

This n8n workflow is designed to automate responses or actions based on incoming Telegram messages, with different handling based on the message type, specifically differentiating voice messages.

Here's a breakdown of the workflow:

1. **Telegram Trigger (Updates: message)**: The workflow initiates when there's an update to a message in Telegram. This could be a new message, an edited message, etc.
2. **Switch Node (mode: Rules)**: This node acts as a conditional junction. It evaluates the incoming data (from the Telegram trigger) against defined rules. In this specific setup:
   * **Output 1 (labeled 'voice')**: If the incoming Telegram message meets a condition defined to identify it as a voice message, the workflow proceeds along the top path.
   * **Default Output (bottom path)**: If the message does not meet the 'voice' condition, or if other rules direct it, the workflow proceeds along the bottom path.
3. **Top Path (Voice Message Handling)**:
   * **HTTP Request1 (POST)**: If the Switch node identifies the message as 'voice', this node sends a POST request to an external URL (partially visible as https://4c2f-176-28-24...). This is likely an API endpoint designed to process audio data, perhaps for transcription, analysis, or another voice-related service.
   * **Telegram1 (sendMessage: message)**: After the HTTP request is completed (successfully or not, depending on error handling not visible here), this node sends a message back via Telegram. This message could be the result of the voice processing (e.g., a transcription), a confirmation, or an error message.
4. **Bottom Path (Other Message/File Handling)**:
   * **Telegram (get: file)**: If the Switch node directs the workflow down this path, this node first attempts to retrieve a file associated with the Telegram message. This suggests this path might handle messages containing files or commands to fetch specific files.
   * **HTTP Request (POST)**: Similar to the top path, this node then sends a POST request to the *same* external URL (https://4c2f-176-28-24...). This implies the external service might be capable of handling different types of data (e.g., file content from this path, voice data from the top path).
   * **Telegram2 (sendMessage: message)**: Finally, after this HTTP request, a message is sent back via Telegram, likely containing the results of the file processing or a relevant notification.

**In summary for a report:**

This n8n workflow automates the processing of incoming Telegram messages. It uses a conditional switch to differentiate message types:

* **For voice messages**: The workflow directly sends the voice data to an external service via an HTTP POST request and then sends a reply back through Telegram.
* **For other message types (potentially involving files)**: The workflow first attempts to retrieve a file from Telegram, then sends data (likely from the file) to the same external service via an HTTP POST request, and finally sends a reply through Telegram.

The workflow centralizes interaction with an external processing service (identified by the common HTTP request URL) for different types of Telegram inputs, providing automated responses or results back to the user on Telegram.

from fastapi import FastAPI, Body, UploadFile, File

from pydantic import BaseModel

import uvicorn

import whisper

from rag import RAG

app = FastAPI()

LLM = RAG()

LLM.load\_from\_csv("jordan\_transactions.csv")

model = whisper.load\_model("base")

@app.post("/test/")

async def predict(x: str = Body(...)):

    answer = LLM.chat(x)

    print(answer)

    return {"answer": answer , "answer\_ID" : 123}

@app.post("/audio/")

async def transcribe\_audio(file: UploadFile = File(...)):

    contents = await file.read()

    with open("temp\_audio\_file.wav", "wb") as f:

        f.write(contents)

    result = model.transcribe("temp\_audio\_file.wav")

    print(result)

    answer = LLM.chat(result["text"])

    return {"massege":answer}

uvicorn.run(app, host="0.0.0.0", port=8000)

This Python script defines a web API using the FastAPI framework. The API has two endpoints: one for processing text input and another for transcribing audio and then processing the transcribed text. It utilizes a custom RAG (likely Retrieval Augmented Generation) model for chat-like interactions and OpenAI's Whisper model for speech-to-text transcription.

Here's a breakdown of the script:

1. **Imports:**
   * FastAPI, Body, UploadFile, File: Core components from FastAPI for creating the API, handling request bodies, and file uploads.
   * pydantic.BaseModel: Used for data validation and settings (though not explicitly used for a request/response model in this snippet, it's a common FastAPI import).
   * uvicorn: An ASGI server used to run the FastAPI application.
   * whisper: OpenAI's library for automatic speech recognition (ASR).
   * rag.RAG: A custom module/class named RAG, presumably implementing a Retrieval Augmented Generation model.
2. **Application Initialization and Model Loading:**
   * app = FastAPI(): Creates an instance of the FastAPI application, which will handle incoming web requests.
   * LLM = RAG(): An instance of the custom RAG class is created and assigned to the variable LLM. This object will likely handle natural language understanding and generation tasks.
   * LLM.load\_from\_csv("jordan\_transactions.csv"): This line suggests that the RAG model is being loaded or fine-tuned with data from a CSV file named "jordan\_transactions.csv". This data might be used as a knowledge base for the LLM to answer questions or engage in conversations related to these transactions.
   * model = whisper.load\_model("base"): The "base" version of the Whisper ASR model is loaded. This model will be used to convert spoken audio into text.
3. **API Endpoints:**
   * **@app.post("/test/")**
     + **Purpose**: This endpoint is designed to receive a text string, process it using the LLM (RAG model), and return an answer.
     + **Request**: It expects a POST request to the /test/ path. The request body should contain a JSON object with a key x whose value is a string. Body(...) indicates that x is a required field from the request body.
     + **Processing**:
       1. answer = LLM.chat(x): The input string x is passed to the chat method of the LLM object. This method likely queries the RAG model with the input text.
       2. print(answer): The answer obtained from the LLM is printed to the server's console (for debugging or logging).
     + **Response**: Returns a JSON object: {"answer": answer, "answer\_ID": 123}. It includes the answer from the LLM and a static answer\_ID.
   * **@app.post("/audio/")**
     + **Purpose**: This endpoint is designed to receive an audio file, transcribe it to text using Whisper, process the transcribed text with the LLM, and return a response.
     + **Request**: It expects a POST request to the /audio/ path. The request should contain an uploaded file (e.g., in a form-data). UploadFile = File(...) indicates a required file upload.
     + **Processing**:
       1. contents = await file.read(): Reads the entire content of the uploaded audio file asynchronously.
       2. with open("temp\_audio\_file.wav", "wb") as f: f.write(contents): The received audio content is written to a temporary file named "temp\_audio\_file.wav" in binary write mode ("wb"). This step is necessary because the Whisper model typically expects a file path for transcription.
       3. result = model.transcribe("temp\_audio\_file.wav"): The Whisper model (model) transcribes the audio from "temp\_audio\_file.wav". The result is usually a dictionary containing the transcribed text among other details.
       4. print(result): The transcription result is printed to the server's console.
       5. answer = LLM.chat(result["text"]): The transcribed text (accessed via result["text"]) is then passed to the chat method of the LLM object, similar to the /test/ endpoint.
     + **Response**: Returns a JSON object: {"massege": answer}. It includes the answer from the LLM (note the typo: "massege" instead of "message").
4. **Running the Application:**
   * uvicorn.run(app, host="0.0.0.0", port=8000): This line starts the FastAPI application using the Uvicorn ASGI server.
     + app: The FastAPI instance to run.
     + host="0.0.0.0": Makes the server accessible on all available network interfaces (i.e., from any IP address that can reach the machine).
     + port=8000: Specifies that the server will listen for incoming requests on port 8000.

In essence, this script sets up a backend service that can interact via text or voice. For text interactions, it directly uses a custom RAG model presumably trained or loaded with transaction data. For voice, it first converts the audio to text using Whisper and then feeds this text to the same RAG model to get a response.

import nltk

from langchain\_community.document\_loaders import PyPDFLoader, WebBaseLoader

from langchain.text\_splitter import NLTKTextSplitter

from langchain.prompts import ChatPromptTemplate, HumanMessagePromptTemplate

from langchain\_core.messages import SystemMessage, HumanMessage

from langchain.schema.runnable import RunnablePassthrough

from langchain\_core.output\_parsers import StrOutputParser

from langchain.vectorstores import Chroma

from langchain\_google\_genai import ChatGoogleGenerativeAI, GoogleGenerativeAIEmbeddings

from langchain.schema import Document

import pandas as pd

class RAG:

    def \_\_init\_\_(self):

        self.\_download\_nltk()

        self.key = 'AIzaSyBkeAaLuUE8mkyDSNdNc6ULcVTqjfcx-ro'

        self.chat\_model = ChatGoogleGenerativeAI(google\_api\_key=self.key, model="gemini-2.0-flash")

        self.embedding\_model = GoogleGenerativeAIEmbeddings(google\_api\_key=self.key, model="models/embedding-001")

        self.pages = None

        self.chunks = None

        self.retriever = None

        self.rag\_chain = None

    def \_download\_nltk(self):

        for pkg in ['punkt', 'averaged\_perceptron\_tagger']:

            try:

                nltk.data.find(f'tokenizers/{pkg}')

            except LookupError:

                nltk.download(pkg)

    def load\_from\_pdf(self, path):

        try:

            loader = PyPDFLoader(path)

            self.pages = loader.load\_and\_split()

            self.\_process\_documents()

        except Exception as e:

            print(f"[ERROR] Failed to read PDF: {e}")

    def load\_from\_website(self, url):

        try:

            loader = WebBaseLoader(url)

            self.pages = loader.load()

            self.\_process\_documents()

        except Exception as e:

            print(f"[ERROR] Failed to read website: {e}")

    def load\_from\_csv(self, path):

