend (Retool) to build a chatbot interface.

Design decisions

The task workflow is modular, handling:

- Input validation: Ensures valid data is processed.
- Response processing: Prepares responses for downstream APIs or frontend integration.
- Frontend integration: Sends responses back to Retool for chatbot functionality.

Task Workflow

The flow is as follows:

- **Frontend Request to Trigger.dev**:

  Retool sends a POST request to the backend (Trigger.dev task) containing:

  - The user’s question.
  - An identifier (id) for tracking the question and its corresponding response.
- **Processing and Output Generation**:

  The backend:

  - Processes the question.
  - Sends it to the Hugging Face API for text generation.
  - Retrieves the output and formats it for the frontend.
- **Response Retrieval by Retool**:

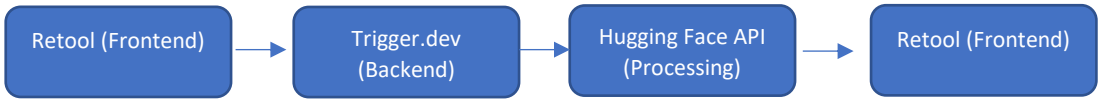
  - After processing, the response is stored in an output object tied to the provided id.
  - Retool performs a GET request to the backend (Trigger.dev) using the same id to retrieve the processed information.
  - The output is then displayed in the chatbot interface.

Tool Selection Rationale

- Trigger.dev: Modular task-based workflows and real-time updates.
- Hugging Face: State-of-the-art text generation with GPT-2.
- Retool: Quick development of a functional frontend interface.

Architecture overview

I've designed the architecture diagram to be as simple and streamlined as possible, emphasizing clarity over complexity.



- Retool (Frontend):

The user interacts with the chatbot interface on Retool, submitting their question. This triggers a POST request to the backend via Trigger.dev.

- Trigger.dev (Backend):

The backend processes the incoming request, validates the input, and forwards it to the Hugging Face API for text generation.

- Hugging Face API (Processing):

Hugging Face generates a response using the GPT-2 model, which is then sent back to the backend for further processing.

- Retool (Frontend):

After processing, the final response is sent back to Retool, where it is displayed in the chatbot interface for the user.

Challenges Faced and Solutions

Learning New Technologies

- **Challenge**: Both **Trigger.dev** and **Retool** were new tools for me. This necessitated a considerable amount of time spent on studying their documentation and exploring their features. As these tools were unfamiliar, I had to understand their workflows and best practices before integrating them.
- **Solution**: I approached this challenge by working through smaller, isolated examples to understand the behavior of each tool. By experimenting with basic features first, I

gradually built up my knowledge, allowing me to integrate them successfully into the main project.

Retool's Chat Component and Redundant Data

- **Challenge**: The **Retool chat component** required configuring the query from which the bot would retrieve responses. However, once I set up the query, it was being executed multiple times instead of once, which led to redundant data being returned. This redundancy cluttered the chat interface and made it difficult to manage responses effectively. I could not figure out how to gain more control over the execution flow of the chat component within the given time frame.
- **Solution**: Initially, I tried to work within the constraints of the chat component by adjusting how the query was triggered and making sure the backend responses were properly formatted. However, because the component executed the query multiple times, the solution was only a partial fix. Unfortunately, I was unable to identify a specific option or setting within Retool that would allow me to have finer control over the query execution to prevent multiple triggers.