# Code Explanation

## Import Statements

import streamlit as st  
import os  
import time  
from groq import Groq  
from langchain.embeddings import HuggingFaceEmbeddings  
from langchain.vectorstores import Chroma  
from langchain.text\_splitter import TokenTextSplitter  
from langchain.document\_loaders import PyPDFLoader  
from langchain.prompts import PromptTemplate  
from langchain.memory import ConversationBufferMemory

These imports bring in necessary libraries: - streamlit for the web interface - os for environment variable handling - time for adding delays in response display - groq for interacting with the Groq API - Various langchain components for NLP tasks

## API Key Setup

os.environ["GROQ\_API\_KEY"] = "gsk\_uod8Lw1go81ZeGWAnZz1WGdyb3FYrECUBnzBSwT7tA7iDZzkvrzb"

This sets the Groq API key as an environment variable. In production, it’s better to use a more secure method like loading from a .env file.

## Session State Initialization

def initialize\_session\_state():  
 # ... (content of the function)

This function initializes various components in the Streamlit session state: - template: A prompt template for the chatbot - prompt: A PromptTemplate object using the template - memory: A ConversationBufferMemory for maintaining chat history - vectorstore: A Chroma vector store for document embeddings - chat\_history: A list to store the conversation

## Groq API Interaction

def get\_groq\_response(client, user\_query, context):  
 # ... (content of the function)

This function sends a request to the Groq API: - It constructs a system prompt with the given context - It sends the user query along with the system prompt - It uses the “mixtral-8x7b-32768” model - It returns the generated response

## Main Function

def main():  
 # ... (content of the function)

This is the main function that runs the Streamlit app:

1. It initializes the Groq client and session state.
2. It sets up a file uploader for PDF documents.
3. When a file is uploaded:
   * It saves the file if it doesn’t exist.
   * It processes the PDF, splits it into chunks, and creates a vector store.
4. It displays the chat history.
5. It handles user input:
   * Retrieves relevant context from the vector store.
   * Gets a response from the Groq API.
   * Displays the response with a typing effect.
   * Updates the chat history.

## Key Components Explained

1. **PDF Processing**:

* loader = PyPDFLoader(fp)  
  data = loader.load()
* This loads the PDF and extracts its content.

1. **Text Splitting**:

* text\_splitter = TokenTextSplitter(chunk\_size=1500, chunk\_overlap=200)  
  all\_splits = text\_splitter.split\_documents(data)
* This splits the document into manageable chunks for processing.

1. **Vector Store Creation**:

* st.session\_state.vectorstore = Chroma.from\_documents(  
   documents=all\_splits,  
   embedding=HuggingFaceEmbeddings(model\_name="sentence-transformers/all-MiniLM-L6-v2")  
  )
* This creates a Chroma vector store from the document chunks, using HuggingFace embeddings.

1. **Context Retrieval**:

* context\_documents = st.session\_state.retriever.get\_relevant\_documents(user\_input)  
  context = " ".join([doc.page\_content for doc in context\_documents])
* This retrieves relevant document chunks based on the user’s input.

1. **Response Generation and Display**:

* response = get\_groq\_response(client, user\_input, context)  
  # ... (code for displaying response with typing effect)
* This gets a response from the Groq API and displays it with a typing effect.

## Error Handling

The code includes try-except blocks to handle potential errors during document processing and API interactions, displaying error messages to the user when issues occur.

## Streamlit Interface

The code uses Streamlit’s chat interface components (st.chat\_input(), st.chat\_message()) to create an interactive chat experience.