Build a Chatbot for Your Data

Estimated time needed: 60 min

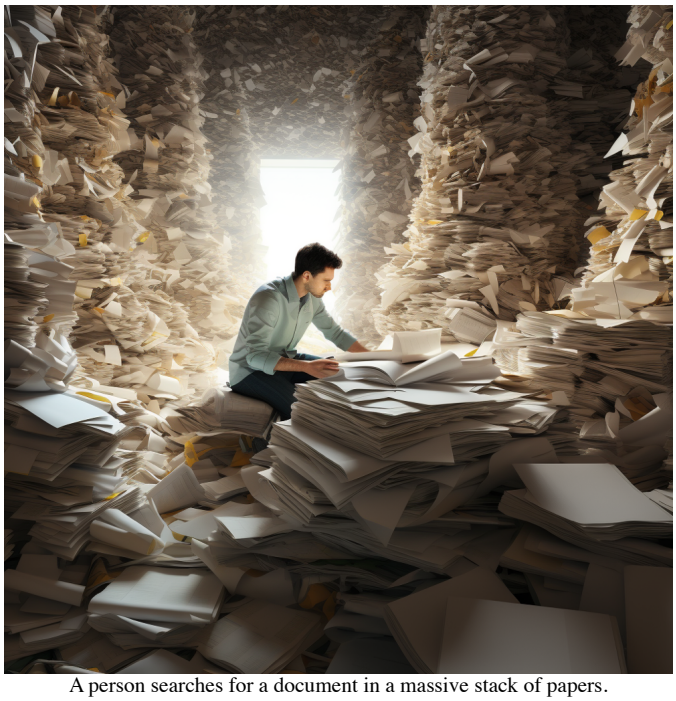
Introduction

In this project, you will create a chatbot for your own pdf file using Flask, a popular web framework, and LangChain, another popular framework for working with large language models (LLMs). The chatbot you develop will not just interact with users through text but also comprehend and answer questions related to the content of a specific document.

Click on the demo link below to try the final application that you will create!

Try the demo app

At the end of this project, you will gain a deeper understanding of chatbots, web application development using Flask and Python, and the use of LangChain framework in interpreting and responding to a wide array of user inputs. And most important, you would have built a comprehensive and impressive chatbot application!



A person searches for a document in a massive stack of papers.

Chatbots

Chatbots are software applications designed to engage in human-like conversation. They can respond to text inputs from users and are widely used in various domains, including customer service, eCommerce, and education. In this project, you will build a chatbot capable of not only engaging users in a general conversation but also answering queries based on a particular document.

LangChain

LangChain is a versatile tool for building AI-driven language applications. It provides various functionalities such as text retrieval, summarization, translation, and many more, by leveraging pretrained language models. In this project, you will be integrating LangChain into your chatbot, empowering it to understand and respond to diverse user inputs effectively.

Flask

Flask is a lightweight and flexible web framework for Python, known for its simplicity and speed. A web framework is a software framework designed to support the development of web applications, including the creation of web servers, and management of HTTP requests and responses.

You will use Flask to create the server side or backend of your chatbot. This involves handling incoming messages from users, processing these messages, and sending appropriate responses back to the user.

Routes in Flask

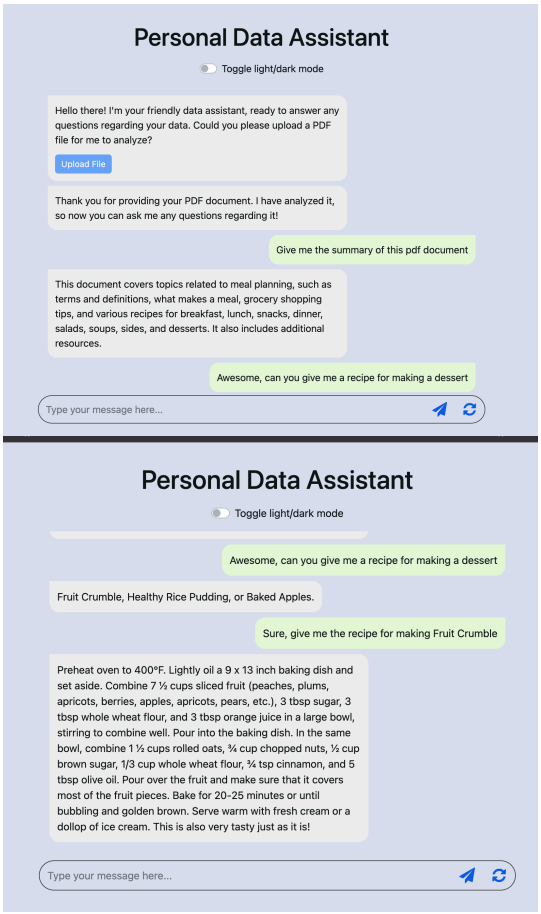
Routes are an essential part of web development. When your application receives a request from a client (typically a web browser), it needs to know how to handle that request. This is where routing comes in.

In Flask, routes are created using the @app.route decorator to bind a function to a URL route. When a user visits that URL, the associated function is executed. In your chatbot project, you will use routes to handle the POST requests carrying user messages and to process document data.

HTML - CSS - JavaScript

You are provided with a ready-to-use chatbot front-end, built with HTML, CSS, and JavaScript. HTML structures web content, CSS styles it and JavaScript adds interactivity. These technologies create a visually appealing and interactive chatbot interface.