        """Load and process data from a CSV file with structured content."""

        try:

            df = pd.read\_csv(path)

            # Create documents where each document represents a row with explicit columns

            self.pages = []

            for \_, row in df.iterrows():

                page\_content = ""

                for col\_name, value in row.items():

                    page\_content += f"{col\_name}: {value}\n"

                self.pages.append(Document(page\_content=page\_content.strip()))

            if not self.pages:

                raise ValueError("No data found in CSV.")

            self.\_process\_documents()

        except Exception as e:

            print(f"[ERROR] Failed to read CSV: {e}")

    def \_process\_documents(self):

        self.text\_splitter = NLTKTextSplitter(chunk\_size=4000, chunk\_overlap=1000)

        self.chunks = self.text\_splitter.split\_documents(self.pages)

        self.db = Chroma.from\_documents(self.chunks, self.embedding\_model, persist\_directory="./chroma\_db\_")

        self.retriever = self.db.as\_retriever(search\_kwargs={"k": 500})

        self.setup\_chat\_template()

    def setup\_chat\_template(self):

        chat\_template = ChatPromptTemplate.from\_messages([

            SystemMessage(content="""You are a helpful assistant. Use the context to answer the user's question.

            Give the answer without introduction. Provide an answer based on your information if context does not provide answers."""),

            HumanMessagePromptTemplate.from\_template("Context:\n{context}\n\nQuestion: {question}")

        ])

        output\_parser = StrOutputParser()

        def format\_docs(docs):

            return "\n\n".join(doc.page\_content for doc in docs)

        self.rag\_chain = (

            {"context": self.retriever | format\_docs, "question": RunnablePassthrough()}

            | chat\_template

            | self.chat\_model

            | output\_parser

        )

    def chat(self, query):

        if not self.rag\_chain:

            return "[ERROR] No data loaded. Please load PDF, website, or CSV content first."

        try:

            return self.rag\_chain.invoke(query)

        except Exception as e:

            print(f"[ERROR] Chat failed: {e}")

            return "Sorry, an error occurred."

This Python script defines a class named RAG (Retrieval Augmented Generation) designed to build and interact with a question-answering system. This system can ingest data from PDF files, websites, or CSV files, and then use that data as context to answer user queries using Google's Generative AI models. The implementation heavily relies on the LangChain library for orchestrating the different components.

Here's a detailed breakdown of the class and its methods:

1. **Imports:**
   * nltk: Natural Language Toolkit, used here for text splitting.
   * langchain\_community.document\_loaders: For loading documents from PDFs (PyPDFLoader) and web pages (WebBaseLoader).
   * langchain.text\_splitter.NLTKTextSplitter: For splitting large documents into smaller chunks using NLTK's sentence tokenization.
   * langchain.prompts, langchain\_core.messages: For creating and managing prompt templates for the language model.
   * langchain.schema.runnable.RunnablePassthrough, langchain\_core.output\_parsers.StrOutputParser: LangChain components for building complex chains and parsing outputs.
   * langchain.vectorstores.Chroma: For creating and using a Chroma vector database to store and retrieve document embeddings.
   * langchain\_google\_genai: For using Google's Generative AI models for chat (ChatGoogleGenerativeAI) and text embeddings (GoogleGenerativeAIEmbeddings).
   * langchain.schema.Document: A LangChain class representing a piece of text and its metadata.
   * pandas as pd: For reading and processing CSV files.
2. **Class RAG:**
   * **\_\_init\_\_(self) (Constructor):**
     + Calls \_download\_nltk() to ensure necessary NLTK resources (punkt tokenizer and averaged\_perceptron\_tagger) are available.
     + **Security Warning:** Initializes self.key with a hardcoded Google API key ('AIzaSyBkeAaLuUE8mkyDSNdNc6ULcVTqjfcx-ro'). **This is a significant security risk, as API keys should be managed securely (e.g., through environment variables) and not embedded directly in code.**
     + Initializes self.chat\_model using ChatGoogleGenerativeAI with the specified API key and the "gemini-2.0-flash" model.
     + Initializes self.embedding\_model using GoogleGenerativeAIEmbeddings with the API key and the "models/embedding-001" model.
     + Initializes instance variables self.pages, self.chunks, self.retriever, and self.rag\_chain to None. These will be populated as data is loaded and processed.
   * **\_download\_nltk(self):**
     + A helper method that checks if the NLTK packages 'punkt' (for sentence tokenization) and 'averaged\_perceptron\_tagger' are downloaded. If not, it downloads them. This is essential for the NLTKTextSplitter.
   * **load\_from\_pdf(self, path):**
     + Takes a file path to a PDF.
     + Uses PyPDFLoader from LangChain to load the PDF content. The load\_and\_split() method likely splits the PDF into individual pages.
     + Stores the loaded pages in self.pages.
     + Calls self.process\_documents() to further process these pages.
     + Includes basic error handling for PDF reading failures.
   * **load\_from\_website(self, url):**
     + Takes a website url.
     + Uses WebBaseLoader to fetch and load the content from the given URL.
     + Stores the loaded content (as documents) in self.pages.
     + Calls self.process\_documents().
     + Includes basic error handling for website loading failures.
   * **load\_from\_csv(self, path):**
     + Takes a file path to a CSV file.
     + Uses pandas to read the CSV into a DataFrame.
     + It then iterates through each row of the DataFrame and constructs a LangChain Document object. The page\_content of each document is a string where each column name and its corresponding value from the row are combined (e.g., "Column1: Value1\nColumn2: Value2").
     + These created documents are stored in self.pages.
     + Calls self.process\_documents().
     + Includes error handling, including a check if the CSV was empty.
   * **\_process\_documents(self):**
     + This is a crucial method where the loaded documents are prepared for retrieval.
     + Initializes NLTKTextSplitter with a chunk\_size of 4000 characters and chunk\_overlap of 1000 characters. This breaks down the documents in self.pages into smaller, manageable self.chunks.
     + Creates a Chroma vector store (self.db) from these self.chunks using the self.embedding\_model. The vector store is persisted to a local directory named "./chroma\_db\_". This process involves generating embeddings (numerical representations) for each chunk and storing them for efficient similarity searches.
     + Initializes self.retriever from the self.db. The retriever is configured to fetch the top k=500 most relevant document chunks when queried.
     + Calls self.setup\_chat\_template() to construct the actual RAG chain.
   * **setup\_chat\_template(self):**
     + Defines the structure and logic of the RAG chain using LangChain Expression Language (LCEL).
     + chat\_template: A ChatPromptTemplate is created with:
       - A SystemMessage: Instructs the AI model to act as a helpful assistant, use the provided context to answer, avoid introductions, and use its general knowledge if the context is insufficient.
       - A HumanMessagePromptTemplate: Formats the user's question along with the retrieved context.
     + output\_parser: An StrOutputParser is used to ensure the final output from the language model is a string.
     + format\_docs(docs): An inner helper function to combine the page\_content of multiple retrieved documents into a single string, separated by double newlines.
     + self.rag\_chain: The RAG chain is defined:
       - The input to the chain is a dictionary containing the context and question.
       - context is obtained by piping the self.retriever (which fetches relevant documents based on the input question) to the format\_docs function.
       - question is passed through directly from the input query using RunnablePassthrough().
       - This dictionary is then passed to the chat\_template.
       - The formatted prompt from chat\_template is sent to self.chat\_model (Gemini).
       - The output from the chat model is parsed by output\_parser.
   * **chat(self, query):**
     + This is the public method used to interact with the RAG system.
     + It takes a user's query (question) as input.
     + Checks if self.rag\_chain has been initialized (i.e., if data has been loaded and processed). If not, it returns an error message.
     + Invokes the self.rag\_chain with the query. This triggers the entire RAG process: retrieval, context injection, prompt formatting, and generation by the LLM.
     + Returns the answer from the chain.
     + Includes error handling for issues during the chat invocation.

In summary, this RAG class provides a comprehensive solution for building a question-answering system that can leverage external documents (PDFs, websites, CSVs) as a knowledge base. It handles data loading, preprocessing (chunking, embedding), storage in a vector database, context retrieval, and answer generation using Google's Gemini models, all orchestrated through the LangChain framework. The most critical point to address if using this code would be the secure management of the API key.