**Smart PDF Analyzer**

* **Concepts Used**: OCR (Tesseract), PyPDF2/PyMuPDF, LangChain, Azure OpenAI
* **Description**: Extract and summarize key insights from PDFs, useful for legal or financial documents.

**Project Overview**

A **Smart PDF Analyzer** will:

1. Extract text from PDFs using **PyPDF2/PyMuPDF**.
2. Use **OCR (Tesseract)** for scanned or image-based PDFs.
3. Process and clean the extracted text.
4. Generate summaries and insights using **LangChain & Azure OpenAI**.
5. (Optional) Implement a **GUI or Web App** to upload and analyze PDFs.

**Description:**

We are using PyPDF2

Extracting text from the pdf

What if the pdf contains images like bills , we need to use ocr kind of.

What if the pdf is scanned pdf like image so we cannot do in **pypdf2** that’s why we are going to convert each pdf page into image and from image we are extracting text using OCR

Even if we have invoice image inside a pdf, we convert that page into an image and extract text using tesseract, poppler, pdf2image

We are using

Pdf2image – to convert each pdf pages into image

Tesseract – for performing OCR in image

Poppler – pdf render lib, pdf2image depends on it

* **Tesseract** is an open-source OCR (Optical Character Recognition) engine used to extract text from images or scanned documents.
* **pdf2image** converts PDF files into images (usually for OCR processing), while **Poppler** is a PDF rendering library that pdf2image depends on to handle and convert PDF pages to image formats.

We need to download the Tesseract from github under releases you can download as exe file

Install it in windows add path to the system env variable

We need to download the Poppler from github under releases you can download as exe file

Install it in windows add path to the system env variable.

It will not detect in progm so add path in program also.

**Set up the project:**

**Step1: Installing these packages**

pip install pytesseract pdf2image langchain groq fastapi uvicorn

pip install python-multipart #for fastapi asking this for pdf

pip install PyPDF2, pypdf2

for pymupdf -- > pip install PyMuPDF fitz

**Step 2: Extract Text from PDFs**

* **For text-based PDFs** (PyMuPDF):

from PyPDF2 import PdfReader

#reader = PdfReader(r"C:\Users\jsril\Downloads\pdfextract.pdf")

pdf\_path = "C:\\Users\\jsril\\Downloads\\Python\_Complete\_Roadmap.pdf"

reader = PdfReader(pdf\_path)

text = ""

for page in reader.pages:

    text = text + page.extract\_text()

print(text)

**For scanned PDFs** (OCR with Tesseract):

from pdf2image import convert\_from\_path

import pytesseract

import os

# Path to your PDF

pdf\_path = "C:/Users/jsril/Downloads/pdfextract1.pdf"

# Set Poppler path in the environment variables dynamically

os.environ["PATH"] += ";C:\\Program Files\\poppler-24.08.0\\Library\\bin;C:\\poppler\\bin"

# Set the path to Tesseract executable (if not added to PATH globally)

pytesseract.pytesseract.tesseract\_cmd = r"C:\Program Files\Tesseract-OCR\tesseract.exe"

# Convert the PDF to images

images = convert\_from\_path(pdf\_path)

# Extract text using OCR

text = ""

for img in images:

    text += pytesseract.image\_to\_string(img)

print(text)

**Step 3: Summarize Using Groq**

* **Set up Groq API: (since it is free and opensource)**

from groq import Groq

import os

client = Groq()

api\_key = os.getenv('GROQ\_API\_KEY')

#os.environ['GROQ\_API\_KEY'] = "gsk\_ob7P7sSitv8VgYNJHfuoWGdyb3FYSLokevCWW3b0BXlwpOplatAj"

chat\_completion = client.chat.completions.create(

    #

    # Required parameters

    #

    messages=[

        # Set an optional system message. This sets the behavior of the

        # assistant and can be used to provide specific instructions for

        # how it should behave throughout the conversation.

        {

            "role": "system",

            "content": "you are a helpful assistant."

