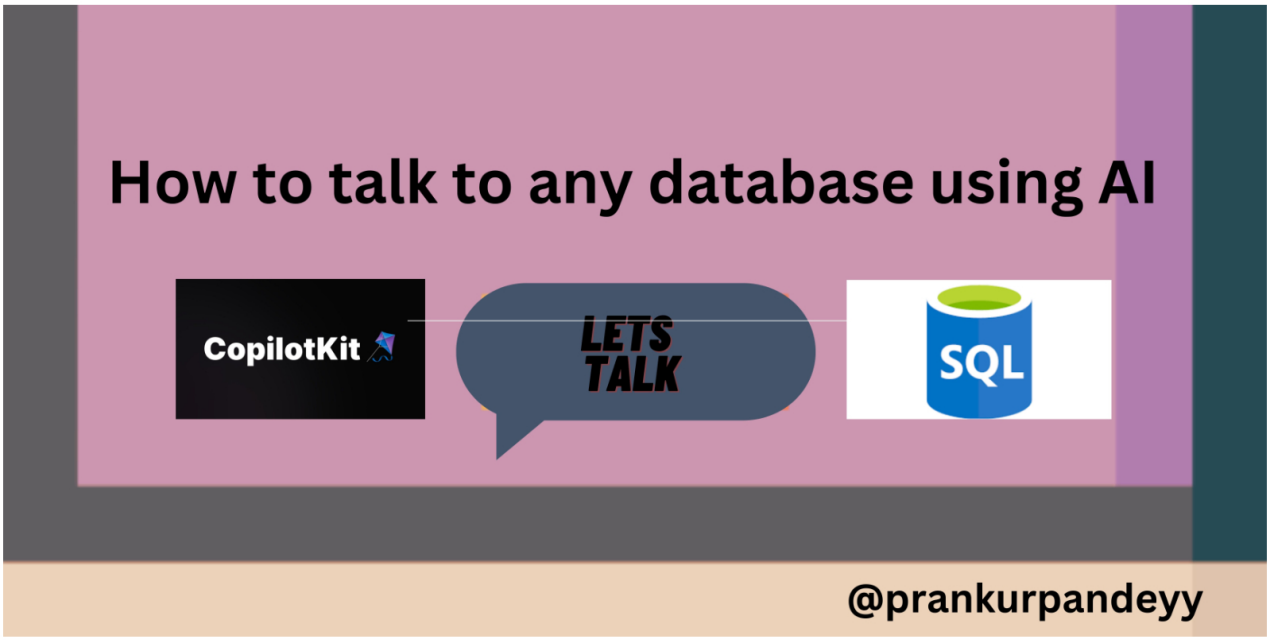
JANUARY 8, 2025 / #AI

How to Talk to Any Database Using AI – Build Your Own SQL Query Data Extractor


Prankur Pandey



Recently, I took a break from writing to focus on my exams. During this time, I had an interesting experience: I had the chance to explain SQL (Structured Query Language) to my peers. While exploring SQL in-depth, I encountered a common frustration: writing SQL queries to fetch specific data from a database.

This sparked an idea. What if I could build a tool where I didn't have to write SQL queries manually? Instead, I could type in plain, natural English and let the database do the heavy lifting for me.

Given that we live in the era of AI, leveraging artificial intelligence was the only way to turn this vision into reality.

In this tutorial, I'll walk you through creating an AI-powered SQL query data extractor. This tool will enable you to fetch data from a database effortlessly, without writing a single line of SQL code.

What we'll cover:

What we'll cover:

Prerequisites & Tools

In this tutorial, we'll build an AI-powered SQL query data extractor tool. It'll allow us to interact with a database using natural language, like plain English, and receive the same results as if we had written SQL queries.

Here's an overview of the tools we'll use to create this cool app:

Database

The database is a critical component where we'll store data and later extract it for our AI model to use when performing NLP operations. Instead of hosting a database locally, I chose a cloud-based free database that allows data extraction via REST APIs. For this project, I opted for restdb.io because it offers seamless SQL database provisioning and supports REST APIs.

AI Agent

An AI Agent will act as the intermediary between the database and the AI model. This agent will manage the AI model's operations and facilitate seamless communication. For this, I am using **CopilotKit**, which simplifies the integration process.

AI (LLM) Model

The AI model translates plain English queries into SQL queries. For this, I am using **GroqAI**, which supports various popular AI models and provides the flexibility needed for this project.

Next.js

To develop a web application that supports both frontend and backend functionalities, I chose **Next.js**. It's an ideal framework for building robust, scalable web apps with server-side rendering capabilities.

Deployment

For deployment, you can choose any service. I prefer **Vercel**, as it integrates seamlessly with Next.js and is free for hobby projects.

By combining these tools, we'll build a powerful, user-friendly application that effortlessly bridges natural language and SQL databases.

What We'll Do Here:

These are the steps we'll follow in this tutorial to build our app:

Step 1 – Set Up the Database: Either set up the database locally, deploy it, and access it, or use an online database tool that allows data access and extraction via REST APIs.

Step 2 – Obtain Cloud API Keys: Get the necessary API keys for your AI model to enable seamless integration.

Step 3 – Build a Web App: Create a web application and set up the backend to integrate CopilotKit. Configure it within the app for optimal functionality.

Step 4 – Train CopilotKit on Your Database: Provide your database's data to CopilotKit. It will read and understand the data to facilitate natural language processing.

Step 5 – Integrate CopilotKit Chat: Add the CopilotKit chat interface into your application and configure it to ensure smooth operation.

Step 6 – Test Locally: Test the app on your local machine to identify and fix any issues.

Step 7 – Deploy the App: Once everything is working as expected, deploy the application to a hosting platform.

How Does the App Work?

Have you ever wondered how writing plain English could allow you to fetch data from a SQL database?

The magic lies in CopilotKit. It lets you create AI-powered copilots that can perform operations on your applications. Think of CopilotKit as your personal AI assistant or chatbot. So how does it work?

Well, first we have CopilotKit which serves as our chatbot powered by advanced AI models.

Then when you provide data to the chatbot, it uses that data to train itself,

building an understanding of your database structure and content.

Finally, when a natural language query (like "Who is using this email address?") is inputted, the AI model processes it, translates it into a corresponding SQL query, and retrieves the desired data from the database.

With CopilotKit's powerful AI capabilities, your application can seamlessly bridge natural language and SQL, making database interactions more intuitive.

How to Set Up Your Tools

Now we'll go through everything you need to set up the project.

1. Install Next.js and dependencies:

First, you'll need to create a NextJS app. Go to the terminal and run the following command:

```
npx create-next-app@latest my-next-app
```

Replace `my-next-app` with your desired project name.

Navigate to the project folder:

```
cd my-next-app
```

Start the development server:

```
npm run dev
```

Open your browser and navigate to `http://localhost:3000` to see your Next.js app in action.