Here is an snapshot of the interface:



Chatbot Interface

Learning objectives

At the end of this project, you will be able to:

- Explain the basics of Langchain and AI applications
- Set up a development environment for building an assistant using Python Flask
- Implement PDF upload functionality to allow the assistant to comprehend file input from users
- Integrate the assistant with open source models to give it a high level of intelligence and the ability to understand and respond to user requests
- (Optional) Deploy the PDF assistant to a web server for use by a wider audience

Prerequisites

Knowledge of the basics of HTML/CSS, JavaScript, and Python is nice to have but not essential. Each step of the process and code will have a comprehensive explanation in this lab.

With the background in mind, let's get started on your project!

Setting up and understanding the user interface

In this project, the goal is to create a chatbot with an interface that allows communication.

First, let's set up the environment by executing the following code:

```
pip3 install virtualenv
virtualenv my_env # create a virtual environment my_env
source my_env/bin/activate # activate my_env
```

The frontend will use HTML, CSS, and JavaScript. The user interface will be similar to many chatbots you see and use online. The code for the interface is provided and the focus of this guided project is to connect this interface with the backend that handles the uploading of your custom documents and integrates it with an LLM model to get customized responses. The provided code will help you to understand how the frontend and backend interact, and as you go through it, you will learn about the important parts and how it works, giving you a clear understanding of how the frontend works and how to create this simple web page.

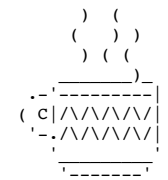
Run the following commands to retrieve the project, give it an appropriate name, and finally move to that directory by running the following:

```
git clone https://github.com/sinanazeri/build_own_chatbot_without_open_ai.git
mv build_own_chatbot_without_open_ai build_chatbot_for_your_data
cd build_chatbot_for_your_data
```

installing the requirements for the project

```
pip install -r requirements.txt
```

Have a cup of coffee, it will take 5-10 minutes to install the requirements (You can continue this project while the requirements are installed).



The next section gives a brief understanding of how the frontend works.

HTML, CSS, and JavaScript

The `index.html` file is responsible for the layout and structure of the web interface. This file contains the code for incorporating external libraries such as JQuery, Bootstrap, and FontAwesome Icons, and the CSS (`style.css`) and JavaScript code (`script.js`) that control the styling and interactivity of the interface.

The `style.css` file is responsible for customizing the visual appearance of the page's components. It also handles the loading animation using CSS keyframes. Keyframes are a way of defining the values of an animation at various points in time, allowing for a smooth transition between different styles and creating dynamic animations.

The `script.js` file is responsible for the page's interactivity and functionality. It contains the majority of the code and handles all the necessary functions such as switching between light and dark mode, sending messages, and displaying new messages on the screen. It even enables the users to record audio.

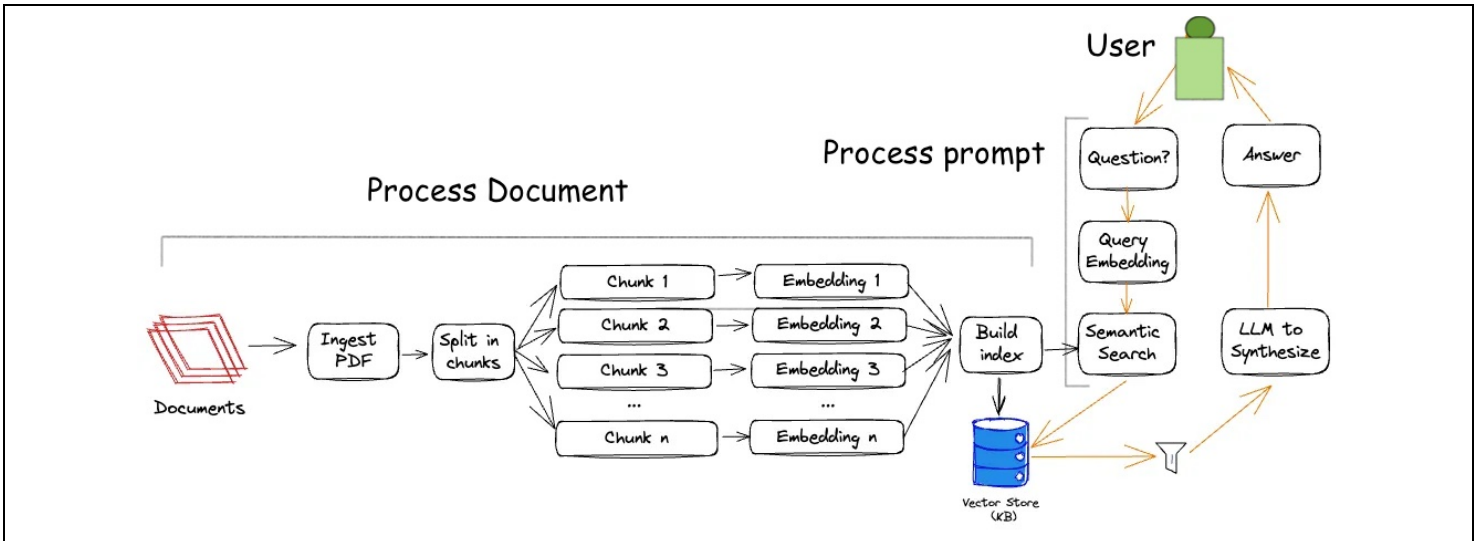
Understanding the worker: Document processing and conversation management worker, part 1

`worker.py` is part of a chatbot application that processes user messages and documents. It uses the `langchain` library, which is a Python library for building conversational AI applications. It is responsible for setting up the language model, processing PDF documents into a format that can be used for conversation retrieval, and handling user prompts to generate responses based on the processed documents. Here's a high-level overview of the script:

Open **worker.py** in IDE

Your task is to fill in the `worker.py` comments with the appropriate code.