        },

        # Set a user message for the assistant to respond to.

        {

            "role": "user",

            "content": text,

        }

    ],

    # The language model which will generate the completion.

    model="llama-3.3-70b-versatile",

    #

    # Optional parameters

    #

    # Controls randomness: lowering results in less random completions.

    # As the temperature approaches zero, the model will become deterministic

    # and repetitive.

    temperature=0.5,

    # The maximum number of tokens to generate. Requests can use up to

    # 32,768 tokens shared between prompt and completion.

    max\_completion\_tokens=1024,

    # Controls diversity via nucleus sampling: 0.5 means half of all

    # likelihood-weighted options are considered.

    top\_p=1,

    # A stop sequence is a predefined or user-specified text string that

    # signals an AI to stop generating content, ensuring its responses

    # remain focused and concise. Examples include punctuation marks and

    # markers like "[end]".

    stop=None,

    # If set, partial message deltas will be sent.

    stream=False,

)

# Print the completion returned by the LLM.

print(chat\_completion.choices[0].message.content)

**Step 4: Build a Web API (Optional)**

Use **FastAPI** for a web-based PDF summarization API

@app.post("/upload/")

async def upload\_pdf(file: UploadFile = File(...)):

    file\_path = f"uploads/{file.filename}"

    with open(file\_path, "wb") as buffer:

        shutil.copyfileobj(file.file, buffer)

    text = extract\_text\_pymupdf(file\_path)

    summary = summarize\_text(text)

    return {"summary": summary}

-------------------------------------------------------------------

Full code smart app.py

from fastapi import FastAPI, UploadFile, File

import shutil

from groq import Groq

import os

from dotenv import load\_dotenv

load\_dotenv()

client = Groq()

api\_key = os.getenv('GROQ\_API\_KEY')

app = FastAPI()

from PyPDF2 import PdfReader

def extract\_text\_pymupdf(file\_path):

    reader = PdfReader(file\_path)

    text = ""

    for page in reader.pages:

        text = text + page.extract\_text()

    return text

def summarize\_text(text):

    chat\_completion = client.chat.completions.create(

        messages=[

        {

            "role": "system",

            "content": "you are a helpful assistant."

        },

        # Set a user message for the assistant to respond to.

        {

            "role": "user",

            "content": text, #passing our extracted text

        }

    ],

    # The language model which will generate the completion.

    model="llama-3.3-70b-versatile",

    temperature=0.5,

    # The maximum number of tokens to generate. Requests can use up to

    # 32,768 tokens shared between prompt and completion.

    max\_completion\_tokens=1024,

    # Controls diversity via nucleus sampling: 0.5 means half of all

    # likelihood-weighted options are considered.

    top\_p=1,

    # A stop sequence is a predefined or user-specified text string that

    # signals an AI to stop generating content, ensuring its responses

    # remain focused and concise. Examples include punctuation marks and

    # markers like "[end]".

    stop=None,

    # If set, partial message deltas will be sent.

    stream=False,

    )

    summary = chat\_completion.choices[0].message.content

    return summary

@app.post("/upload/")

async def upload\_pdf(file: UploadFile = File(...)):

    file\_path = f"uploads/{file.filename}"

    with open(file\_path, "wb") as buffer:

        shutil.copyfileobj(file.file, buffer)

    text = extract\_text\_pymupdf(file\_path)

    summary = summarize\_text(text)

    return {"summary": summary}

Run the app: (main.py)

uvicorn main:app –reload

uvicorn filename:app-reload

Open the **Swagger UI** at chrome:  
👉 <http://127.0.0.1:8000/docs>

Try it out option on right side click upload pdf and click execute

----------------------------------------------------------

**How to use both implementations**

**Text based pdf extraction**

**Scanned pdf extraction**

**We first check if it text based pdf or scanned pdf the process accordingly.**

**Python code:**

from fastapi import FastAPI, UploadFile, File, HTTPException

from PyPDF2 import PdfReader

import shutil

from pdf2image import convert\_from\_path

import pytesseract

import os

from groq import Groq

from dotenv import load\_dotenv

load\_dotenv()

app = FastAPI()

api\_key = os.getenv('GROQ\_API\_KEY')

client = Groq()

# Set Poppler path in the environment variables dynamically

os.environ["PATH"] += ";C:\\Program Files\\poppler-24.08.0\\Library\\bin;C:\\poppler\\bin"

# Set the path to Tesseract executable (if not added to PATH globally)

pytesseract.pytesseract.tesseract\_cmd = r"C:\Program Files\Tesseract-OCR\tesseract.exe"

UPLOAD\_FOLDER = "uploads"

os.makedirs(UPLOAD\_FOLDER, exist\_ok=True)

def extract\_text\_pypdf2(file\_path):

    text=""

    try:

        with open(file\_path, "rb") as file:

            reader = PdfReader(file)

            for page in reader.pages:

                text += page.extract\_text() or ""

    except Exception as e:

        print(f"Error extracting text with PyPDF2: {e}")

    return text.strip() if text.strip() else None

def extract\_text\_ocr(pdf\_path):

    """Extract text from scanned PDFs using OCR."""

