Table of contents

Introduction

Welcome to the documentation for the WhatsApp Webhook implementation for a ChatGPT powered chatbot! This document serves as a comprehensive guide to understanding the Node.js code powering the webhook.

The webhook service is essential for connecting the external systems which are the WhatsApp API, OpenAI's GPT, and backend data storage systems. It handles packet reception, interaction with external services, database management, and caching, enabling seamless and efficient operation of chatbot.

Project Setup

To set the chatbot up a webhook service, postgreSQL database, Redis cache and PostHog project will be required.

Create WhatsApp API Project

To connect the webhook to the WhatsApp API a project will need to be created to generate a token. In order to generate a WhatsApp Project API Token, it is necessary to establish a Facebook Business account.

You can view the documentation for setting up a project and obtaining a WhatsApp Cloud API Token at the following link:

https://developers.facebook.com/docs/whatsapp/

Create postgreSQL Database And Redis Cache

Creating a PostgreSQL database is essential for enabling the logging and caching functionalities of the webhook. Additionally, deploying a Redis query & response cache database can significantly reduce ongoing costs and enhance response times.

The WhatsApp Bot is compatible with any PostgreSQL or PostgreSQL-compatible database, including options such as Amazon RDS or Azure Database for PostgreSQL.

It is recommended to use Neon (neon.tech) for PostgreSQL services. Neon provides many of the same features as Vercel, along with a very generous free tier.

For the Redis cache a free option is Upstash, which also offers a serverless and distributed Redis database solution.

With both of these services, maintenance is virtually nonexistent, eliminating the need for manual upgrades or downgrades based on demand.

OpenAI ChatGPT Setup

You will need to obtain your own OpenAI API Key, which can be done on the following webpage:

https://platform.openai.com/

Webhook Service Setup

The project was developed on Glitch.com which provides a free webhook service. However, this service is slow and currently only supports Node.js up to version 16. A faster webhook service is recommended to improve the response times of the chatbot.

Node Version And Required Packages:

The webhook service will need to run Node.js version 16 or higher however the code was developed on version 16 and may have errors on newer versions.

The following Node.js packages will be required to run the code:

- axios: ^0.28.0,
- body-parser: ^1.20.2,
- express: ^4.18.2,
- request: ^2.88.2,
- openai: ^4.28.0,
- encoding: ^0.1.13,
- drizzle-kit: ^0.20.14,
- drizzle-orm: ^0.29.3,
- postgres: ^3.4.3,
- cosine-similarity: ^1.0.1,
- node-fs: ^0.1.7,
- ioredis: ^5.3.2,
- posthog-node: ^4.0.0,
- bcrypt: ^5.1.1,

Environmental Variables:

The following environmental variables will need to be defined for the code to work:

- WHATSAPP_TOKEN: This is the token provided by the WhatsApp API.
- VERIFY_TOKEN: This is the token you create to verify that the webhook.

- OPENAI_API_KEY: The API key to openAI.
- ASSISTANT_ID: The ID of the openAI assistant.
- DATABASE_URL: The URL of the postgreSQL user ID database.
- REDIS_URL: The URL of the Redis cache.
- REDIS_TOKEN: The token for accessing the cache.
- POSTHOG_TOKEN: The token for accessing the PostHog project.

SQL Database Commands

The following terminal commands will need to be run at the creation of the webhook to set up the user ID database.

Drop the database:

This command is used to delete the database associated with the Drizzle project. It essentially drops all tables and removes the database schema, effectively wiping the database clean. This will only need to be run if the database was already created previously and changes have occurred.

```
drizzle-kit drop
```

Generate the database:

This command is used to regenerate the database schema based on the project's configuration and schema files. In this case, generate:pg specifies that the schema should be generated for a PostgreSQL database.

```
drizzle-kit generate:pg
```

Push updates to the database:

This command is used to push the database schema and any pending migrations to the PostgreSQL database. It specifies the schema file (src/schema.js), the database driver (pg for PostgreSQL), and the connection string necessary to connect to the database. The connection string contains information such as the username, password, host, and database name. Once executed, this command updates the database schema and applies any pending migrations.

```
drizzle-kit push:pg --schema src/schema.js --driver pg --connectionString
"DATABASE_URL_HERE"
```

PostHog Service Setup

PostHog is an open-source product analytics platform that allows businesses to understand user behavior within their applications or services. It provides features for event tracking, user analytics, and cohort analysis, among others.