Let's break down each section in the worker file.
The worker.py is designed to provide a conversational interface that can answer questions based on the contents of a given PDF document.



The diagram illustrates the procedure of document processing and information retrieval, seamlessly integrating a large language model (LLM) to facilitate the task of question answering. The whole process happens in `worker.py`. *image credit link*.

1. Initialization `init_llm()`:

- Setting environment variables: The environment variable for the HuggingFace API token is set.
- Loading the language model: The WatsonX language model is initialized with specified parameters.
- Loading embeddings: Embeddings are initialized using a pre-trained model.

2. Document processing `process_document(document_path)`:
   This function is responsible for processing a given PDF document.

   - Loading the document: The document is loaded using PyPDFLoader.
   - Splitting text: The document is split into smaller chunks using RecursiveCharacterTextSplitter.
   - Creating embeddings database: An embeddings database is created from the text chunks using Chroma.
   - Setting Up the RetrievalQA chain: A RetrievalQA chain is set up to facilitate the question-answering process. This chain uses the initialized language model and the embeddings database to answer questions based on the processed document.

3. User prompt processing `process_prompt(prompt)`:
   This function processes a user's prompt or question.

   - Receiving user prompt: The system receives a user prompt (question).
   - Querying the model: The model is queried using the retrieval chain, and it generates a response based on the processed document and previous chat history.
   - Updating chat history: The chat history is updated with the new prompt and response.

Delving into each section

> IBM watsonX utilizes various language models, including Llama models by Meta, which have been among the strongest open-source language model published so far (in Feb 2024).

1. Initialization `init_llm()`:

This code is for setting up and using an AI language model, from IBM watsonX:

1. **Credentials setup**: Initializes a dictionary with the service URL and an authentication token (`"skills-network"`).
2. **Parameters configuration**: Sets up model parameters like maximum token generation (`256`) and temperature (`0.1`, controlling randomness).
3. **Model initialization**: Creates a model instance with a specific `model_id`, using the credentials and parameters defined above, and specifies `"skills-network"` as the project ID.
4. **Model usage**: Initializes an interface (`WatsonxLLM`) with the configured model for interaction.

This script is specifically configured for a project or environment associated with the “skills-network”.

Complete the following code in `worker.py` by inserting the embeddings.

In this project, you do not need to specify your own `Watsonx_API` and `Project_id`. You can just specify `project_id="skills-network"` and leave `Watsonx_API` blank.

But it's important to note that this access method is exclusive to this Cloud IDE environment. If you are interested in using the model/API outside this environment (e.g., in a local environment), detailed instructions and further information are available in this tutorial.

```
# placeholder for Watsonx_API and Project_id incase you need to use the code outside this environment
Watsonx_API = "Your WatsonX API"
Project_id= "Your Project ID"
# Function to initialize the language model and its embeddings
def init_llm():
    global llm_hub, embeddings
    logger.info("Initializing WatsonxLLM and embeddings...")
    # Llama Model Configuration
    MODEL_ID = "meta-llama/llama-3-3-70b-instruct"
    WATSONX_URL = "https://us-south.ml.cloud.ibm.com"
    PROJECT_ID = "skills-network"
    # Use the same parameters as before:
    #   MAX_NEW_TOKENS: 256, TEMPERATURE: 0.1
    model_parameters = {
        # "decoding_method": "greedy",
        "max_new_tokens": 256,
        "temperature": 0.1,
    }
    # Initialize Llama LLM using the updated WatsonxLLM API
    llm_hub = WatsonxLLM(
        model_id=MODEL_ID,
        url=WATSONX_URL,
        project_id=PROJECT_ID,
        params=model_parameters
    )
    logger.debug("WatsonxLLM initialized: %s", llm_hub)
    #Initialize embeddings using a pre-trained model to represent the text data.
    embeddings =  # create object of Hugging Face Instruct Embeddings with (model_name,  model_kwargs={"device": DEVICE} )

    logger.debug("Embeddings initialized with model device: %s", DEVICE)
```

▼ Click here to see the solution

```
 embeddings = HuggingFaceInstructEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2",
    model_kwargs={"device": DEVICE}
)
logger.debug("Embeddings initialized with model device: %s", DEVICE)
    )
```

Understanding the worker, part 2

2. **Processing of documents:** `process_document` function is responsible for processing the PDF documents. It uses the PyPDFLoader to load the document, splits the document into chunks using the `RecursiveCharacterTextSplitter`, and then creates a vector store (Chroma) from the document chunks using the language model embeddings. This vector store is then used to create a retriever interface, which is used to create a `ConversationalRetrievalChain`.
    - **Document loading:** The PDF document is loaded using the `PyPDFLoader` class, which takes the path of the document as an argument. (Todo exercise: assign PyPDFLoader(…) to `loader`)
    - **Document splitting:** The loaded document is split into chunks using the `RecursiveCharacterTextSplitter` class. The `chunk_size` and `overlap` can be specified. (Todo exercise: assign RecursiveCharacterTextSplitter(…) to `text_splitter`)
    - **Vector store creation:** A vector store, which is a kind of index, is created from the document chunks using the language model embeddings. This is done using the `Chroma` class.
    - **Retrieval system setup:** A retrieval system is set up using the vector store. This system, calls a `ConversationalRetrievalChain`, used to answer questions based on the document content.