2. Install CopilotKit and dependencies

Go to the project root folder through the terminal and run the below command. It will install all the important CopilotKit dependencies and other important packages like dotenv and Axios.

```
npm install @copilotkit/react-ui @copilotkit/react-core dotenv axios
```

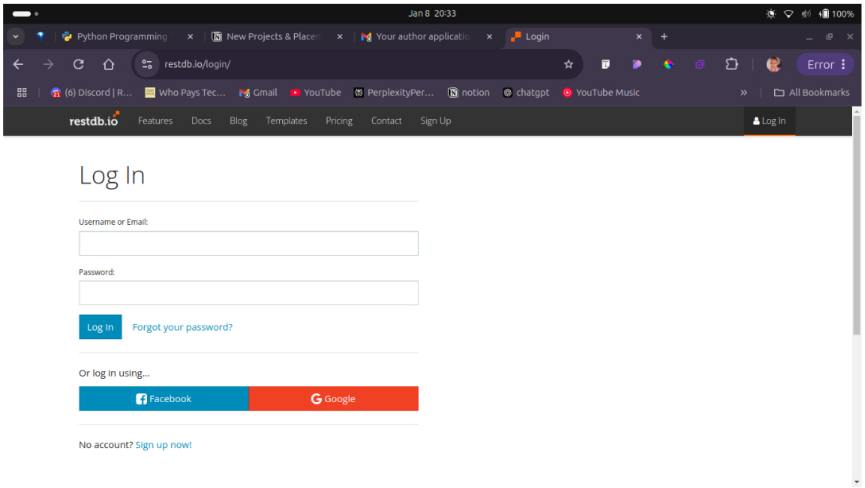
- The **CopitlotKit** dependency is solely for handling CopilotKit

operations and configurations.

- The **Dotenv** dependency is used to handle environment variables as we have to keep important keys in the project, such as environment variables.
- **Axios** is for handling the API calls.

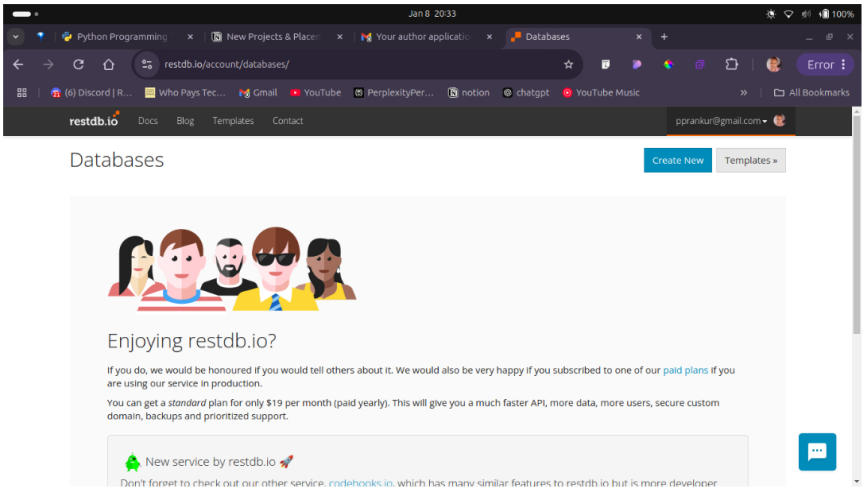
3. Set Up the Database

Visit RestDB.io and either login or create an account.

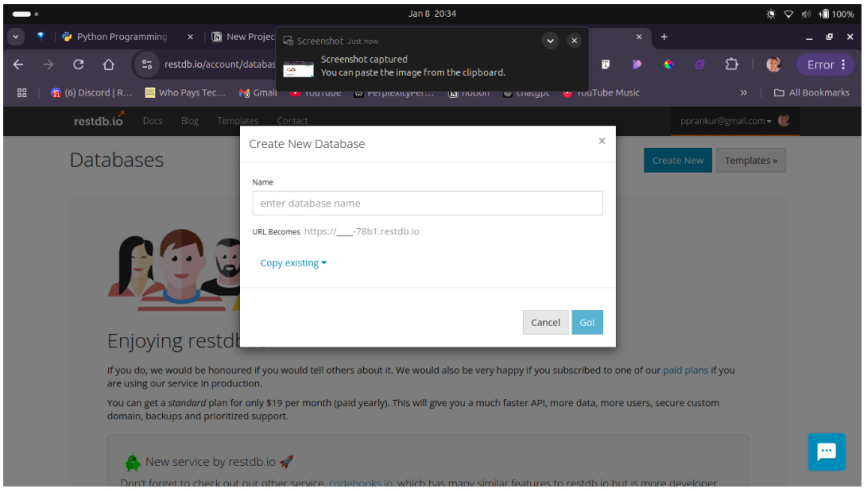


Above you can see the login page for RestDB.io you can either log in if you already have an account or create a new account .

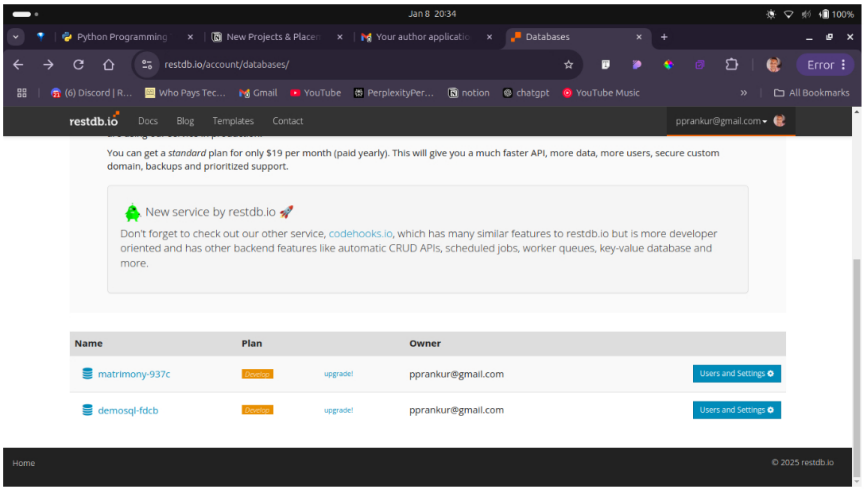
Once logged in you will redirected to this page. There you'll see the button to create a new database.