Businesses use PostHog to gain insights into how users interact with their products, identify areas for improvement, and make data-driven decisions to optimize user experience and drive growth.

In the context of the WhatsApp chatbot, PostHog would is used to log events relating to user to queries and responses. This data can analysed then used to refine the chatbot's functionality, improve user engagement, and enhance overall user satisfaction.

Setup Posthog Project:

PostHogs free tier allows for only 1 project which is enough for this use case.

PostHog can be found here: https://eu.posthog.com/login?next=/

Main.js Code Documentation

This section provides an in-depth explanation of the code used in the "main.js" file of a WhatsApp chatbot. This section will cover various aspects such as message handling, chatbot interactions, webhook management, and verification processes.

Code Sections

Imports & Global Variables

At the top of the file the required dependencies are imported.

```
import request from "request";
import OpenAI from "openai";
import express from "express";
import body_parser from "body-parser";
import axios from "axios";
import fs from "node-fs";
import { PostHog } from 'posthog-node';
import bcrypt from 'bcrypt';
import similarity from './similarity.js';
import { log } from './logging.js';
import { checkDatabase, updateDatabase, checkCache, updateCache, resetUsers,
resetCache} from './database.js';
```

Dependency functionalities:

- The request library is used to make HTTP requests.
- The OpenAI library is used for creating threads and communicating with the user.
- The express library is used to set up a basic HTTP server.
- The body-parser library is used to parse the JSON, buffer, string and URL encoded data submitted using HTTP POST requests.
- The axios library is used to send asynchronous HTTP requests.
- The node-fs library is used for local text file management.
- The PostHog library is used for PostHog logging.

- The bCrypt library is used for hashing and comparing the users phone numbers.
- The similarity library is importing the local NPL functions.
- The logging library is importing the local logging functions.
- The database library is importing the function from the local file. These functions are used to manage the cache and logging database.

Global Variables:

```
// Assign global variables
const app = express().use(body_parser.json());
const openai = new OpenAI({apiKey: process.env['OPENAI_API_KEY']});
const token = process.env.WHATSAPP_TOKEN;
const assistant_id = process.env.ASSISTANT_ID;
const connectionString = process.env.DATABASE_URL;
const saltRounds = 10;
```

- app: An instance of the Express framework, configured to use body_parser middleware for parsing JSON request bodies.
- openai: An instance of the OpenAI class initialized with the API key fetched from the environment variables.
- token: The WhatsApp API access token retrieved from environment variables.
- assistant_id: The ID of the OpenAI Assistant obtained from environment variables.
- connectionString: The connection string for the database fetched from environment variables.
- saltRounds: The number of salt rounds used for bcrypt hashing.

These imports and variable assignments set up the foundational components required for the chatbot application, including API interaction, server setup, database operations, and security measures like data hashing.

Sending Message Function

This function is responsible for sending a message to a user via the WhatsApp API and logging the message sent using PostHog.

Asynchronous Function:

The function is defined as async as it contains asynchronous operations that involve for external API calls.

Sending Message:

It uses Axios, an HTTP client for Node.js, to send a POST request to the WhatsApp API to send a message to the specified phone number using the WhatsApp messaging product.

It constructs the URL for the API request using the phone number ID and an access token. Then sends a POST request to the constructed URL with the message data in the request body.

