Here's a simplified end-to-end explanation of the project:

Overview

This project is a **PDF Question Answering System** that lets users upload PDF files, extract text from them, and ask natural language questions about their content. The system processes the PDFs, identifies the most relevant text, and generates a response using an AI model like Claude.

Key Features

1. **PDF Upload**: Users can upload multiple PDF files through a user-friendly Streamlit interface.

2. **Text Extraction**: The system extracts text content from the uploaded PDFs.

3. **Embedding Creation**: Text is split into smaller chunks and converted into numerical representations (embeddings) using a Hugging Face model.

4. **Vector Store**: These embeddings are stored in a vector database (FAISS) for efficient similarity search.

5. **Question Answering**: Users input a question, and the system retrieves the most relevant text chunks, formats a prompt, and sends it to the Claude AI model for generating answers.

6. **Interactive UI**: Streamlit provides an intuitive interface for all these functionalities.

Step-by-Step Flow

1. **Initialization**

- **Environment Setup**: Loads API keys and other configurations from environment variables.

- **Dependencies**:

  - `PyPDF2`: Extracts text from PDF files.

  - `langchain`: Manages text splitting and embeddings.

  - `Hugging Face Embeddings`: Converts text into embeddings for semantic search.

  - `FAISS`: Handles vector-based similarity search.

  - `Streamlit`: Provides a web-based user interface.

  - `Anthropic`: Communicates with the Claude AI model.

2. **User Interaction**

- The user uploads one or more PDF files through the Streamlit interface.

- Files are temporarily stored in a local directory for processing.

3. **Text Processing**

- **Text Extraction**: Text is extracted from all uploaded PDFs using `PyPDF2`.

- **Splitting**: Long text is split into smaller chunks (e.g., 1000 characters with overlaps) for better search accuracy.

- **Embedding Generation**: Each text chunk is transformed into a numerical format using a Hugging Face embedding model.

- **Vector Store Creation**: The embeddings are saved in a FAISS vector store for efficient retrieval.

4. **Question Answering**

- **Query Input**: The user types a question in the Streamlit input box.

- **Similarity Search**: The system searches the FAISS vector store for the most relevant text chunks.

- **Prompt Creation**: The relevant chunks are combined into a prompt formatted for the Claude AI model.

- **Response Generation**: The prompt is sent to the Claude API, which generates an answer based on the context provided.

5. **Output**

- The AI-generated answer is displayed in the Streamlit app.

- If the system cannot find enough context to answer, it informs the user accordingly.

Technology Stack

1. **Backend**:

   - `Python`

   - `Claude AI API` (Anthropic)

   - `FAISS` for vector search

   - `Hugging Face` for embeddings

2. **Frontend**:

   - `Streamlit` for the web interface

3. **PDF Handling**:

   - `PyPDF2` for text extraction

How to Use

1. **Upload PDFs**: Use the sidebar to upload PDF files.

2. **Process PDFs**: Click the "Process PDFs" button to extract and prepare the data.

3. **Ask Questions**: Enter a question in the text box. The system will fetch the most relevant chunks and provide an answer.

Business Value

- **Efficiency**: Quickly find answers from lengthy documents without manually reading them.

- **Scalability**: Handles multiple PDFs and large datasets.

- **Interactivity**: Easy-to-use web interface accessible to non-technical users.

Example Use Case

1. **Upload**: Add PDFs containing company reports or policy documents.

2. **Process**: Extract and organize the document contents.

3. **Query**: Ask questions like:

  - "What is the main conclusion in these reports?"

  - "Summarize the risk factors discussed in the documents."

Let me know if you'd like clarification on any part or further enhancements!

****
****

Sure! Let's go through the code step by step, explaining each line in simple terms:

1. Importing Libraries:

```python

import os

import faiss

import pickle

from anthropic import Anthropic

from PyPDF2 import PdfReader

from langchain.text_splitter import RecursiveCharacterTextSplitter

from langchain_community.embeddings import HuggingFaceEmbeddings

from langchain_community.vectorstores import FAISS

from dotenv import load_dotenv

import streamlit as st

import textwrap
```

- **os**: Used for interacting with the operating system, like reading files or creating directories.