To do: complete the blank parts

```
# Function to process a PDF document
def process_document(document_path):
    global conversation_retrieval_chain
    logger.info("Loading document from path: %s", document_path)
    # Load the document
    loader =  # ---> use PyPDFLoader and document_path from the function input parameter <---
    documents = loader.load()
    logger.debug("Loaded %d document(s)", len(documents))
    # Split the document into chunks, set chunk_size=1024, and chunk_overlap=64. assign it to variable text_splitter
    text_splitter = # ---> use Recursive Character TextSplitter and specify the input parameters <---
    texts = text_splitter.split_documents(documents)
    logger.debug("Document split into %d text chunks", len(texts))
    # Create an embeddings database using Chroma from the split text chunks.
    logger.info("Initializing Chroma vector store from documents...")
    db = Chroma.from_documents(texts, embedding=embeddings)
    logger.debug("Chroma vector store initialized.")
    # Optional: Log available collections if accessible (this may be internal API)
    try:
        collections = db._client.list_collections()  # _client is internal; adjust if needed
        logger.debug("Available collections in Chroma: %s", collections)
    except Exception as e:
        logger.warning("Could not retrieve collections from Chroma: %s", e)
    # Build the QA chain, which utilizes the LLM and retriever for answering questions.
    conversation_retrieval_chain = RetrievalQA.from_chain_type(
        llm=llm_hub,
        chain_type="stuff",
        retriever=db.as_retriever(search_type="mmr", search_kwargs={'k': 6, 'lambda_mult': 0.25}),
        return_source_documents=False,
        input_key="question"
        # chain_type_kwargs={"prompt": prompt}  # if you are using a prompt template, uncomment this part
    )
    logger.info("RetrievalQA chain created successfully.")
```

▼ Click here to see the solution

```
# Function to process a PDF document
def process_document(document_path):
    global conversation_retrieval_chain
    logger.info("Loading document from path: %s", document_path)
    # Load the document
    loader = PyPDFLoader(document_path)
    documents = loader.load()
    logger.debug("Loaded %d document(s)", len(documents))
    # Split the document into chunks
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1024, chunk_overlap=64)
    texts = text_splitter.split_documents(documents)
    logger.debug("Document split into %d text chunks", len(texts))
    # Create an embeddings database using Chroma from the split text chunks.
    logger.info("Initializing Chroma vector store from documents...")
    db = Chroma.from_documents(texts, embedding=embeddings)
    logger.debug("Chroma vector store initialized.")
    # Optional: Log available collections if accessible (this may be internal API)
    try:
        collections = db._client.list_collections()  # _client is internal; adjust if needed
        logger.debug("Available collections in Chroma: %s", collections)
    except Exception as e:
        logger.warning("Could not retrieve collections from Chroma: %s", e)
    # Build the QA chain, which utilizes the LLM and retriever for answering questions.
    conversation_retrieval_chain = RetrievalQA.from_chain_type(
        llm=llm_hub,
        chain_type="stuff",
        retriever=db.as_retriever(search_type="mmr", search_kwargs={'k': 6, 'lambda_mult': 0.25}),
        return_source_documents=False,
        input_key="question"
        # chain_type_kwargs={"prompt": prompt}  # if you are using a prompt template, uncomment this part
    )
    logger.info("RetrievalQA chain created successfully.")
```

3. **Prompt processing (`process_prompt` function)**: This function handles a user's prompt or question, retrieves a response based on the contents of the previously processed PDF document, and maintains a chat history. It does the following:
    - Passes the prompt and the chat history to the `ConversationalRetrievalChain` object. `conversation_retrieval_chain` is the primary tool used to query the

language model and get an answer based on the processed PDF document's contents.
- Appends the prompt and the bot's response to the chat history.
- Returns the bot's response.

Here's a skeleton of the `process_prompt` function for the exercise:

```
# Function to process a user prompt
def process_prompt(prompt):
    global conversation_retrieval_chain
    global chat_history
    logger.info("Processing prompt: %s", prompt)
    # Query the model using the new .invoke() method
    output = conversation_retrieval_chain.invoke({"question": prompt, "chat_history": chat_history})
    answer = output["result"]
    logger.debug("Model response: %s", answer)
    # Update the chat history
    # TODO: Append the prompt and the bot's response to the chat history using chat_history.append and pass `prompt` `answer` as argument
    # --> write your code here <--

    logger.debug("Chat history updated. Total exchanges: %d", len(chat_history))
    # Return the model's response
    return answer
```

▼ Click here to see the solution

```
# Function to process a user prompt
def process_prompt(prompt):
    global conversation_retrieval_chain
    global chat_history
    logger.info("Processing prompt: %s", prompt)
    # Query the model using the new .invoke() method
    output = conversation_retrieval_chain.invoke({"question": prompt, "chat_history": chat_history})
    answer = output["result"]
    logger.debug("Model response: %s", answer)
    # Update the chat history
    chat_history.append((prompt, answer))
    logger.debug("Chat history updated. Total exchanges: %d", len(chat_history))
    # Return the model's response
    return answer
```

4. **Global variables:**

   - `llm` and `llm_embeddings` are used to store the language model and its embeddings `conversation_retrieval_chain` and `chat_history` is used to store the chat and history. `global` is used inside the functions `init_llm`, `process_document`, and `process_prompt` to indicate that the variables `llm`, `llm_embeddings`, `conversation_retrieval_chain`, and `chat_history` are global variables. This means that when these variables are modified inside these functions, the changes will persist outside the functions as well, affecting the global state of the program.

Here is the complete `worker.py`. The final code can be found in `Worker_completed.py` as well.