It includes error handling to catch any errors that may occur during the request, such as network errors or invalid API responses.

```
axios({
  method: "POST",
  url: "https://graph.facebook.com/v12.0/" + phone_number_id + "/messages?access_token=" + token,
  data: { messaging_product: "whatsapp", to: from, text: { body: message },},
  headers: { "Content-Type": "application/json" },
}).catch(function (error) {
  if (error.response) {
    console.log("ERROR: Error occurred while sending user's message. Check WhatsApp number and API access token!");
    process.exit();
  }
});
```

Logging Message:

It uses PostHog, a product analytics platform, to log the message sent to the user. It initializes a new instance of PostHog client using the provided token and host.

```
const clientPH = new PostHog(
    process.env.POSTHOG_TOKEN,
    { host: 'https://eu.posthog.com' }
)
```

It captures an event "Message sent to user" with properties including the sender, the message content, and the hashed user's ID.

```
const hashed_id = bcrypt.hashSync(user_number_id, saltRounds);
clientPH.capture({
    distinctId: hashed_id,
    event: 'Message sent to user',
    properties: {
        from: from,
        message: message,
    },
})
```

It shuts down the PostHog client after capturing the event.

```
clientPH.shutdown();
```

Chatbot Function

This function is used for interactions between the WhatsApp API, the openAI chatbot, and a database to provide responses to user queries received via the webhook. It manages user sessions, interacts with the chatbot service to generate responses, and logs relevant information for analyses.

Asynchronous Function:

The function is defined as async as it contains asynchronous operations that involve waiting for external resources like API calls, database calls or file reads.

Initialization:

The function initializes by retrieving the necessary resources and configurations, such as an assistant instance using OpenAI. "openai" is an instantiated client for the OpenAI API, and assistant_id is a variable containing the ID of the chatbot assistant.

```
const assistant = await openai.beta.assistants.retrieve(assistant_id);
```

Checking User in Database:

It checks if the user ID is present in the database. If the user is found in the database, it retrieves the associated thread ID.

```
const result = await checkDatabase(user_number_id);
```

Handling Existing User:

If the user is found in the database, it proceeds to retrieve the existing thread or create a new one if necessary. If the user is not found in the database, it sends GDPR and usage instructions to the user, creates a new thread, and updates the database with the new thread ID.

```
if (found) {
    // Retrieve existing thread or create a new one
    var thread = await openai.beta.threads.retrieve(thread_id);
} else {
    // Handle GDPR message and create a new thread
    // Update User Database with the new thread ID
    var thread = await openai.beta.threads.create();
    await updateDatabase(user_number_id, thread.id);
}
```

Interacting with Chatbot:

If the query is new (i.e., not already processed), it interacts with the chatbot by sending the user's question to the chatbot assistant and retrieving the response. It then waits for the chatbot response to become available and retrieves it from the chat thread.

```
await openai.beta.threads.messages.create(thread.id, {
    role: "user",
    content: user_question,
});

const run = await openai.beta.threads.runs.create(thread.id, {
    assistant_id: assistant.id,
});
```

Sending Response:

Once the chatbot response is obtained, it sends the response back to the user via WhatsApp API.

```
sendMessage(user_number_id, phone_number_id, user_question, from,
chatbot_response);
```

Logging:

It logs relevant information, such as the existence of the query, user details, and thread ID, for analyses or debugging purposes.

```
const logbook = {
    query_exists: queryExists,
    query_message: user_question,
    user_exists: found,
    user_id: user_number_id,
    thread_id: thread.id,
};
log(logbook);
```

Error Handling:

It includes error handling to catch and log any errors that may occur during the execution of the function.

```
} catch (error) {
    console.error(error);
}
```

Webhook

This code is used to handle incoming POST requests to a webhook endpoint ("/webhook") for the WhatsApp API. It checks the validity of incoming requests, processes the messages, generates responses using a chatbot if necessary, and sends the appropriate response back to the user.

