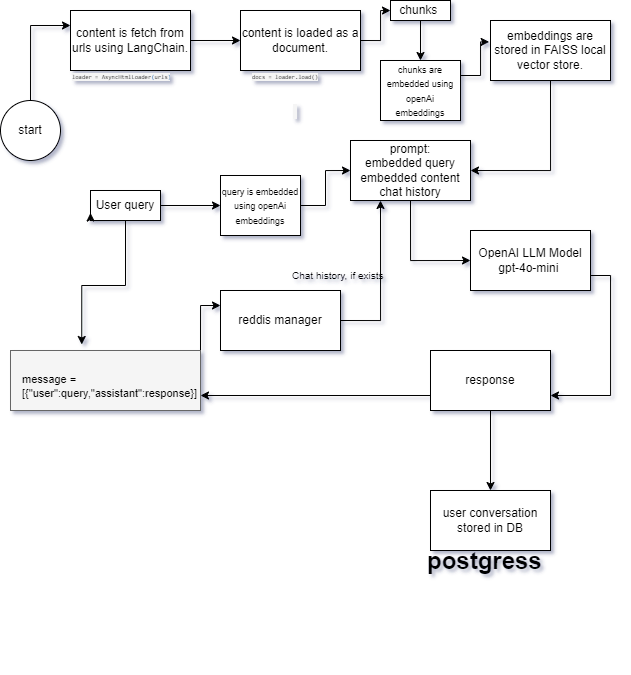
**WhatsApp Bot Using Custom ­­Retrieval-Augmented Generation.**



**1. Start**

* The process begins with initiating the system.

**2. Content Fetching**

* Using sitemap Urls are collected and dumped in a python list.
* Using LangChains **AsynChromiumLoader** content from urls are loaded as HTML into docs.
  + from langchain\_community.document\_loaders import AsyncHtmlLoader  
      
    urls = urls  
    loader = AsyncHtmlLoader(urls)  
    docs = loader.load()
* Using LangChains **Html2TextTransformer** the content is converted into text document.
  + from langchain\_community.document\_transformers import Html2TextTransformer  
      
    html2text = Html2TextTransformer()  
    docs\_transformed = html2text.transform\_documents(docs)

**3. Content Chunking**

* The content is broken down into chunks. This step helps in efficient processing and embedding creation.
* text\_splitter = RecursiveCharacterTextSplitter.from\_tiktoken\_encoder(
* chunk\_size=550, chunk\_overlap=50
* )
* chunked\_documents = text\_splitter.split\_documents(docs\_transformed)

**4. Embedding Generation**

* The chunks are embedded using the OpenAI embedding model. The embeddings represent the content in a vectorized form that can be compared against queries.
* The user query is also embedded using the same model.
* embeddings = OpenAIEmbeddings(model="text-embedding-ada-002")

**5. Storing Embeddings**

* These embeddings are stored in a local FAISS vector store. This storage mechanism allows quick similarity searches when a query is made.
* faiss\_vectorstore = FAISS.from\_documents(
* documents=chunked\_documents,
* embedding=embeddings,
* )
* faiss\_vectorstore.save\_local("fais\_store")

---LOADING FAISS STORE--------------

vector\_store = FAISS.load\_local(

    "faiss\_store", embeddings, allow\_dangerous\_deserialization=True

)

**WHY FAISS?**

* **Performance:** FAISS is highly optimized for fast similarity search on large datasets, making it suitable for real-time applications
* **Open Source:** FAISS is open-source and actively maintained, with extensive documentation and community support.
* **Search optimization: FAISS support exact search algorithm which makes it better then approximate search.**
* **Other DBs:** I tried working with **PINECONE** but it has a limit of upserting data , then tried **ChromaDB** still got errors and the similarity search was not as good as FAISS, so in the end decided to use FAISS.

**6. User Query**

* When the user provides a query, it undergoes embedding using the same OpenAI embedding model. The query is now ready for comparison with stored content using FAISS.similaritysearch().
* def get\_docs(query,top\_k=4):
* results = vector\_store.similarity\_search(
* query=query,
* k=top\_k,
* )
* formated\_docs = ""
* for result in results:
* formated\_docs += result.page\_content
* return formated\_docs

**7. History and Embedding Combination**

* A prompt is generated that includes the query, content, and chat history if exist.