▼ Click here to see the complete worker.py

```
import os
import torch
import logging
# Configure logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)
from langchain_core.prompts import PromptTemplate  # Updated import per deprecation notice
from langchain.chains import RetrievalQA
from langchain_community.embeddings import HuggingFaceInstructEmbeddings  # New import path
from langchain_community.document_loaders import PyPDFLoader  # New import path
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma  # New import path
from langchain_ibm import WatsonxLLM
# Check for GPU availability and set the appropriate device for computation.
DEVICE = "cuda:0" if torch.cuda.is_available() else "cpu"
# Global variables
conversation_retrieval_chain = None
chat_history = []
llm_hub = None
embeddings = None
# Function to initialize the language model and its embeddings
def init_llm():
    global llm_hub, embeddings
    logger.info("Initializing WatsonxLLM and embeddings...")
    # Llama Model Configuration
    MODEL_ID = "meta-llama/llama-3-3-70b-instruct"
    WATSONX_URL = "https://us-south.ml.cloud.ibm.com"
    PROJECT_ID = "skills-network"
    # Use the same parameters as before:
    #   MAX_NEW_TOKENS: 256, TEMPERATURE: 0.1
    model_parameters = {
        # "decoding_method": "greedy",
        "max_new_tokens": 256,
        "temperature": 0.1,
    }
    # Initialize Llama LLM using the updated WatsonxLLM API
    llm_hub = WatsonxLLM(
        model_id=MODEL_ID,
        url=WATSONX_URL,
        project_id=PROJECT_ID,
        params=model_parameters
    )
    logger.debug("WatsonxLLM initialized: %s", llm_hub)
```

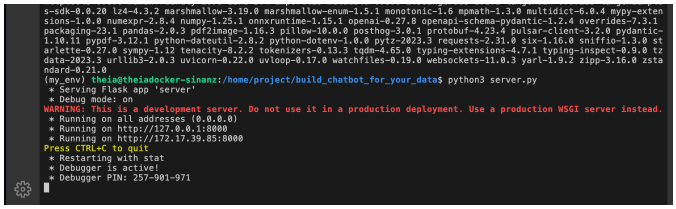
```
    # Initialize embeddings using a pre-trained model to represent the text data.
    ### --> if you are using huggingFace API:
    # Set up the environment variable for HuggingFace and initialize the desired model, and load the model into the HuggingFaceHub
    # dont forget to remove llm_hub for watsonX
    # os.environ["HUGGINGFACEHUB_API_TOKEN"] = "YOUR API KEY"
    # model_id = "tiiuae/falcon-7b-instruct"
    #llm_hub = HuggingFaceHub(repo_id=model_id, model_kwargs={"temperature": 0.1, "max_new_tokens": 600, "max_length": 600})
    embeddings = HuggingFaceInstructEmbeddings(
        model_name="sentence-transformers/all-MiniLM-L6-v2",
        model_kwargs={"device": DEVICE}
    )
    logger.debug("Embeddings initialized with model device: %s", DEVICE)
# Function to process a PDF document
def process_document(document_path):
    global conversation_retrieval_chain
    logger.info("Loading document from path: %s", document_path)
    # Load the document
    loader = PyPDFLoader(document_path)
    documents = loader.load()
    logger.debug("Loaded %d document(s)", len(documents))
    # Split the document into chunks
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=1024, chunk_overlap=64)
    texts = text_splitter.split_documents(documents)
    logger.debug("Document split into %d text chunks", len(texts))
    # Create an embeddings database using Chroma from the split text chunks.
    logger.info("Initializing Chroma vector store from documents...")
    db = Chroma.from_documents(texts, embedding=embeddings)
    logger.debug("Chroma vector store initialized.")
    # Optional: Log available collections if accessible (this may be internal API)
    try:
        collections = db._client.list_collections()  # _client is internal; adjust if needed
        logger.debug("Available collections in Chroma: %s", collections)
    except Exception as e:
        logger.warning("Could not retrieve collections from Chroma: %s", e)
    # Build the QA chain, which utilizes the LLM and retriever for answering questions.
    conversation_retrieval_chain = RetrievalQA.from_chain_type(
        llm=llm_hub,
        chain_type="stuff",
        retriever=db.as_retriever(search_type="mmr", search_kwargs={'k': 6, 'lambda_mult': 0.25}),
        return_source_documents=False,
        input_key="question"
        # chain_type_kwargs={"prompt": prompt}  # if you are using a prompt template, uncomment this part
    )
    logger.info("RetrievalQA chain created successfully.")
# Function to process a user prompt
def process_prompt(prompt):
    global conversation_retrieval_chain
    global chat_history
    logger.info("Processing prompt: %s", prompt)
    # Query the model using the new .invoke() method
    output = conversation_retrieval_chain.invoke({"question": prompt, "chat_history": chat_history})
    answer = output["result"]
    logger.debug("Model response: %s", answer)
    # Update the chat history
    chat_history.append((prompt, answer))
    logger.debug("Chat history updated. Total exchanges: %d", len(chat_history))
    # Return the model's response
    return answer
# Initialize the language model
init_llm()
logger.info("LLM and embeddings initialization complete.")
```

Running the app in CloudIDE

To implement your chatbot, you need to run the `server.py` file, first.

```
python3 server.py
```

You will have the following output in the terminal. This shows the server is running.

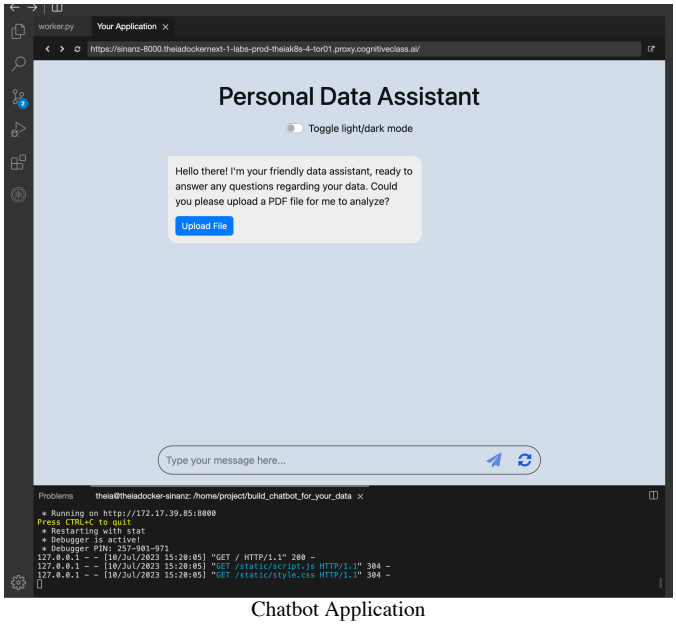


Server.py

Now click the following button to open your application:

Open the application

A new window will open for your application.