Server Initialization:

The code sets up the server to listen for incoming requests on either the port specified in the environment variable PORT or port 1337. It logs a message when the server is successfully started.

```
// Sets server port and logs message on success
app.listen(process.env.PORT || 1337, () => console.log("webhook is
listening"));
```

Webhook Handler:

It defines a route to handle incoming POST requests at the "/webhook" endpoint.

```
app.post("/webhook", (req, res) => {
    // Handler logic goes here
});
```

Parsing Request Body:

Within the route handler, it parses the request body from the POST request.

```
let body = req.body;
```

Checking Incoming Webhook Message:

It checks if the request body contains an "object" field, indicating that it's a valid WhatsApp API webhook event.

```
if (req.body.object) {
    // Webhook event is from WhatsApp API, proceed with handling
} else {
    // Return a '404 Not Found' if event is not from WhatsApp API
    res.sendStatus(404);
}
```

Processing Incoming Message:

If the event is from the WhatsApp API, it further processes the incoming message. It extracts various information from the webhook payload such as the phone number, message text, etc.

```
let phone_number_id =
req.body.entry[0].changes[0].value.metadata.phone_number_id;
let user_number_id = req.body.entry[0].changes[0].value.contacts[0].wa_id;
let from = req.body.entry[0].changes[0].value.messages[0].from; // extract
the phone number from the webhook payload
let user_question =
req.body.entry[0].changes[0].value.messages[0].text.body; // extract the
message text from the webhook payload
```

It then proceeds to send a response to the user explaining it received the users message.

```
// Send message to user
const message = "Answering: " + user_question;
sendMessage(user_number_id, phone_number_id, user_question, from, message);
```

It checks if the query has already been cached.

```
// Check if user question exists in cache
const result = await checkCache(user_question);
let found = result.found;
let response = result.response;
```

Then the chatbot function is called and if the query has been cached then the cached response is forwarded to the user.

```
// Call chatbot function to handle user interaction
await chatbot(user_number_id, phone_number_id, user_question, from, found);

if (found) {
// Send cached response to user
sendMessage(user_number_id, phone_number_id, user_question, from, response);
}
```

After processing the incoming message, it sends an HTTP status code 200 (OK) response to acknowledge receipt of the webhook event.

```
res.sendStatus(200);
```

Webhook Verification

This code sets up a route to handle verification requests from WhatsApp API. It checks if the request contains the correct parameters and verifies the authenticity of the webhook by comparing the provided token with the one stored in the environment variable. If the verification is successful, it responds with the challenge token provided by Meta. Otherwise, it responds with a Forbidden status code.

Route Definition:

The code starts by defining a route for GET requests to "/webhook" using Express.js.

```
app.get("/webhook", (req, res) => {
    // Handler logic goes here
});
```

Extracting Parameters:

Within the route handler, it extracts query parameters from the request URL. These parameters are part of the verification request sent by the WhatsApp API.

```
let mode = req.query["hub.mode"];
let token = req.query["hub.verify_token"];
let challenge = req.query["hub.challenge"];
```

Verification Logic:

It checks if the required parameters (mode and token) are present in the request.

```
if (mode && token) {
    // Verification logic goes here
}
```

Token Verification:

If the required parameters are present, it verifies the mode and token. If the mode is "subscribe" and the token matches the one stored in the environment variable VERIFY_TOKEN, it responds with HTTP status 200 OK and sends the challenge token back to Facebook. If the mode is not "subscribe" or the tokens do not match, it responds with HTTP status 403 Forbidden.

```
if (mode === "subscribe" && token === verify_token) {
    // Respond with challenge token
    res.status(200).send(challenge);
} else {
    // Respond with Forbidden if tokens do not match
    res.sendStatus(403);
}
```

Logging:

It logs "WEBHOOK_VERIFIED" to the console if the verification is successful.

Database Code Documentation

This section will document and explain the code used in the "database.js" file. The file contains the code for retrieving, updating and deleting data in the Redis Caching database as well as the postgresSQL user id database.