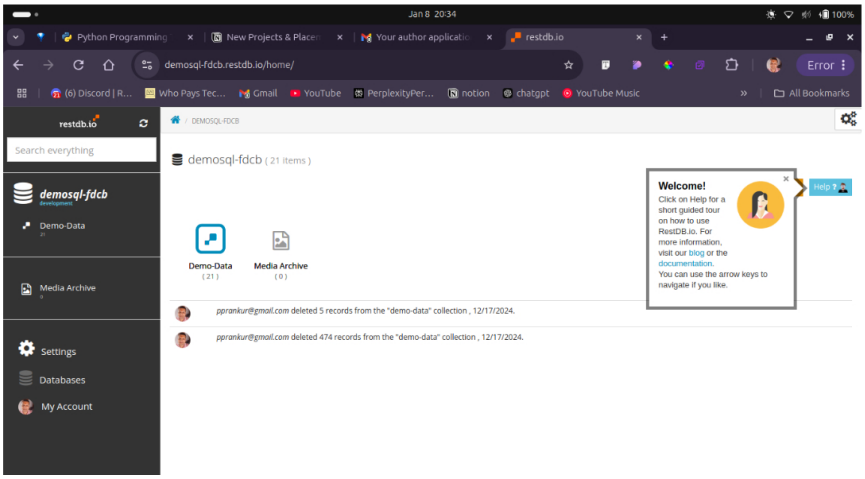
When you click on the Create New button, a pop will appear. There, you'll have to enter the database name as shown in the image below:



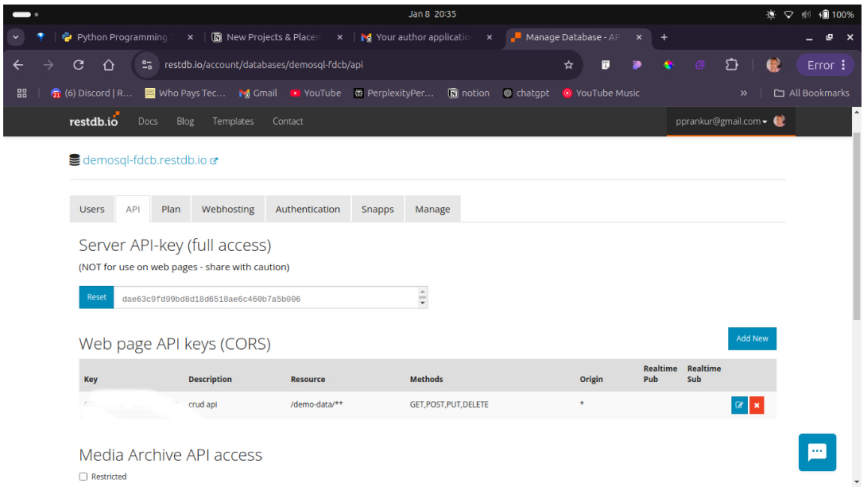
When you enter the database name, then click “Go”. I have put **demosql** as the database name. At this point, you’ll get your newly created database link as shown in the image below:



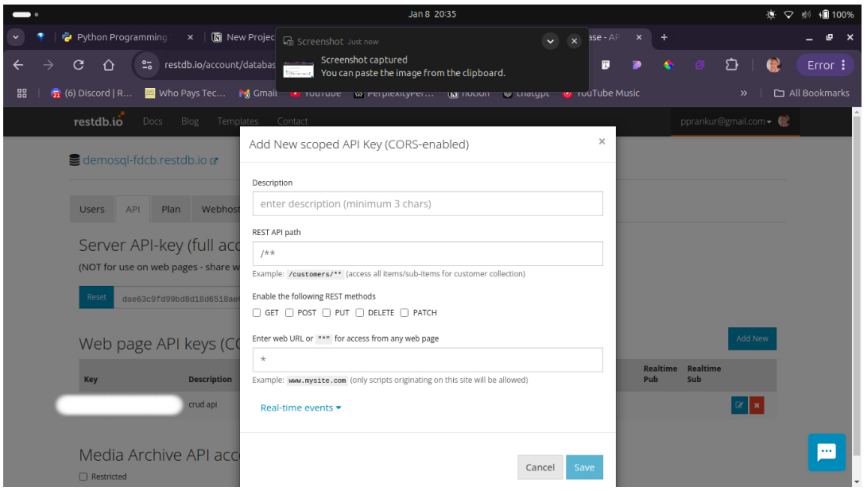
Now Click on the database URL it will take you to this page shown in the image :



Now it is time to make an API Key for accessing the database. To do this, click on **Settings** and it will take you to a new page shown below:



On this page click on the **Add New** button it will open a pop up shown below in the image:



Now you can configure your API actions here like GET, POST, PUT, and DELETE, name it whatever you want, and save it. Your database is now ready to interact via the REST API.

Copy the database URL and API KEY and put it into the .env file.

You can add tables, define the schema with columns and data types (for example, VARCHAR, INTEGER), and populate data manually or via uploads (Excel, CSV, or JSON). For this project, we've added 21 records.

4. Set Up the LLM for Action:

This part is pivotal for the project, as we're setting up the LLM (Large Language Model) to handle the conversion of NLP (plain English) queries into SQL queries.

Numerous LLMs are available in the market, each with its strengths. While some are free, others are paid, which made selecting the right one for this project a challenge.

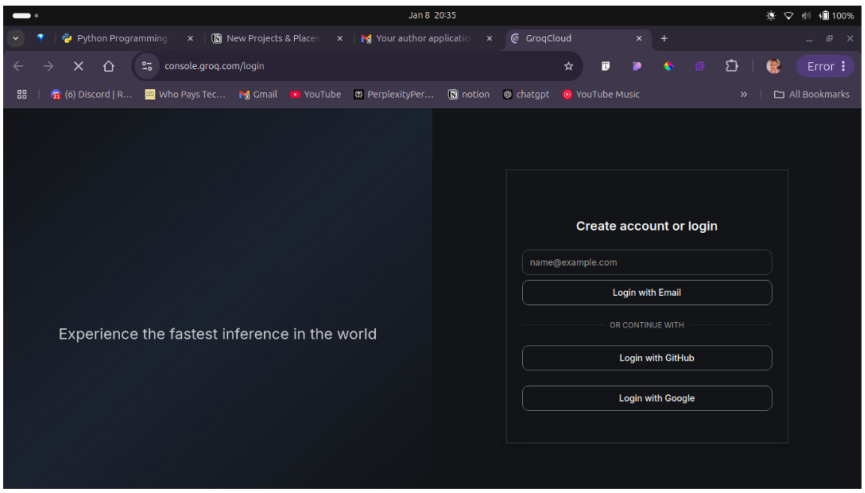
After extensive experimentation, I chose the **Groq Adapter** because:

- It consolidates various LLMs under a single platform.
- It provides access through a unified API key.
- It's compatible with CopilotKit.