Chatbot Application

Great job, you completed your project!

If you want to understand the server file and JavaScript, continue with the project. You will also learn to containerize the app for deployment using Docker.

Once you've had a chance to run and play around with the application, please press `Crtl` (a.k.a. `control (^)` for Mac) and `C` at the same time to stop the container and continue the project (as it is also mentioned in terminal).

(optional) Using HuggingFace API for the worker

Another option for watsonX is using HuggingFace API. However, the model type and performance are very limited for the free version. As of Sep 2023, Llama2 is not supported for free, and you will use the `falcon7b` model instead.

Install the specified versions of LangChain and HuggingFace Hub, copy and paste the following commands into your terminal:

```
pip install langchain==0.1.17
pip install huggingface-hub==0.23.4
```

You need to update `init_llm` and insert your Hugging Face API key.

```
* The `HuggingFace` is hosting some LLM models which can be called free (if the model is below 10GB). You can also download any model and
```

The `HuggingFaceHub` object is created with the specified `repo_id` and additional parameters like `temperature`, `max_new_tokens`, and `max_length` to control the behavior of the model. Here you can find more examples.

- The embeddings are initialized using a class called `HuggingFaceInstructEmbeddings`, pre-trained model named `sentence-transformers/all-MiniLM-L6-v2`, and a list of leaderboards of embeddings are available here. This embedding model has shown a good balance in both performance and speed.
- The model uses the specified device (CPU or GPU) for computation.

To do: Complete the function `init_llm()`

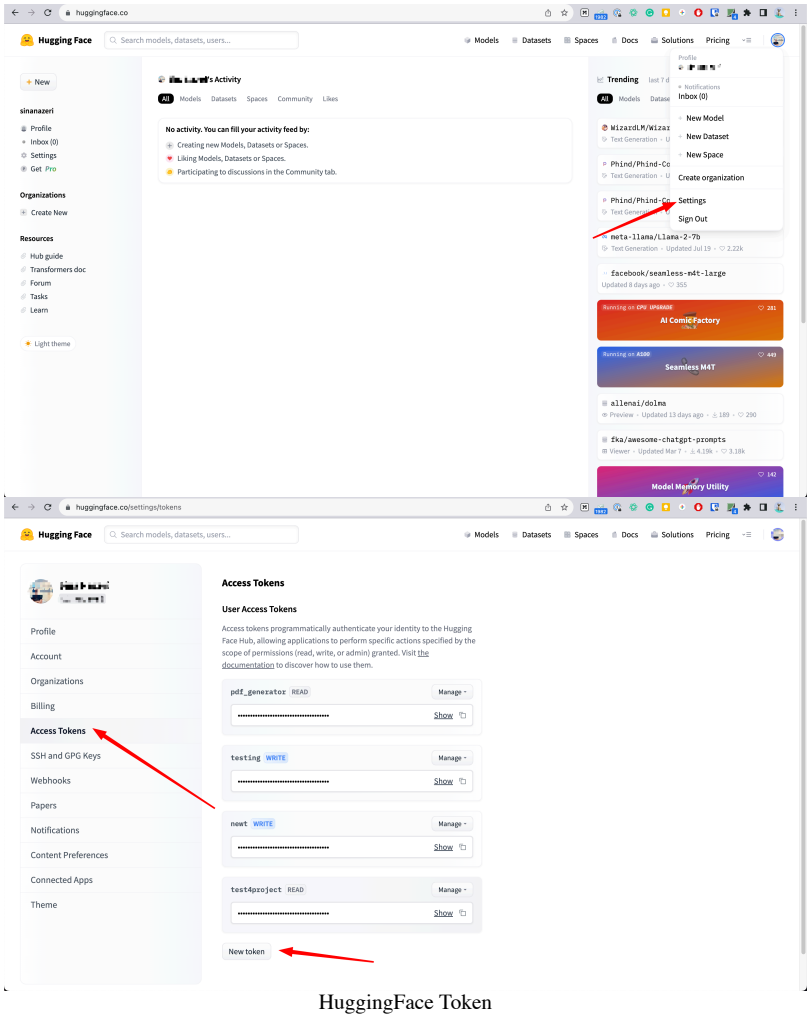
```
def init_llm():
    global llm_hub, embeddings
    # Set up the environment variable for HuggingFace and initialize the desired model.
    os.environ["HUGGINGFACEHUB_API_TOKEN"] = "Your HuggingFace API"
    # Insert the name of repo model
    model_id = "tiiuae/falcon-7b-instruct"

    # load the model into the HuggingFaceHub
    llm_hub = # --> specify hugging face hub object with (repo_id, model_kwargs={"temperature": 0.1, "max_new_tokens": 600, "max_length":

    #Initialize embeddings using a pre-trained model to represent the text data.
    embeddings = # --> create object of Hugging Face Instruct Embeddings with (model_name,  model_kwargs={"device": DEVICE} )
```

▶ Click here to see the solution

You also need to insert your LLM API key. Here's a demonstration to show you how to get your API key.