Code Sections

Imports & Global Variables

At the top the necessary dependencies are imported for working with PostgreSQL databases, including libraries for hashing, ORM, database connection, cache interaction, and schema definitions. Additionally, it sets up the connection string for connecting to the PostgreSQL database.

```
import bcrypt from "bcrypt";
import { drizzle } from 'drizzle-orm/postgres-js';
import postgres from 'postgres';
import { Redis } from "ioredis";
import { user, logs } from './src/schema.js';
import similarity from './similarity.js';
```

Dependency functionalities:

- bcrypt: The bCrypt library is used for hashing and comparing the users phone numbers.
- drizzle: Drizzle ORM for PostgreSQL database interaction. It provides an object-relational mapping (ORM) interface for interacting with PostgreSQL databases.
- postgres: PostgreSQL client library for Node.js. It allows Node.js applications to interact with PostgreSQL databases.
- Redis: Redis client library for Node.js, used for interacting with Redis cache.
- user and logs: Import of schemas for the user and logs from the local file schema.js. These schemas define the structure of the tables in the database.
- similarity: Import of the similarity function from the similarity.js file. This function is used for comparing strings and performing similarity checks.

```
const connectionString = process.env.DATABASE_URL;
```

The "connectionString" global variable stores the URL for the SQL database.

Check Database Function

This function connects to a PostgreSQL database, retrieves all user records, checks if the users IDs matches any hashed values in the database, and returns an object indicating whether the target was found and its associated thread_id.

Function Documentation:

The code is preceded by documentation comments. This provides information about the function's purpose, parameters, and return type. The target parameter is expected to be a string, representing the value to search for in the database. The function returns an object with two properties "found" a boolean indicating whether the target was found and "thread_id" a string representing an associated thread ID, or undefined if not found.

Async Function:

The function is defined as async as it contains asynchronous operations that involve waiting for external database calls.

Connecting to Database:

It connects to the PostgreSQL database using the provided connection string then initializes the Drizzle ORM with the connected client.

```
const client = postgres(connectionString);
const db = drizzle(client);
```

Initialization:

It initializes variables to store the thread ID and the found status.

```
var thread_id = undefined;
var found = false;
```

Querying Database:

It queries the database to retrieve all user records.

```
var users = await db.select({ID: user.id, thread_id:
```

```
user.thread_id}).from(user);
```

Iterating Through Records:

It iterates through each user record retrieved from the database. It retrieves the hashed value from each user record and compares it with the target value using bcrypt. If the target is found, it sets the found variable to true and exits the loop.

```
for (var i = 0, lens = users.length; i less than len; i++) {
     const hash = users[i].ID;
     found = bcrypt.compareSync(target, hash);
     if (found == true){
          break;
     }
}
```

Returning Result:

It returns an object containing the found status and associated thread ID: This object is returned containing whether the target was found and the associated thread ID, if any.

```
return {found, thread_id};
```

Update Database Function

This function updates the PostgreSQL database with the provided user ID and thread ID by inserting a new record into the user table using the Drizzle ORM. It ensures proper cleanup by closing the database connection after the update is complete.

Function Description:

The function is preceded by a comment that provides documentation about the function's purpose, parameters, and return type. It describes that the function updates the database with a given user ID and thread ID. It specifies the parameters user_id and user_thread_id, both of type string. It specifies that the function doesn't return any value but indicates the completion of the database update.

Database Update:

It establishes a connection to the PostgreSQL database using the provided connection string. It initializes the Drizzle ORM with the connected client.

```
// Establish a connection to the PostgreSQL database.
const client = postgres(connectionString);
// Initialize the Drizzle ORM with the connected client.
const db = drizzle(client);
```

It uses the db.insert() method to insert a new record into the user table of the database. The inserted record contains the id and thread_id values provided as parameters to the function.

```
// Insert the user ID and thread ID into the database.
await db.insert(user).values({ id: user_id, thread_id: user_thread_id });
```

Closing Database Connection:

After the database update is complete, it closes the database connection to release resources and ensure proper cleanup. It awaits the completion of the client.end() method, which closes the connection.

```
await client.end();
```

Check Cache Function

This function checks the Redis cache for a given query and return the associated cached response if found.