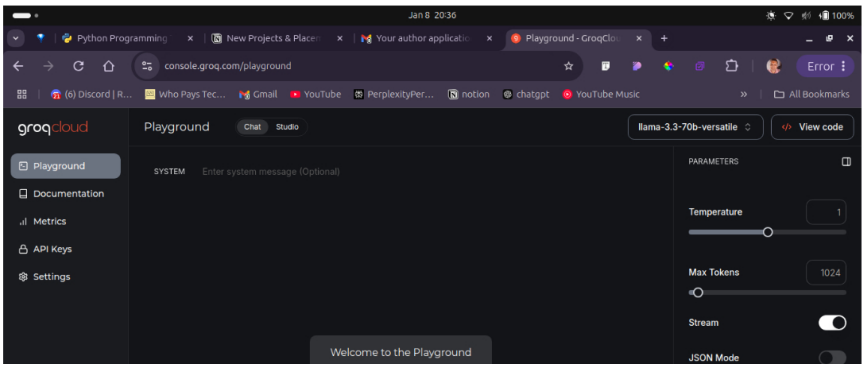
How to Set Up Groq Cloud

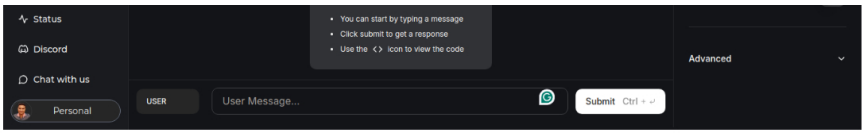
To get started with Groq Cloud, visit its website and either login if already have an account or create a new account if you're new. Once logged in, navigate to the Groq Dashboard.

This is the homepage of groq cloud:

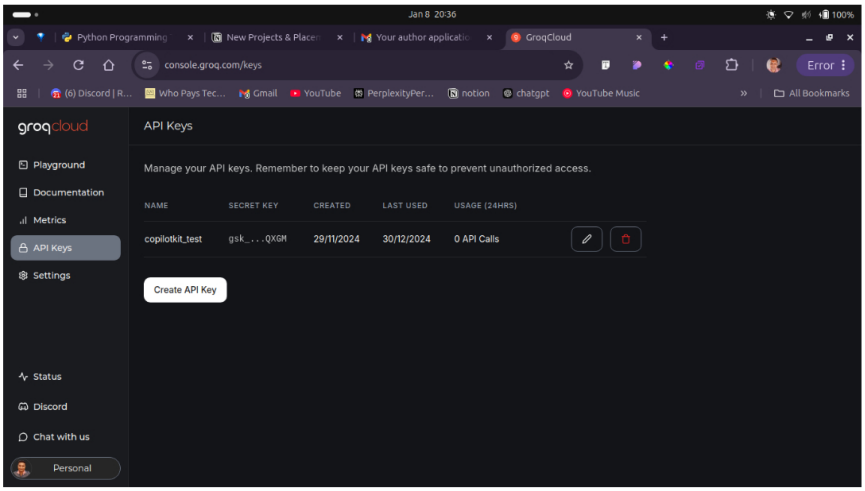


Once logged in, a new page will open that'll look like this:

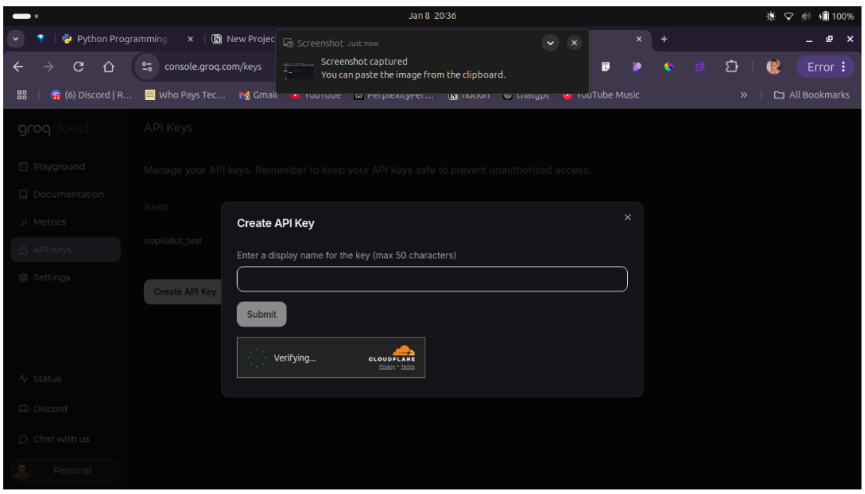




As you can see, the sidebar has an API Keys link. Click on it, and it will open a new page as shown in the image below. You can also select any LLM of your choice which is given at the top right before the view code option.



Here, click on the Create API Key button it will open a pop up like you see below. Just enter the name of your API key and click on Submit it will create a new API key for you. Then copy this API key and paste it inside your .env file.



To enable seamless access to various LLMs on Groq Cloud, generate an API key by going to the Groq API Keys section. Create a new API key specifically for the LLM, ensuring that it is properly configured.

With the LLM set-up and all components ready, you are now prepared to build the project.

build the project.

Structure and Features of the App

We will approach this project in a straightforward way, focusing on simplicity and functionality. The primary goal is to create a basic webpage that allows us to:

- Verify if our API calls were successful.
- View the data received from the API.
- Interact with the CopilotKit chatbot integrated into the front end.

Webpage Structure

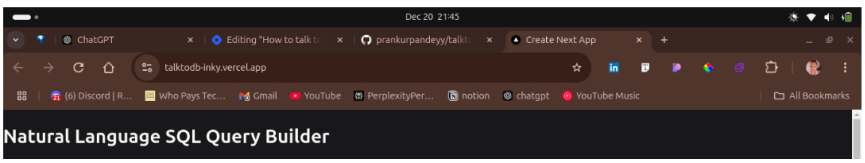
Since we have already set up the **Next.js app**, the next step is to build a minimalistic webpage comprising:

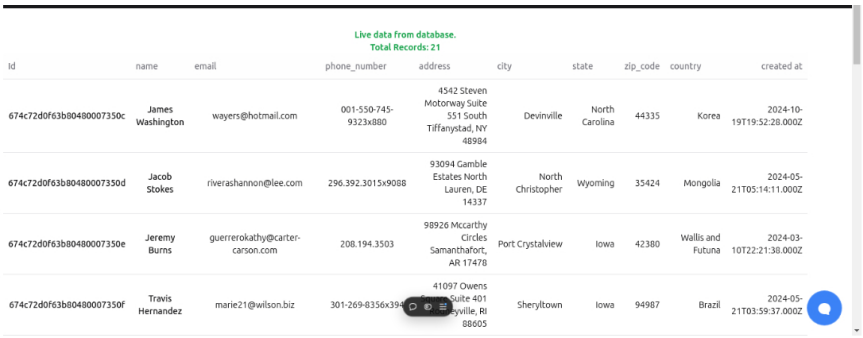
1. **Header Section:** Displays the title of the application.
2. **Main Area:**
    - **Tables:** Show the data fetched from the database.
    - **Status Indicators:** Show the status of API calls and database operations. If there are any issues, such as API or database failures, errors will be displayed in **red text** for clarity.