Initialize HuggingFace API key from your account with the following steps:

1. Go to the https://huggingface.co/
2. Log in to your account (or sign up free if it is your first time)
3. Go to Settings -> Access Tokens -> click on New Token (refer image below)
4. Select either read or write option and copy the token



HuggingFace Token

▶ Click here to see the complete worker.py for huggingface version

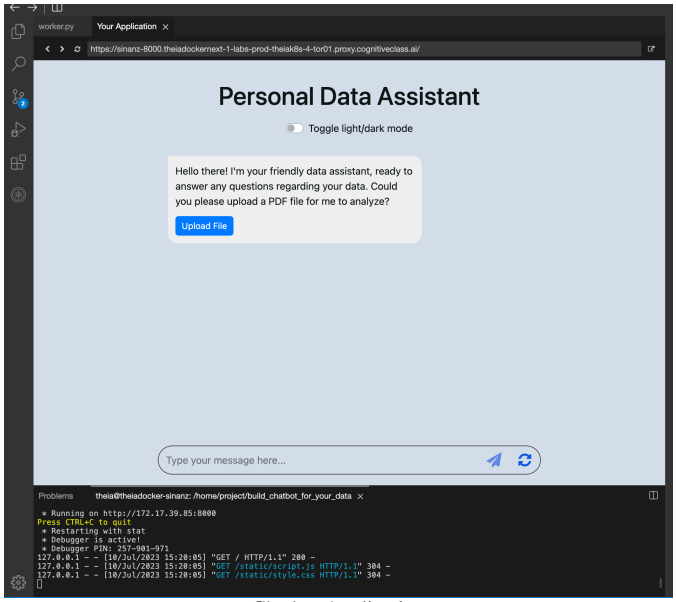
To implement your chatbot, you need to run the `server.py` file, first.

```
python3 server.py
```

Now click the following button to open your application:

Open the application

A new window will open for your application.



Chatbot Application

Understanding the server

The server is how the application will run and communicate with all of your services. Flask is a web development framework for Python and can be used as a backend for the application. It is a lightweight and simple framework that makes it quick and easy to build web applications.

With Flask, you can create web pages and applications without needing to know a lot of complex coding or use additional tools or libraries. You can create your own routes and handle user requests, and it also allows you to connect to external APIs and services to retrieve or send data.

This guided project uses Flask to handle the backend of your chatbot. This means that you will be using Flask to create routes and handle HTTP requests and responses. When a user interacts with the chatbot through the front-end interface, the request will be sent to the Flask backend. Flask will then process the request and send it to the appropriate service.

Open **server.py** in IDE

In `server.py`, at the top of the file, you import `worker` which refers to the worker.py file which you will use to handle the core logic of your chatbot. Underneath the imports, the Flask application is initialized, and a CORS policy is set. A CORS policy is used to allow or prevent web pages from making requests to different domains than the one that served the web page. Currently, it is set to `*` to allow any request.

The `server.py` file consists of 3 functions which are defined as routes, and the code to start the server.

The first route is:

```
@app.route('/', methods=['GET'])
def index():
    return render_template('index.html')
```

When a user tries to load the application, they initially send a request to go to the `/` endpoint. They will then trigger this `index` function and execute the code above. Currently, the returned code from the function is a render function to show the `index.html` file which is the frontend interface.

The second and third routes are what will be used to process all requests and handle sending information between the application. The `process_document_route()` function is responsible for handling the route when a user uploads a PDF document, processing the document, and returning a response. The `process_message_route()` function is responsible for processing a user's message or query about the processed document and returning a response from the bot.

Finally, the application is started with the `app.run` command to run on port `8080` and the host as `0.0.0.0` (a.k.a. `localhost`).

(Optional) Explaining JavaScript file `Script.js`

The JavaScript file is responsible for managing the user interface and interactions of a chatbot application. It is located in `static` folder. The main components of the file are as follows:

1. **Message processing:** The function `processUserMessage(userMessage)` sends a POST request to the server with the user's message and waits for a response. The server processes the message and returns a response that is displayed in the chat window.

   ```
   const processUserMessage = async (userMessage) => {
     let response = await fetch(baseUrl + "/process-message", {
       method: "POST",
       headers: { Accept: "application/json", "Content-Type": "application/json" },
       body: JSON.stringify({ userMessage: userMessage }),
     });
     response = await response.json();
     console.log(response);
     return response;
   };
   ```

2. **Loading animations:** The functions `showBotLoadingAnimation()` and `hideBotLoadingAnimation()` show and hide a loading animation while the server is processing a message or document.

   ```
   async function showBotLoadingAnimation() {
     await sleep(200);
     $(".loading-animation")[1].style.display = "inline-block";
     document.getElementById('send-button').disabled = true;
   }
   function hideBotLoadingAnimation() {
     $(".loading-animation")[1].style.display = "none";
     if(!isFirstMessage){
       document.getElementById('send-button').disabled = false;
     }
   }
   ```

3. **Message display:** The functions `populateUserMessage(userMessage, userRecording)` and `populateBotResponse(userMessage)` format and display user messages and bot responses in the chat window.