Function Description:

The function is preceded by a comment that provides documentation about the function's purpose, parameters, and return type. It describes that the function checks the cache for a given query and returns the response if found. It specifies the parameter query, which is the query to search for in the cache. It specifies that the function returns an object containing whether the query was found in the cache (found) and the associated response (response), if found.

Cache Checking:

It initializes found flag to false and response to "error" and establishes a connection to the Redis cache server using the provided URL.

```
let found = false; // Initialize found flag to false
let response = "error"; // Initialize response to "error" by default
```

```
// Establish a connection to the Redis cache server.
const client = new Redis(process.env.REDIS_URL);
```

It retrieves all keys from the cache then iterates through each key in the cache and checks for similarity between the query and the key using the similarity function.

```
// Retrieve all keys from the cache.
let keys = await client.keys('*');

// Iterate through each key in the cache.
for(var i = 0, len = keys.length; i < len; i++) {
  var key = keys[i];

  // Check if there is a similarity between the query and the key.
  const check = await similarity(query, key);
```

If a match is found, it sets the found flag to true and retrieves the associated response from the cache. It breaks out of the loop once a match is found to avoid unnecessary iterations.

```
if (check == "yes") {
    found = true; // Set found flag to true
    // Retrieve the response associated with the key from the cache.
    response = await client.get(key);
    break; // Exit loop once a match is found
}
```

Disconnecting from Cache Server and Return Value:

After checking the cache, it disconnects from the Redis cache server to release resources. It returns an object containing whether the query was found in the cache and the associated response.

```
// Disconnect from the Redis cache server.
client.disconnect();

// Return an object containing whether the query was found in the cache and
the associated response.
return {found, response};
```

Update Cache Function

This function is responsible for updating a Redis cache with a given query and its associated response enabling faster retrieval of data and cheaper costs for frequent queries.

Function Description:

The function is preceded by a JSDoc comment that provides documentation about its purpose, parameters, and return type. It describes that the function updates the cache with a given query and response. It specifies the input parameters are query and a response, both of type string. It specifies that the function doesn't return any value but indicates the completion of the cache update.

Cache Update:

The function establishes a connection to the Redis cache server using the provided URL stored in the REDIS_URL environment variable.

```
const client = new Redis(process.env.REDIS_URL);
```

It sets the provided query and response pair in the cache.

```
await client.set(query, response);
```

After the update is complete, it disconnects from the Redis cache server to release resources.

```
await client.disconnect();
```

Error Handling:

The cache update operation is wrapped in a try-catch block to handle any potential errors that may occur during the process. If an error occurs, it logs a message indicating a minor error updating the cache.

```
catch {console.log("MINOR ERROR UPDATING CACHE!")}
```

Reset Cache & Database Functions

These functions allow for the manual reset of user data in the database and clearing of the cache. They are useful for administrative purposes or debugging scenarios but should be used with caution due to the potential erasure of important and irretrievable data. The warning messages serve as reminders to ensure that these operations are performed intentionally.

Reset Users Function:

This function resets user data in the database. It begins by logging a warning message indicating that manual user reset is enabled.

```
console.log("WARNING MANUAL USER RESET IS ENABLED!!!!");
```

Then, it establishes a connection to the PostgreSQL database using the provided connection string and initializes the Drizzle ORM with the connected client.

```
const client = postgres(connectionString);

const db = drizzle(client);
```

Finally, it deletes all user data from the database.

```
await db.delete(user);
```

Reset Cache Function:

This function resets the cache by deleting all keys. Similar to resetUsers, it logs a warning message indicating that manual cache reset is enabled.

```
console.log("WARNING MANUAL CACHE RESET IS ENABLED!!!");
```