- **faiss**: A library to efficiently search through large datasets, like finding similar text chunks.

- **pickle**: Used to save and load Python objects (like our FAISS index) between runs of the program.

- **Anthropic**: Used to interact with the Claude AI model for generating answers.

- **PdfReader**: A tool to read PDF files and extract text.

- **RecursiveCharacterTextSplitter**: Helps to split large chunks of text into smaller parts, making it easier to process.

- **HuggingFaceEmbeddings**: A tool for converting text into numerical embeddings (representations) that machine learning models can use.

- **FAISS**: A tool that allows you to store and search for similar text chunks.

- **load_dotenv**: Loads environment variables (like API keys) from a `.env` file.

- **streamlit**: A framework used to create interactive web apps.

- **textwrap**: Used to format the text output, making it easier to read.

2. Loading Environment Variables:

```python

load_dotenv()

```

- This loads environment variables (like API keys) from a `.env` file into the program. This is important for keeping sensitive information (like API keys) secure.

3. The PDFQuestionAnswerer Class:

```python

class PDFQuestionAnswerer:

    def __init__(self, claude_api_key):

```

- **PDFQuestionAnswerer**: A class that handles processing PDFs and answering questions based on their contents.

- **`__init__`**: The constructor method initializes the class, setting up necessary tools (like the Claude API for answering questions and the Hugging Face embeddings for text processing).

4. Extract Text from PDF:

```python
def extract_text_from_pdf(self, pdf_path):
    try:
        reader = PdfReader(pdf_path)
        text = ""
        for page in reader.pages:
            text += page.extract_text() + "\n"
        return text
```

- **`extract_text_from_pdf`**: This method reads a PDF file, extracts text from each page, and combines it into a single string of text.

- **`PdfReader(pdf_path)`**: Reads the PDF file from the given path.

- **`page.extract_text()`**: Extracts the text from each page of the PDF.

5. Process Multiple PDFs and Create Vector Store:

```python
def process_pdfs(self, pdf_directory):
    all_texts = []
    pdf_files = [f for f in os.listdir(pdf_directory) if f.endswith('.pdf')]


    if not pdf_files:
        raise FileNotFoundError(f"No PDF files found in directory: {pdf_directory}")
```

- **`process_pdfs`**: This method processes multiple PDF files from a directory and prepares them for answering questions.

- **`os.listdir(pdf_directory)`**: Lists all files in the specified directory.

- **`f.endswith('.pdf')`**: Filters only PDF files.

```python
    for pdf_file in pdf_files:
        pdf_path = os.path.join(pdf_directory, pdf_file)
        text = self.extract_text_from_pdf(pdf_path)
        if text:
            all_texts.append(text)
```

- This loop reads each PDF file, extracts its text, and adds it to a list of all extracted texts.

```python
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000,
        chunk_overlap=200,
        length_function=len
    )

    chunks = text_splitter.create_documents(all_texts)
```

- **`RecursiveCharacterTextSplitter`**: Splits the long text into smaller chunks, each with a maximum size of 1000 characters, and allows some overlap between chunks (200 characters).

- **`create_documents(all_texts)`**: Creates smaller text documents from the list of all extracted texts.

```python
    self.vector_store = FAISS.from_documents(
        chunks,
        self.embeddings
    )
```

- **`FAISS.from_documents`**: Creates a FAISS vector store using the smaller text chunks, and the text embeddings (numerical representations) generated by the Hugging Face model.

- **`self.embeddings`**: Converts the text into embeddings so that FAISS can store and search them efficiently.

```python
    with open('faiss_vector_store.pkl', 'wb') as f:
        pickle.dump(self.vector_store, f)
```

- This saves the vector store to a file (`faiss_vector_store.pkl`) using **pickle**, so it can be used later without having to process the PDFs again.

6. Load the Saved Vector Store:

```python
def load_vector_store(self):
    try:
        with open('faiss_vector_store.pkl', 'rb') as f:
            self.vector_store = pickle.load(f)
```

- **`load_vector_store`**: This method loads the vector store from the saved file.

- **`pickle.load(f)`**: Loads the vector store from the pickle file so it can be used again.