Key Features

- **Error Handling:** Any failures, such as API or database issues, will be clearly marked with red text for immediate visibility.
- **Data Presentation:** For demonstration purposes, the entire database will be displayed in neatly structured tables.
- **CopilotKit Chatbot Integration:** This chatbot will be configured to allow natural language interactions with the database. The **blue-colored ball** on the page represents the **CopilotKit chatbot**. This chatbot is the key interface for interacting with the database.
    - Using natural language queries, we can ask questions about the database data.
    - The chatbot processes these queries, converts them into SQL queries, and fetches the results seamlessly.

The frontend will look something like this:





How to Build the Back End

Before we start building the back end, you'll need to put all important credentials into your **.env** file which will look something like this:

```
NEXT_PUBLIC_COPILOTKIT_BACKEND_URL=http://localhost:3000/api/copilotkit
NEXT_PUBLIC_GROQ_CLOUD_API_KEY=
NEXT_PUBLIC_RESTDB_API_KEY=
NEXT_PUBLIC_RESTDB_BASE_URL=https://demosql-fdcb.restdb.io/rest/demo-data
```

So what are all these? Let's go through them one by one:

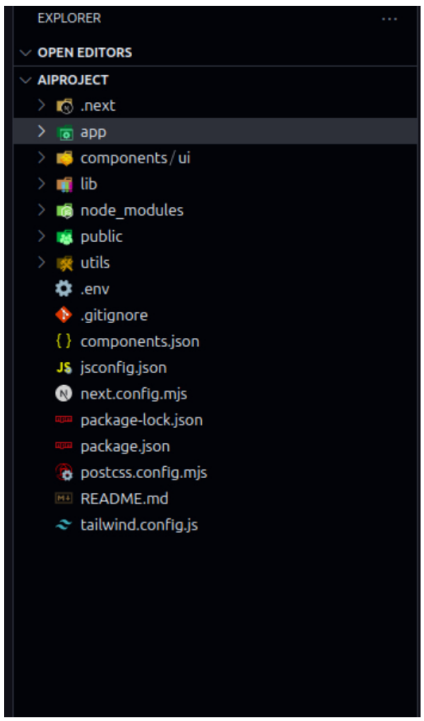
1. `NEXT_PUBLIC_COPILOTKIT_BACKEND_URL= http://localhost:3000/api/copilotkit`: This specifies the base URL for the CopilotKit backend API.
    - The `NEXT_PUBLIC_` prefix makes this variable accessible both on the server side and in the client-side code of a Next.js application.
    - The value `http://localhost:3000/api/copilotkit` indicates the API is running locally during development.
2. `NEXT_PUBLIC_GROQ_CLOUD_API_KEY=`: This variable is intended to store an API key for a GROQ Cloud service. GROQ Cloud could be related to querying or data processing you will have to paste your own Groq API key.
    - The variable is empty, indicating the API key is not set yet. It will likely need to be filled in with the appropriate value before the application can access the GROQ Cloud service.
3. `NEXT_PUBLIC_RESTDB_API_KEY=`: Intended to hold the API key for accessing a **RESTdb** service. You will have to paste your own Groq API key.
    - RESTdb is a database service that provides APIs for database interactions.
    - The variable is also empty, meaning the key must be filled in with a valid API key for the application to authenticate and interact with the RESTdb service.
4. `NEXT_PUBLIC_RESTDB_BASE_URL= https://demosql`

4. `NEXT_PUBLIC_RESTDB_BASE_URL= https://demosql-fdcb.restdb.io/rest/demo-data` : Defines the base URL for interacting with the RESTdb database. This URL will be created when you make your database. Here, I have given the URL of my database.
    - The value `https://demosql-fdcb.restdb.io/rest/demo-data` points to a specific RESTdb database endpoint called `demo-data`.
    - This could be the endpoint where the application fetches or manipulates demo data for testing or development.

We have successfully added the environment variables to our project. Now, it's time to configure the CopilotKit API backed.

How to Configure the CopilotKit Back End

Open your Next.js app in any code editor – I prefer VSCode – and go to the root folder, which looks like this:



Inside the app folder, make a new folder called `api`. Inside the API folder, make another folder called `copilotkit`. Then in there, make a new file called `route.js` and inside the file paste this code:

```
import {
  CopilotRuntime,
  GroqAdapter,
  copilotRuntimeNextJSAppRouterEndpoint,
} from "@copilotkit/runtime";

import Groq from "groq-sdk";
```

```
import Groq from "groq-sdk";

const groq = new Groq({ apiKey: process.env.NEXT_PUBLIC_GROQ_CLOUD_API_KEY });
console.log(process.env.NEXT_PUBLIC_GROQ_CLOUD_API_KEY);
const copilotKit = new CopilotRuntime();

const serviceAdapter = new GroqAdapter({
  groq,
  model: "llama-3.1-70b-versatile",
});

export const POST = async (req) => {
  const { handleRequest } = copilotRuntimeNextJSAppRouterEndpoint({
    runtime: copilotKit,
    serviceAdapter,
    endpoint: "/api/copilotkit",
  });

  return handleRequest(req);
};
```

Here's a detailed explanation of each part:

This code defines a server-side handler for a Next.js API route using CopilotKit and Groq SDKs. It sets up a runtime environment to process requests to a specified endpoint.

1. Imports:

```
import {
  CopilotRuntime,
  GroqAdapter,
  copilotRuntimeNextJSAppRouterEndpoint,
} from "@copilotkit/runtime";

import Groq from "groq-sdk";
```

- `CopilotRuntime` and `GroqAdapter`: These are classes from the CopilotKit library used to set up and configure the runtime environment and adapters for AI-based services.
  - `CopilotRuntime`: A runtime environment to manage the CopilotKit operations.
  - `GroqAdapter`: Adapts and connects a Groq service (used for querying or data processing) with CopilotKit.
- `copilotRuntimeNextJSAppRouterEndpoint`: A utility function to create a handler for a Next.js App Router API endpoint that integrates CopilotKit.
- `Groq` from `"groq-sdk"`: A library for interacting with Groq services is initialized here for querying or processing data.

2. Initialize Groq:

```
const groq = new Groq({ apiKey: process.env.NEXT_PUBLIC_GROQ_CLOUD_API_KEY });
console.log(process.env.NEXT_PUBLIC_GROQ_CLOUD_API_KEY);
```

- `Groq` Initialization:
  - The `Groq` an object is created with an API key (`NEXT_PUBLIC_GROQ_CLOUD_API_KEY`) fetched from environment variables.
  - This key authenticates the app with the Groq Cloud service.
- `console.log( process.env.NEXT _PUBLIC_GROQ_CLOUD_API_KEY)`: Logs the API key to the server console. **Note:** Avoid logging sensitive data in production to ensure security.