   ```
   const populateUserMessage = (userMessage, userRecording) => {
     $("#message-input").val("");
     $("#message-list").append(
       `<div class='message-line my-text'><div class='message-box my-text${
         !lightMode ? " dark" : ""
       }'><div class='me'>${userMessage}</div></div></div>`
     );
     scrollToBottom();
   };
   const populateBotResponse = async (userMessage) => {
     // ... omitted for brevity
     $("#message-list").append(
       `<div class='message-line'><div class='message-box${!lightMode ? " dark" : ""}'>${response.botResponse.trim()}<br>${uploadButtonHtml}
     );
     scrollToBottom();
   ```

```
};
```

4. **Input cleaning:** The function `cleanTextInput(value)` cleans the user's input to remove unnecessary spaces, newlines, tabs, and HTML tags.

```
const cleanTextInput = (value) => {
  return value
    .trim() // remove starting and ending spaces
    .replace(/[\n\t]/g, "") // remove newlines and tabs
    .replace(/<[^>]*>/g, "") // remove HTML tags
    .replace(/[<>&;]/g, ""); // sanitize inputs
};
```

5. **File upload:** The event listener for `$("#file-upload").on("change", ...)` handles the file upload process. When a file is selected, it reads the file data and sends it to the server for processing.

```
$("#file-upload").on("change", function () {
  const file = this.files[0];
  const reader = new FileReader();
  reader.onload = async function (e) {
    // Now send this data to /process-document endpoint
    let response = await fetch(baseUrl + "/process-document", {
      method: "POST",
      headers: { Accept: "application/json", "Content-Type": "application/json" },
      body: JSON.stringify({ fileData: e.target.result }),
    });
    response = await response.json6. **Chat Reset:** The event listener for `$("#reset-button").click(...)` provides a way to reset the c
```

6. **Chat reset:** It provides a way to reset the chat, clearing all messages and starting over with the initial bot greeting.

```
$("#reset-button").click(async function () {
  // Clear the message list
  $("#message-list").empty();
  // Reset the responses array
  responses.length = 0;
  // Reset isFirstMessage flag
  isFirstMessage = true;
  // Start over
  populateBotResponse();
});
```

7. **Light/Dark mode switch:** The event listener for `$("#light-dark-mode-switch").change(...)` allows the user to switch between light and dark modes for the chat interface.

```
$("#light-dark-mode-switch").change(function () {
  $("body").toggleClass("dark-mode");
  $(".message-box").toggleClass("dark");
  $(".loading-dots").toggleClass("dark");
  $(".dot").toggleClass("dark-dot");
  lightMode = !lightMode;
});
```

Running the application

First, in the `server.py` you have the following code:

```
if __name__ == "__main__":
    app.run(debug=True, port=8000, host='0.0.0.0')
```

The line `if __name__ == "__main__":` checks if the script is being run directly. If so, it starts the Flask application with `app.run(debug=True, port=8000, host='0.0.0.0')`. This starts a web server that listens on port 8000 and is accessible from any IP address (`'0.0.0.0'`). The server runs in debug mode (`debug=True`), which provides detailed error messages and automatically reloads the server when code changes.

Creating the Docker container

Docker allows for the creation of “containers” that package an application and its dependencies together. This allows the application to run consistently across different environments, as the container includes everything it needs to run. Additionally, using a Docker image to create and run applications can simplify the deployment process, as the image can be easily distributed and run on any machine that has Docker. This easy distribution of image ensures that the application runs in the same way in development, testing, and production environments.

The `git clone` from the second page already comes with a `Dockerfile` and `requirements.txt` for this application. These files are used to build the image with the dependencies already installed. Looking into the `Dockerfile` you can see it is fairly simple, it just creates a Python environment, moves all the files from the local directory to the container, installs the required packages, and then starts the application by running the `python` command.

There are 3 different containers that need to run simultaneously for the application to run and interact with Text-to-Speech and Speech-to-Text capabilities.

Starting the application

This image is quick to build as the application is quite small. These commands first build the application running the commands in the `Dockerfile` and provide a tag or name to the built container as `build-your-own-chatbot`, then run it in the foreground on `port 8000`. **You'll need to run these commands every time you wish to make a new change to one of the files.**

```
docker build . -t build_chatbot_for_your_data
docker run -p 8000:8000 build_chatbot_for_your_data
```

Open app

The application must be opened in a new tab since the minibrowser in this environment does not support certain required features.

Your browser may deny “pop-ups” but please allow them for the new tab to open up.

At this point, the application will run but return `null` for any input.

Once you've had a chance to run and play around with the application, please press `Crtl` (a.k.a. `control (^)` for Mac) and `c` at the same time to stop the container and continue the project.

The application will only run while the container is up. If you make new changes to the files and would like to test them, you will have to rebuild the image.

Conclusion

Congratulations on completing this guided project!
You learned how to implement “Retrieval Augmented Search”, in Generative AI. You also learned how to work with LLMs, and vector store, how to create Embeddings, and how to integrate everything using Langchain.You created a real application, a chatbot, using Python, Flask, and JavaScript and you packaged and deployed it using containers and Kubernetes. You can share your achievements on LinkedIn, Twitter, and other social media. The guided project detail page has buttons to help you do this.

Next steps

If generative AI and large language models (LLMs) interest you, we encourage you to apply for a free trial of the IBM watsonx.ai. WatsonX is IBM enterprise-ready AI and data platform designed to multiply the impact of AI across your business. The platform comprises three powerful products: the watsonx.ai studio for new foundation models, generative AI and machine learning; and the watsonx.data fit-for-purpose data store, built on an open lakehouse architecture; and the watsonx.governance toolkit, to accelerate AI workflows that are built with responsibility, transparency, and explainability.

Moving forward, dive deeper into chatbot creation. The following guided project can assist you in acquiring the necessary skills for that endeavor.

Create a voice assistant with IBM Watson

Furthermore, you can delve into learning about Langchain and its functionalities. This allows you to add more capabilities to the chatbot, such as analyzing various types of files and generating output plots. Following guided project can be helpful.

Create AI-powered apps with open source LangChain

Author(s)

Sina Nazeri

Talha Siddiqui