**8. Redis Manager**

* We are using Redis for chat history retrieval purpose and for managing processed messages.
* **Adding chat to history / Getting Chat history**
* def add\_chat\_history(self, chat\_id: str, messages: List[Dict[str, Any]]) -> None:
* stored\_chat\_history = self.r.hget(chat\_id, "chat\_history")
* if stored\_chat\_history:
* chat\_history = json.loads(stored\_chat\_history)
* else:
* chat\_history = {"messages": []}
* chat\_history["messages"].extend(messages)
* self.r.hset(chat\_id, "chat\_history", json.dumps(chat\_history)
* def get\_history(self, chat\_id: str,limit=20):
* """Retrieve the chat history for a specific chat ID."""
* stored\_chat\_history = self.r.hget(chat\_id, "chat\_history")
* if stored\_chat\_history:
* history = json.loads(stored\_chat\_history)
* return history['messages'][:limit]
* else:
* return []
* **Adding and Checking processed message:**
* def is\_message\_processed(self, chat\_id: str, message\_id: str) -> bool:
* """Check if a message has already been processed."""
* processed\_set = self.r.smembers(f"{chat\_id}\_processed\_messages")
* return message\_id.encode('utf-8') in processed\_set
* def add\_processed\_message(self, chat\_id: str, message\_id: str) -> None:
* """Add a processed message ID to the set in Redis."""
* self.r.sadd(f"{chat\_id}\_processed\_messages", message\_id)
* Redis Manager then is invoked, where the query and responses are managed, retrieved, or stored in the Redis database, also it saves the message against unique\_id of each user that is the user number.
* message = [{"user":query,"assistant":response}]
* redis\_manager.add\_chat\_history(chat\_id=chat\_id,messages=message)

response\_message = run(chat\_id=from\_number,query=message\_body)

-----invoking get\_history using chat\_id that is user number--------------------

    history = redis\_manager.get\_history(chat\_id=chat\_id)

    history\_formated = ""

    if history:

        for hist in history:

            history\_formated += f"USER: {hist['user']}\nAIassistant:{hist['assistant']}\n"

    docs = get\_docs(query)

* **ALSO CREATED APIs of these new Functions for backend.**
* if redis\_manager.is\_message\_processed(from\_number, message\_id):
* continue
* print(f"Received message from {from\_number}: {message\_body}")
* redis\_manager.add\_processed\_message(from\_number, message\_id)
* # print("TRYING TO SAVE TO DATABASE")
* self.save\_to\_database(sender\_name, from\_number, message\_body, response\_message)
* print("SAVED TO DATABASE")
* **Why Redis?** **In-memory storage:** Redis operates primarily in memory, which means it provides extremely fast read and write operations. This speed is crucial in real-time applications like chat systems, where quick access to history and previous messages is essential.
* **Low latency:** Redis can handle a high number of operations per second with very low latency, ensuring that chat responses are delivered promptly to users.
* **Horizontal scaling:** Redis supports horizontal scaling through clustering, allowing it to handle increasing loads as your chat application grows in user base and data volume. This is particularly useful for systems that need to handle thousands or even millions of users simultaneously.
* **Optional persistence:** While Redis is an in-memory database, it provides options for data persistence, allowing you to save the chat history to disk. This feature ensures that your chat history is not lost in case of a server failure or restart.
* **Broad language support:** Redis has client libraries available for nearly every programming language, making it easy to integrate with your existing application stack.
* **Automatic cleanup:** Redis supports setting an expiration time on keys, which is useful for chat history management where you might want to automatically delete old messages after a certain period.

**9. Model Invocation**

* The prepared prompt is passed to the OpenAI LLM model (gpt-4o-mini), which processes the input to generate an appropriate response.
* def run(chat\_id,query):
* prompt = PromptTemplate(
* input\_variables=["chat\_history","question","context"], template=BRB\_PROMPT
* )
* llm = ChatOpenAI(model="gpt-4o-mini")
* chain = prompt | llm | StrOutputParser()
* data = {
* "chat\_history":history\_formated,
* "question":query,
* "context":docs
* }
* response =  chain.invoke(data)
* return response

**10. Response Generation**

* The LLM (gpt-4o-mini) model will generate a response/reply based on prompts provided to it.

**11. Response Storage**

* The generated response, along with the user query, is stored in the database. This step ensures that the conversation history is maintained for future reference.

**WHATSAPP CONNECTION USING APIs FROM META:**

load\_dotenv()

WHATSAPP\_ACCESS\_TOKEN = os.getenv("WHATSAPP\_ACCESS\_TOKEN")

WHATSAPP\_API\_URL = os.getenv("WHATSAPP\_API\_URL")

def send\_whatsapp\_message(to\_number: str, text: str):

    url = WHATSAPP\_API\_URL

    headers = {

        'Authorization': f'Bearer {WHATSAPP\_ACCESS\_TOKEN}',

        'Content-Type': 'application/json',

    }

    data = {

        "messaging\_product": "whatsapp",

        "to": to\_number,

        "type": "text",

        "text": {

            "body": text

        }

    }

    response = requests.post(url, headers=headers, json=data)

    print(f"Sent message response: {response.json()}")

**Prerequisites**

* Python
* OpenAI API key.
* LangChain
* WhatsApp Business API access
* Redis
* FAISS