3. Initialize CopilotKit Runtime

```
const copilotKit = new CopilotRuntime();
```

- `CopilotRuntime` Initialization: Creates an instance of CopilotKit's runtime environment to manage CopilotKit's features and services.

4. Configure Service Adapter

```
const serviceAdapter = new GroqAdapter({
  groq,
  model: "llama-3.1-70b-versatile",
});
```

- `GroqAdapter`:
  - Configures an adapter to connect CopilotKit with Groq.
  - The `model` parameter specifies the AI model to use. Here, it is `"llama-3.1-70b-versatile"`, a versatile language model with 70 billion parameters.

5. Exported POST Handler

```
export const POST = async (req) => {
  const { handleRequest } = copilotRuntimeNextJSAppRouterEndpoint({
    runtime: copilotKit,
    serviceAdapter,
    endpoint: "/api/copilotkit",
  });

  return handleRequest(req);
};
```

- Defines a `POST` handler for a Next.js App Router API endpoint.
- **Key Components:**

1. `copilotRuntimeNextJSAppRouterEndpoint`:
    - Sets up the handler for the `/api/copilotkit` endpoint.
    - Takes `runtime` (CopilotKit) and `serviceAdapter` (GroqAdapter) as inputs to configure the endpoint's behaviour.
2. `handleRequest`:
    - A function that processes incoming HTTP requests (in this case, `POST` requests).
    - This allows the CopilotKit runtime and service adapter to handle requests dynamically.

- `return handleRequest(req);`: Invokes the handler and processes the incoming request (`req`), returning the appropriate response.

How it all works:

1. The Groq SDK is initialized with an API key for authentication.
2. A CopilotKit runtime is set up.
3. A GroqAdapter connects the runtime to the Groq service with a specified AI model.
4. The `/api/copilotkit` endpoint is configured to handle POST requests, pass the requests to CopilotKit's runtime, and return the processed response.

With this setup, you have successfully integrated CopilotKit into your Next.js application. The backend is now fully functional, enabling seamless communication with the database via REST APIs and the CopilotKit interface.

How to Build the Front End

For the front end, we'll keep it as simple as we can. We just need a few things to get this project done: we need a Header component and a Table component.

1. **Header component**: To display the title or description of the application.
2. **Table component**: To visualize the data fetched from the database.

To achieve this, we'll use ShadCN, a popular frontend component library known for its clean design and ease of use.

ShadCN provides pre-built components that help speed up development without compromising on quality. By leveraging this library, we can focus on

functionality while ensuring the UI looks polished and professional.

How to Install ShadCN in a Next Project

Run the following command to install ShadCN components:

```
npx shadcn@latest init
```

This command:

- Initialize ShadCN in your project.
- Creates a `components` folder for storing ShadCN components.
- Updates the `tailwind.config.js` file with required configurations.

You will be asked a few questions to configure `components.json`:

```
Which style would you like to use? › New YorkWhich
color would you like to use as base color? › Zinc
Do you want to use CSS variables for colors? › no / yesadd components
```

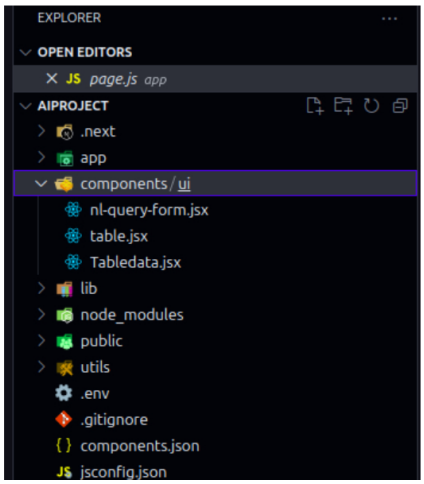
To add specific components, use the following command:

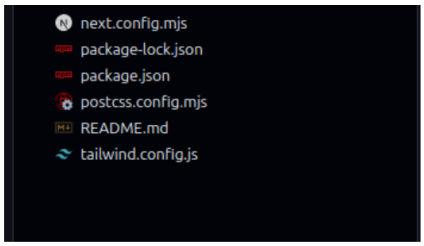
```
npx shadcn@latest add <component-name>
```

For example, to add a table component:

```
npx shadcn@latest add table
```

The `components` folder now contains a ready-to-use `button` component.





In the frontend, we have a `components` folder that contains the Table component. This component is responsible for displaying the database data in a structured tabular format.

Apart from the `Table` component, there are two additional files in the front end. These files serve different purposes and will be integrated later in the project for specific functionalities.

This modular structure ensures the front end remains clean and organized, making it easier to manage and expand as needed.

Let's explore each file:

1. **Table.jsx:** This file is auto-generated by ShadCN when we installed the Table component. It contains the default configuration for the table component provided by the ShadCN library. **Do not modify this file**, as it is essential for the component's proper functionality.
2. **Tabledata.jsx:** This file is where we populate the table with data fetched from the database through API calls. The `Tabledata.jsx` file bridges the gap between the backend API and the frontend table display.

Let's take a closer look at the code:

```
import {
  Table,
  TableBody,
  TableCaption,
  TableCell,
  TableFooter,
  TableHead,
  TableHeader,
  TableRow,
} from "@/components/ui/table";

export function Tabledata({ data }) {
  return (
    <Table className="text-center">
      <TableCaption className="text-sm text-green-600 font-bold ml-8">
        Live data from database.
      </TableCaption>
      <TableHeader>
        <TableRow className="text-center ">
          <TableHead>Id</TableHead>
          <TableHead>name</TableHead>
          <TableHead>email</TableHead>
          <TableHead>phone_number</TableHead>
          <TableHead>address</TableHead>
          <TableHead>city</TableHead>
```

```
          <TableHead>state</TableHead>
          <TableHead>zip_code</TableHead>
          <TableHead>country</TableHead>
          <TableHead className="text-right">created at </TableHead>
        </TableRow>
      </TableHeader>
      <TableBody>
        {data.map((db) => (
          <TableRow key={db._id}>
            <TableCell className="font-medium text-wrap w-12">
              {db._id}
            </TableCell>
            <TableCell className="font-medium">{db.name}</TableCell>
            <TableCell>{db.email}</TableCell>
            <TableCell>{db.phone_number}</TableCell>
            <TableCell className="text-right">{db.address}</TableCell>
            <TableCell className="text-right">{db.city}</TableCell>{" "}
            <TableCell className="text-right">{db.state}</TableCell>
            <TableCell className="text-right">{db.zip_code}</TableCell>{" "}
            <TableCell className="text-right">{db.country}</TableCell>
            <TableCell className="text-right">{db.created_at}</TableCell>
          </TableRow>
        ))}
      </TableBody>
    </Table>
  );
}
```

This code renders a styled, dynamic table with data passed from a database or API.

