Chatbot implementation in application

1. **LLM model integration -** first you have to find a LLM model and make a client to implement it in application like this

- You have to make a client

```
import os
from groq import Groq

# Initialize Groq client
client =
Groq(api_key="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")
```

2. **Then you have to pass prompt message to client**

```
SYSTEM_PROMPT = ("You are a real estate assistant. Answer questions
about property buying and selling."

                 "You are a concise assistant. "
                 "Always answer in MAX 150 characters. "
                 "Use short bullet points or highlighted key points only."
                 "If someone ask property near indore suggest him/her 1.
khandwa road properties are better and value for money 2. Super corridor
road")
```

```
3. chat_completion = client.chat.completions.create(
4.                 messages=[{"role": "system", "content":
   SYSTEM_PROMPT},  # System prompt
5.                     {"role": "user", "content": user_message}],
6.                 model="llama-3.1-8b-instant",
7.             )
```

3. **After this you have to pass the actual message to the model and it will generate it's reply based upon the prompt and the message**

```
bot_reply = chat_completion.choices[0].message.content
```

4. Take a look of full code of that bot reply

```
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
from groq import Groq

# Initialize Groq client
client =
Groq(api_key="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")


# Define the system prompt
SYSTEM_PROMPT = ("You are a real estate assistant. Answer questions about
property buying and selling."
                "You are a concise assistant. "
                "Always answer in MAX 150 characters. "
                "Use short bullet points or highlighted key points only."
                "If someone ask property near indore suggest him/her 1.
khandwa road properties are better and value for money 2. Super corridor
road")


@csrf_exempt
def chatbot_response(request):
   if request.method == "POST":
       data = json.loads(request.body)
       user_message = data.get("message", "")

       try:
           chat_completion = client.chat.completions.create(
```

```
                messages=[{"role": "system", "content": SYSTEM_PROMPT},  # System prompt
                          {"role": "user", "content": user_message}],
                model="llama-3.1-8b-instant",
            )

            bot_reply = chat_completion.choices[0].message.content
            # breakpoint()
            return JsonResponse({"response": bot_reply})

        except Exception as e:
            return JsonResponse({"error": str(e)}, status=500)

    return JsonResponse({"error": "Invalid request"}, status=400)
```

<u>To implement RAG and Vector DB</u>

To implement these features in your Django web application, here's how you can proceed:

1. Implementing RAG (Retrieval-Augmented Generation) with External Data

RAG helps your chatbot retrieve relevant information from external data sources before generating a response.

Steps to Implement:

- **Data Collection: Gather external data (documents, PDFs, databases, etc.).**
- **Vector Database: Use a vector database like FAISS, Weaviate, ChromaDB, or Pinecone to store and retrieve embeddings.**
- **Text Embeddings: Use an embedding model (like OpenAI's `text-embedding-ada-002` or Groq-supported models) to convert documents into vectors.**
- **Retrieval Mechanism: When a user asks a question, retrieve relevant documents from the vector database.**
- **LLM Processing: Combine retrieved context with user input and pass it to your LLM (Groq in your case).**

- Integration with Django: Modify your chatbot view to fetch relevant data before calling the LLM.

Tools Required:

- `sentence-transformers` or OpenAI embeddings for text embedding
- FAISS or ChromaDB for vector search
- Django integration to fetch relevant data before sending to the LLM

---

2. Implementing LLM Tools

LLM tools allow the model to call external APIs or interact with structured data (e.g., databases, web scraping, API calls).

Steps to Implement:

- Choose Tools: Define tools such as web search, database query, weather API, or financial data API.
- Define Custom Functions: Implement custom functions in Django to fetch data (e.g., calling an API).
- Connect to LLM: Modify your chatbot backend to call these tools when needed.
- Use Agents: Frameworks like LangChain allow defining tools and agents that decide when to use them.

Examples of LLM Tools:

- Database Querying: Fetch property data dynamically from your Django database.
- Web Scraping: Retrieve real-time market trends for property prices.
- API Calls: Fetch real-time weather or finance data.
- PDF/Text Search: Upload documents and let the chatbot extract answers.

Tools Required:

- `LangChain` or `LlamaIndex` for tool usage
- APIs for fetching real-time data
- Custom Django functions for processing information

Next Steps

Would you like to start with RAG first, or do you want to explore both simultaneously? Also, do you have a specific external data source in mind for RAG?