It establishes a connection to the Redis cache server using the provided URL stored in the REDIS_URL environment variable then it retrieves all keys from the cache.

```
const client = new Redis(process.env.REDIS_URL);

await client.keys('*', async function (err, keys) {
```

It iterates through each key and deletes it from the cache.

```
for(var i = 0, len = keys.length; i < len; i++) {
    await client.del(keys[i]);
}
```

Finally, it disconnects from the Redis cache server.

```
await client.disconnect();
```

Similarity.js Code Documentation

This section will document and explain the code used in the "similarity.js" file. The file contains the code that provides a simple way to determine the similarity between two strings using embeddings obtained from the OpenAI text embedding model.

Code Sections

Imports & Global Variables

At the top the necessary dependencies are imported for working with the OpenAI API and a library for computing the cosine similarity.

```
import OpenAI from "openai";
import cosineSimilarity from "cosine-similarity";
```

Dependency functionalities:

- openai: for interfacing with the OpenAI API.
- cosine-similarity: for computing cosine similarity.

```
const threshold = 0.81;
```

A threshold value is defined which is used to determine whether the similarity between two strings is considered significant or not. The higher the value, the more queries must resemble each other to be deemed similar. This threshold value is set to 0.81.

Get Embedding Function

The function's purpose is to take a piece of text as input and return its embedding vector. Embedding vectors represent textual information in a continuous numerical space, capturing semantic and contextual similarities between words and phrases.

Function Documentation:

The code is preceded by documentation comments. This provides information about the function's purpose, parameters, and return type. The input parameter is expected to be a string, representing the text that needs to be embedded. The function returns an array of numbers, which represents the embedding vector of the input text.

Initializing OpenAI Client:

The OpenAI client is initialized using the API key retrieved from an environment variable named OPENAI_API_KEY.

```
apiKey: process.env['OPENAI_API_KEY'],
```

API Call:

The function makes an asynchronous call to OpenAI's API to obtain the embedding of the input text. It uses the openai object, which is the OpenAI client. The openai.embeddings.create() method is used to create an embedding for the provided input text. The method requires specifying the model to use and the input text.

```
// Request text embedding from OpenAI API
const embedding = await openai.embeddings.create({
    model: "text-embedding-3-small",
    input: input,
})
```

Extracting And Returning Embedding Vector:

Once the embedding is obtained from the API, the function extracts the embedding vector from the response data. The response contains an array of objects, each representing a portion of the input text. Since only one piece of text is embedded at a time, the embedding vector is extracted from the first object in the array. The function returns the extracted embedding vector.

```
return embedding.data[0].embedding
```

Dependency functionalities:

desc

```
code
```

Get Similarity Function

The function's purpose is to determine the similarity between two embedding vectors using cosine similarity. Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space. It measures the cosine of the angle between the vectors and is often used in natural language processing tasks to compare the similarity of documents or word embeddings.

Function Documentation:

The code is preceded by documentation comments. This provides information about the function's purpose, parameters, and return type. The string1 and string2 parameters are expected to be a string, representing the two embedded vectors of strings. The function returns a single number, which represents the cosine similarity score between the two input vectors.

Cosine Similarity Calculation:

The function calculates the cosine similarity between the two input vectors using the cosineSimilarity function provided by the cosine-similarity library. It passes string1 and string2 as arguments to the cosineSimilarity function. The function computes the cosine similarity score between the two vectors and assigns it to the variable similarity.

```
const similarity = cosineSimilarity(string1, string2);
```

The function returns the calculated cosine similarity score.

```
return similarity;
```

Similarity Function

The function's purpose is to compare the similarity between two strings by first embedding them and then computing the cosine similarity between their embeddings. It returns a string that resolves to either "yes" if the similarity is above a specified threshold or "no" otherwise.

Function Documentation:

The code is preceded by documentation comments. This provides information about the function's purpose, parameters, and return type. The query parameter represents the query string that needs to be compared against the value string. The value parameter represents the value string that needs to be compared against the query string. The function returns a string, either "yes" or "no".