7. Get Relevant Chunks for a Query:

```python
def get_relevant_chunks(self, query, k=3):
    if not self.vector_store:
        raise ValueError("No PDFs have been processed yet. Call process_pdfs first.")
    docs = self.vector_store.similarity_search(query, k=k)
    return [doc.page_content for doc in docs]
```

- **`get_relevant_chunks`**: This method finds the most relevant chunks of text for a given query.

- **`self.vector_store.similarity_search(query, k=k)`**: Searches the vector store for the top `k` most relevant chunks for the given query.

8. Format the Prompt for Claude (AI):

```python
def format_prompt(self, query, relevant_chunks):
    context = "\n\n".join(relevant_chunks)
    prompt = f"""\n\nHuman: Here are some relevant passages from the documents:
{context}
Based on the passages above, please answer this question: {query}
If the answer cannot be fully determined from the provided passages, please say so. Include specific references to the source material where possible.
Assistant:"""
    return prompt
```

- **`format_prompt`**: This method formats the query and the relevant text chunks into a prompt that will be sent to Claude (AI).

- It organizes the chunks and the question into a friendly format that Claude can understand.

9. Ask a Question Using Claude AI:

```python
def ask_question(self, query):
    if self.vector_store is None:
        print("Error: No vector store available. Please process PDFs first.")
        return "Error: No PDFs processed yet."
    relevant_chunks = self.get_relevant_chunks(query)
    prompt = self.format_prompt(query, relevant_chunks)
    response = self.anthropic.messages.create(
        model="claude-3-sonnet-20240229",
        max_tokens=1000,
        temperature=0,
        messages=[{
            "role": "user",
            "content": prompt
        }]
    )

    return response.content[0].text
```

- **`ask_question`**: This method asks Claude (AI) to answer a question based on the relevant chunks of text.

- It first checks if the vector store is available, then formats the prompt and sends it to Claude using the **Anthropic API**.

- **`response.content[0].text`**: Extracts the answer from the Claude API's response.

10. Streamlit User Interface (UI):

```python
def main():
    try:
        claude_api_key = os.getenv("CLAUDE_API_KEY")
        pdf_dir = os.getenv("PDF_DIR", "./pdfs")
```

- **`main`**: The main entry point for the program. It initializes and runs the Streamlit app.

- It loads the API key (`CLAUDE_API_KEY`) and the PDF directory (`PDF_DIR`).

```python
```

```
qa_system = PDFQuestionAnswerer(claude_api_key)
qa_system.load_vector_store()
```

- Initializes the `PDFQuestionAnswerer` object and loads the existing vector store if available.

```python
uploaded_files = st.sidebar.file_uploader("Upload PDF files", type="pdf", accept_multiple_files=True)
```

- Creates a file uploader in the Streamlit sidebar that allows users to upload multiple PDF files.

```python
if uploaded_files:
    pdf_directory = "./temp_pdfs"
    os.makedirs(pdf_directory, exist_ok=True)
```

- If PDF files are uploaded, it creates a temporary directory to store them.

```python
for uploaded_file in uploaded_files:
    file_path = os.path.join(pdf_directory, uploaded_file.name)
    with open(file_path, "wb") as f:
        f.write(uploaded_file.read())
```

- Saves the uploaded files to the `temp_pdfs` directory.

```python
if st.sidebar.button("Process PDFs") and qa_system.vector_store is None:
    qa_system.process_pdfs(pdf_directory)
    st.sidebar.success("PDFs processed successfully.")
```

    qa_system.load_vector_store()

- When the "Process PDFs" button is clicked, it processes the PDFs and saves the vector store.

```python
query = st.text_input("Ask a question about the PDFs:")
if query:
    answer = qa_system.ask_question(query)
    st.write("### Answer:")
    st.write(textwrap.fill(answer, width=80))
```

- Takes a user query and asks Claude (AI) to answer based on the processed PDFs.

Summary:

This code:

- Allows users to upload PDF files.
- Extracts text from the PDFs and splits it into chunks.
- Creates a vector store (FAISS index) from the text chunks.
- Saves and loads the vector store to preserve the processed data across interactions.
- Uses Claude (AI) to answer questions based on the content of the PDFs.

Let me know if you need further clarification!