===

To implement RAG (Retrieval-Augmented Generation) in your Django application, follow these steps:

1. Understand RAG in LLMs
RAG enhances LLM responses by retrieving relevant data from an external source (e.g., a database, documents, or APIs) before generating an answer. This helps provide more accurate and up-to-date responses.

---

2. Choose a Retrieval Source
Decide where your chatbot should retrieve information from. Common sources include:
- **Local Files** (e.g., PDFs, CSVs, text files)
- **Databases** (e.g., PostgreSQL, SQLite)
- **Vector Databases** (e.g., FAISS, ChromaDB, Weaviate)
- **Web Data** (e.g., scraping or APIs)

For now, let’s start with a basic **document-based RAG** using a vector database.

---

3. Install Necessary Libraries
You'll need libraries for:
- **Text Embedding & Retrieval:** `sentence-transformers` (for generating embeddings)
- **Vector Storage:** `chromadb` (for storing embeddings)

- **LLM API:** `groq` (for generating responses)

Run:
```bash
pip install chromadb sentence-transformers groq
```

---

4. Create a Vector Database for Retrieval
Modify your Django project by adding a retrieval system.

1. **Create a new Django app** for RAG:
   ```bash
   python manage.py startapp rag
   ```
   Add `"rag"` to `INSTALLED_APPS` in `settings.py`.

2. **Setup ChromaDB**:
   - Inside the `rag` app, create a `vector_store.py` file:
   ```python
   import chromadb
   from sentence_transformers import SentenceTransformer

   # Load sentence transformer model for embeddings
   model = SentenceTransformer("all-MiniLM-L6-v2")

   # Initialize ChromaDB client
   client = chromadb.PersistentClient(path="./chroma_db")
   collection = client.get_or_create_collection(name="knowledge_base")

   def add_document(text, doc_id):
       """Add a document to the vector database"""
       embedding = model.encode(text).tolist()
       collection.add(ids=[doc_id], embeddings=[embedding], documents=[text])

   def retrieve_documents(query, top_k=3):
       """Retrieve top-k relevant documents"""
       query_embedding = model.encode(query).tolist()
       results = collection.query(query_embeddings=[query_embedding], n_results=top_k)
   ```

        return results["documents"][0] if results else []
   ```

3. **Add sample documents**:
   ```python
   add_document("Django is a high-level Python web framework.", "doc1")
   add_document("RAG enhances chatbot responses using retrieval and generation.", "doc2")
   ```

---

5. Update Chatbot to Use RAG
Modify `views.py` in your `chatapp` to use the retrieval system:

```python
import json
from django.http import JsonResponse
from django.views.decorators.csrf import csrf_exempt
from groq import Groq
from rag.vector_store import retrieve_documents

client = Groq(api_key="your_groq_api_key")

@csrf_exempt
def chatbot_response(request):
    if request.method == "POST":
        data = json.loads(request.body)
        user_message = data.get("message", "")

        # Retrieve relevant documents
        retrieved_docs = retrieve_documents(user_message)
        context = " ".join(retrieved_docs) if retrieved_docs else ""

        prompt = f"Context: {context}\nUser: {user_message}\nAnswer in 150 characters with key points."

        try:
            chat_completion = client.chat.completions.create(
                messages=[{"role": "user", "content": prompt}],

            model="llama-3.3-70b-versatile",
        )
        bot_reply = chat_completion.choices[0].message.content
        return JsonResponse({"response": bot_reply})

    except Exception as e:
        return JsonResponse({"error": str(e)}, status=500)

  return JsonResponse({"error": "Invalid request"}, status=400)
```

---

6. Test Your RAG Chatbot
Run the Django server:
```bash
python manage.py runserver
```
Ask questions related to the stored documents and verify if the chatbot retrieves relevant information before answering.

Next Steps
- **Expand Data Sources**: Store FAQs, company info, or domain knowledge.
- **Improve Search**: Use better embeddings or re-rank results.
- **Optimize Storage**: Move from ChromaDB to a more scalable database like Pinecone or Weaviate.

Would you like to integrate more external data sources like PDFs or databases? 🚀
```