Embedding:

The function calls the get embedding function for both the query and value strings to obtain their embedding vectors and stores the embeddings in variables.

```
const embeddedQuery = await getEmbedding(query)
const embeddedValue = await getEmbedding(value);
```

Cosine Similarity: After obtaining the embeddings, the function computes the cosine similarity between them using the get similarity function. It passes the embedded query and value vectors to the function, which returns the cosine similarity score. The similarity score is stored in a variable named similarity.

```
const similarity = getSimilarity(embeddedQuery, embeddedValue);
```

Threshold Check: The function compares the similarity score against a predefined threshold value. If the similarity score is greater than the threshold, it returns "yes", indicating that the strings are similar. Otherwise, it returns "no", indicating that the strings are not similar.

```
if (similarity > threshold) {
    return("yes");
} else {
    return("no");
}
```

Logging.js Code Documentation

This section will document and explain the code used in the "logging.js" file. This code provides a logging functions to track events within the chatbot system. It logs events to both the console for debugging purposes and to the PostHog analytics platform for more comprehensive analytics and monitoring.

Code Sections

Imports & Global Variables

At the top the necessary dependencies are imported for working with PostgreSQL databases, including libraries for hashing, ORM, database connection, cache interaction, and schema definitions. Additionally, it sets up the connection string for connecting to the PostgreSQL database.

```
import { PostHog } from 'posthog-node';
import { user, logs } from './src/schema.js';
```

Dependency functionalities:

- PostHog: The PostHog client library is for sending analytics data to the PostHog analytics platform.
- user and logs: Import of schemas for the user and logs from the local file schema.js. These schemas define the structure of the tables in the database.

```
const connectionString = process.env.DATABASE_URL;
```

The "connectionString" global variable stores the URL for the SQL database.

PostHog Function

This function logs events to the PostHog analytics platform.

Function:

It initializes a PostHog client using the provided PostHog token and host URL.

```
const clientPH = new PostHog(
    process.env.POSTHOG_TOKEN,
    { host: 'https://eu.posthog.com' }
)
```

It gets the current timestamp in UTC. It determines the user status (new or existing) and the query status (cached or new) based on the information in the logbook object. It extracts user ID and thread ID from the logbook object.

```
var timestamp =  new Date().toUTCString();

var user = "New User";
if (logbook.user_exists) {
    var user = "Existing User";
}

var query = "Cached Query";
if (!logbook.query_exists) {
    var query = "New Query";
}

var user_id = logbook.user_id;
var thread_id = logbook.thread_id;
```

It captures the event in PostHog, providing the distinct ID, event name, and event properties.

```
clientPH.capture({
    distinctId: user_id,
    event: "Chatbot response sent to user",
    properties: {
        webhook_timestamp: timestamp,
        thread_id: thread_id,
        query: query,
        query_message: logbook.query_message,
    },
})
```

It shuts down the PostHog client after capturing the event.

```
clientPH.shutdown();
```

Log Function

This function logs events to both the console and calls the PostHog function.

It calls the posthog function to log the event to PostHog.

```
posthog(logbook);
```

It forms a log message containing date and time, user status, thread ID, query status, and query message.

```
var log_message = "Date & Time: ";
var datetime = new Date().toUTCString();
log_message += datetime;

if (logbook.user_exists == true) {
    log_message += " - Existing User (ID: " + logbook.user_id + ", Thread ID: " + logbook.thread_id + ")";
}
else {
    log_message += " - New User (ID: " + logbook.user_id + ", Thread ID: " + logbook.thread_id + ")";
}

if (logbook.query_exists == true) {
    log_message += " - Asked Cached Query: " + logbook.query_message;
}
else {
    log_message += " - Asked New Query: " + logbook.query_message;
}
```

It logs the message to the console using console.log().

```
console.log(log_message);
```

Export:

The log function is exported so that it can be used in other parts of the application.

```
export { log };
```