- **Imports**: Uses custom table components (`Table`, `TableRow`, `TableCell`, and so on) from `@/components/ui/table`.
- **Props**: Accepts a `data` prop, an array of objects representing table rows.
- **Table Caption**: Displays a caption, "Live data from database," styled with Tailwind CSS.
- **Table Header**: Defines column headers such as `Id`, `name`, `email`, and more.
- **Dynamic Rows**: Maps over the `data` array to generate `TableRow` elements dynamically, using `_id` as the unique key.
- **Data Cells**: Displays object fields (`_id`, `name`, `email`, and so on) in `TableCell` components with custom styles.
- **Tailwind CSS**: Styles applied for alignment, font weight, and spacing.

NLQueryForm.jsx

In this file, we handle the API calls, define CopilotKit actions, and pass the fetched data to the Table component. This file acts as the central logic hub for connecting the backend API, AI actions, and the frontend display.

Key functionalities of `NLQueryForm.jsx`:

1. **API integration**: Fetches data from the database and handles errors or loading states.
2. **CopilotKit actions**: Defines AI actions that allow querying and

interacting with the database using natural language.

3. **Data passing**: Sends the processed data to the `Table` component for display.

Below is the code:

```
"use client";
import React, { useState, useEffect } from "react";
import { useCopilotReadable, useCopilotAction } from "@copilotkit/react-core";
import axios from "axios";
import { Tabledata } from "./Tabledata";

function NLQueryForm() {
  const [nlQuery, setNlQuery] = useState("");
  const [data, setData] = useState([]);
  console.log("🚀 ~ NLQueryForm ~ data:", data);
  const [error, setError] = useState("");
  const [loading, setLoading] = useState(true);

  const API_KEY = process.env.NEXT_PUBLIC_RESTDB_API_KEY;
  const BASE_URL = process.env.NEXT_PUBLIC_RESTDB_BASE_URL;
  console.table({ API_KEY, BASE_URL });
  useEffect(() => {
    async function fetchData() {
      if (!API_KEY || !BASE_URL) {
        setError("API configuration is missing");
        setLoading(false);
        return;
      }
      try {
        const response = await axios.get(BASE_URL, {
          headers: {
            "x-apikey": API_KEY,
            "Content-Type": "application/json",
          },
        });
        setData(response.data);
        setLoading(false);
      } catch (error) {
        console.error("Error fetching data:", error);
        setError(
          error instanceof Error ? error.message : "An unknown error occurred"
        );
        setLoading(false);
      }
    }
    fetchData();
  }, [API_KEY, BASE_URL]);

  useCopilotReadable({
    description: "Query database with detailed information",
    value: JSON.stringify(data.slice(0, 25)),
  });
  useCopilotAction({
    name: "fetchData",
    description: "Search and filter data based on natural language query",
    parameters: [
      {
        name: "nlQuery",
        type: "string",
        description: "Natural language search term for database",
        required: true,
      },
    ],

    handler: async ({ data }) => {
      setNlQuery(data);
      return JSON.stringify(data);
    },
  });

  if (loading) return <div>Loading...</div>;

  return (
```

```
    <div>
      {error && <p style={{ color: "red" }}>{error}</p>}
      <div>
        <p className="text-sm text-green-600 font-bold text-center">
          Live data from database.
        </p>
        <p className="text-sm text-green-600 font-bold text-center">
          Total Records: {data.length}
        </p>
        <Tabledata data={data} />
      </div>
    </div>
  );
}
export default NLQueryForm;
```

Here's a detailed explanation of the `NLQueryForm` component:

Imports and Dependencies:

- Utilizes React for state management (`useState`) and side effects (`useEffect`).
- Imports `axios` for HTTP requests.
- Imports `useCopilotReadable` and `useCopilotAction` from `@copilotkit/react-core` to integrate CopilotKit functionality.
- Imports a custom `Tabledata` component for rendering data.

Component Setup:

- Defines a functional React component `NLQueryForm`.
- Initializes state variables:
  - `nlQuery`: Holds the natural language query input.
  - `data`: Stores fetched data from the API.
  - `error`: Stores any errors that occur during data fetching.
  - `loading`: Tracks the loading state of the component.

API Configuration:

- Fetches API keys and base URL from environment variables (`NEXT_PUBLIC_RESTDB_API_KEY` and `NEXT_PUBLIC_RESTDB_BASE_URL`).
- Logs these values for debugging purposes using `console.table`.

Data Fetching:

- Uses `useEffect` to fetch data from the API on the initial render.
- Makes a GET request to the API using `axios` with required headers.
- Updates `data` with the response and stops the loading state.
- Handles errors by logging them and updating the `error` state.

CopilotKit Integration:

- `useCopilotReadable`: Exposes a readable description and a slice of the first 25 records of `data`.
- `useCopilotAction`: Defines a CopilotKit action named `fetchData` which:
  - Accepts a natural language query (`nlQuery`) as input.
  - Updates the `nlQuery` state and returns it as a string.

Conditional Rendering:

- Displays a loading message (`Loading...`) if `loading` is true.
- Displays an error message in red text if an error occurs.

Rendering:

- Shows a message indicating live data and the total record count.
- Passes the `data` state to the `Tabledata` component for rendering.

Export:

- Exports the `NLQueryForm` component as the default export.

Page.js

Now go to the `page.js` file inside the app folder and add this code:

```
"use client";

import NLQueryForm from "@/components/ui/nl-query-form";
import { CopilotPopup } from "@copilotkit/react-ui";

export default function Home() {
  return (
    <div className="min-h-screen bg-background">
      <header className="bg-primary text-primary-foreground py-6">
        <div className="container">
          <h1 className="text-3xl font-bold">
            Natural Language SQL Query Builder
          </h1>
        </div>
      </header>
      <main className="container py-8">
        <NLQueryForm />
      </main>

      <CopilotPopup
        instructions={
          "You are assisting the user as best as you can. Answer in the best way
        }
        labels={{
          title: "Popup Assistant",
          initial: "Need any help?",
        }}
      />
```

```
      </div>
    );
  }
```

Here's a simple explanation of the code above:

- **Client-Side Rendering**:
  - `"use client";` indicates the file is using React's client-side rendering.
- **Importing Components**:
  - `NLQueryForm` is imported from a local component directory to be used in the app.
  - `CopilotPopup` is imported from the `@copilotkit/react-ui` package for displaying an interactive popup.
- **Main Function**:
  - `Home` is a React functional component that defines the UI for the home page.
- **Page Layout**:
  - A full-page container (`min-h-screen`) with a background color (`bg-background`) wraps all content.
- **Header**:
  - Contains a title with the text **"Natural Language SQL Query Builder"**.
  - Styled with a primary background and text colors (`bg-primary`, `text-primary-foreground`).
- **Main Content**:
  - Renders the `NLQueryForm` component inside a container with padding (`py-8`).
- **Popup Component**:
  - Adds a `CopilotPopup` at the bottom with:
    - **Instructions**: Describes the assistant's role.
    - **Labels**: Includes a title and initial message for the popup.
- **Purpose**:
  - The page is designed to let users interact with a natural language SQL query builder and receive assistance via a popup.

Configuring CopilotKit for the Whole App

This is going to be the last step of building the application. Navigate to the `layout.js` file and add this code:

```
import "./globals.css";
import { CopilotKit } from "@copilotkit/react-core";
import "@copilotkit/react-ui/styles.css";
export const metadata = {
  title: "Create Next App",
  description: "Generated by create next app",
};

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body>
        <CopilotKit runtimeUrl="/api/copilotkit">{children}</CopilotKit>
      </body>
    </html>
  );
```

Here's what's going on in this code:

- **Imports:**
  - `./globals.css`: Imports global CSS styles for the application.
  - `@copilotkit/react-core`: Imports the core functionality of CopilotKit.
  - `@copilotkit/react-ui/styles.css`: Includes predefined styles for the CopilotKit UI components.
- **Metadata:**
  - The `metadata` object defines the app's title and description, which are useful for setting meta tags in the generated HTML for SEO and user information.
- **RootLayout function:**
  - This function serves as the root layout wrapper for the application. It ensures consistent structure across all pages and integrates the CopilotKit runtime.
- **Structure:**
  - The layout returns an `<html>` element with a `lang` attribute set to `en` for English.
  - Inside the `<body>` tag, the CopilotKit component is wrapped around the `children` prop.
    This setup:
    - Connects the app to the CopilotKit runtime using the API endpoint `/api/copilotkit`.
    - Provides access to CopilotKit's functionality, such as handling natural language queries, throughout the

application.

Some Important Notes

Designing and deploying a database can take various forms, depending on the tools and requirements. For this project, I have chosen the simplest and most accessible approach.

Why CopilotKit?

CopilotKit is a powerful tool that converts NLP queries into actionable backend code. If you have an alternative that works similarly, feel free to use it. It bridges the gap between natural language input and technical execution, making it ideal for projects like this.

Why GroqCloud?

I selected **GroqCloud** because it's free and provides access to multiple LLMs with a single API key. While you can opt for alternatives like ChatGPT, note that they may require paid plans. GroqCloud's versatility and affordability make it perfect for this tutorial.

Database Considerations

The size of your database can vary from very small to enormous. However, interacting with the database depends on the token limits of the LLM you're using.

Since I'm working with free-tier tools, my focus is on a small database to ensure seamless interactions.

Security Best Practices

Never expose your credentials publicly. Always store sensitive information like API keys in an `.env` file to keep your project secure.

Future Enhancements

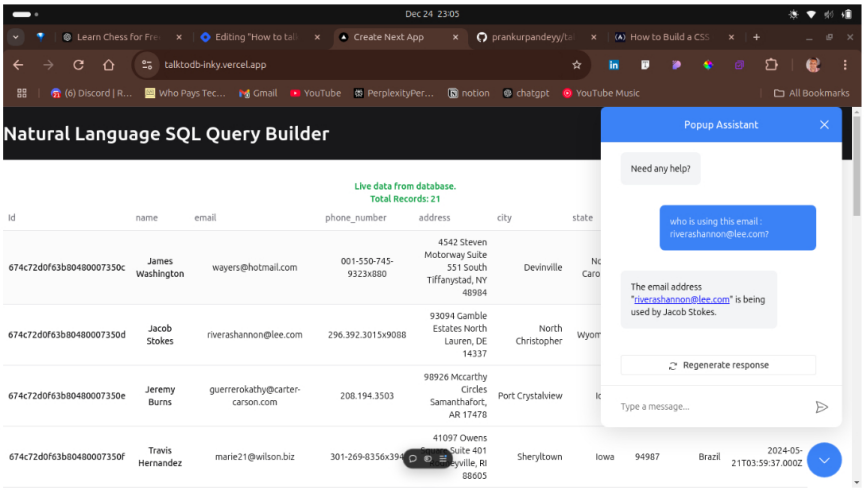
While this tutorial focuses on setting up and querying a database, the potential of CopilotKit extends to **CRUD operations** (Create, Read, Update, Delete). In my next tutorial, I will demonstrate how to implement full CRUD operations using CopilotKit for a more dynamic and functional application.

Playing with the Database

You can explore the live project via the following link and ask any questions related to the database data: live link .

For a deeper understanding of the code, here's the GitHub repository link: github .

Also, here's a screenshot demonstrating its practical use. In this example, instead of writing a plain SQL query like `SELECT * FROM demo_data WHERE email = ' riverashannon@lee.com ';` to extract the name of the person, we used an NLP query to achieve the exact same result.



Conclusion

I hope you've enjoyed building this simple AI chatbot to interact with the database. In this project, we've used a simple SQL database, but you can apply this approach to any database as long as you can retrieve the data.

In the future, I plan to implement many new projects involving AI and other tools. AI tools are truly game-changing in the IT field, and I look forward to providing you with more detailed insights and practical implementations of the latest tools emerging in the space.

So this is the end from my side. If you found this article useful, then do share it and connect with me – I am open to opportunities:

- Follow Me on X: Prankur's Twitter
- Follow me on LinkedIn: Prankur's Linkedin
- Look at my Portfolio here: Prankur's Portfolio

Prankur Pandey

A 4-year-old freelance MERN Stack software developer, skilled in web and mobile app development. I am open to new work & opportunities.

If you read this far, thank the author to show them you care. Say Thanks

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) charity organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public.

Donations to freeCodeCamp go toward our education initiatives, and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Trending Books and Handbooks

- Learn CSS Transform
- Build an AI Chatbot
- Python Code Examples
- HTTP Networking in JS
- Learn Algorithms in JS
- Learn PHP
- Learn Swift
- Learn Node.js
- Learn Solidity
- Learn JS Modules
- REST API Best Practices
- Learn to Build REST APIs
- Command Line for Beginners
- Learn to Build GraphQL APIs
- Distributed Systems Patterns
- Build a Static Blog
- What is Programming?
- Open Source for Devs
- Write React Unit Tests
- How to Write Clean Code
- Learn Java
- Learn Golang
- Learn CSS Grid
- Learn Express.js
- Learn Apache Kafka
- Front-End JS Development
- Intermediate TS and React
- Intro to Operating Systems
- OSS Security Best Practices
- Software Architecture Patterns

Mobile App

Download on the App Store

GET IT ON Google Play