    """Yes, Each page of the PDF is converted into an image, and the text from each page is extracted using Tesseract. Finally, the text from all pages is combined into one large string with newlines separating the text from each page."""

    images = convert\_from\_path(pdf\_path)

    text = "\n".join([pytesseract.image\_to\_string(img, lang='eng', config='--psm 6') for img in images]).strip()

    return text if text else None

def detect\_pdf\_type\_and\_extract(pdf\_path):

    """Determine whether the PDF is text-based or scanned, and extract text accordingly."""

    text = extract\_text\_pypdf2(pdf\_path)

    if text:

        return text, "Text-based PDF"

    # If no text is found, force OCR processing

    text = extract\_text\_ocr(pdf\_path)

    if text:

        return text, "Scanned PDF"

    return None, "Unknown PDF type"

def summarize\_text(text):

    try:

        chat\_completion = client.chat.completions.create(

            messages=[

            {

                "role": "system",

                "content": "you are a helpful assistant."

            },

            # Set a user message for the assistant to respond to.

            {

                "role": "user",

                "content": text,

            }

        ],

        # The language model which will generate the completion.

        model="llama-3.3-70b-versatile",

        temperature=0.5,

        max\_completion\_tokens=1024,

        top\_p=1,

        stop=None,

        # If set, partial message deltas will be sent.

        stream=False,

        )

        summary = chat\_completion.choices[0].message.content

        return summary

    except Exception as e:

        raise HTTPException(status\_code=500, detail=f"Error with AI summarization: {str(e)}")

@app.post("/upload/")

async def upload\_pdf(file: UploadFile = File(...)):

    file\_path = os.path.join(UPLOAD\_FOLDER, file.filename)

    with open(file\_path, "wb") as buffer:

        shutil.copyfileobj(file.file, buffer)

    text, pdf\_type = detect\_pdf\_type\_and\_extract(file\_path)

    if not text:

        raise HTTPException(status\_code=400, detail="Could not extract text from PDF.")

    summary = summarize\_text(text)

    return {"pdf\_type": pdf\_type, "summary": summary}

**---------------------------------------------**

**Tessearact setup :**  
To download and use Tesseract OCR in your application, follow these steps:

**Step 1: Download Tesseract OCR from GitHub**

1. **Go to the Tesseract OCR GitHub page:**
   * [Tesseract GitHub Repository](https://github.com/tesseract-ocr/tesseract)
2. **Download the latest stable release:**
   * On the GitHub page, click on **"Releases"** (on the right side of the page or under the "Code" tab).
   * Download the **latest stable version** for Windows. The file will be a .exe installer for Windows (e.g., tesseract-ocr-w32-x.x.x.setup.exe).
   * <https://github.com/tesseract-ocr/tesseract/releases/tag/5.5.0>
   * Download the exe
3. **Run the installer**:
   * After downloading, run the .exe installer.
   * During the installation, choose the default settings or customize the installation path if needed. Make sure to **select "Add Tesseract to system PATH"** during the installation, so you can use it directly from anywhere in the command line (or in your app).
   * Or else **Add Tesseract to the Path manually**:
   * **Edit the system environment variables"**.
   * Under "System variables", scroll down and find the "Path" variable.
   * Select "Path" and click Edit.
   * In the **Edit Environment Variable** dialog, click **New** and add the path to the Tesseract installation folder (without tesseract.exe at the end).
   * **C:\Program Files\Tesseract-OCR**
   * Click OK

**Step 2: Verify Installation**

After installing Tesseract, verify that it's correctly installed:

1. Open **Command Prompt** (CMD).
2. Type the following command and press **Enter**:

tesseract --version

If Tesseract is installed properly, you should see something like:

tesseract 5.0.0-alpha.20201231

**Optional: Test from Command Line given an image and test.**

You can also directly test Tesseract from the Command Prompt:

tesseract input\_image.png output\_text

This will process the input\_image.png and output the recognized text to the file output\_text.txt. in C:\Users\jsril

**Step 3: Use Tesseract in Your Application (Python)**

To use Tesseract with Python, you need the pytesseract library. Here's how to configure it:

1. **Install pytesseract** using pip:

pip install pytesseract

1. **Configure pytesseract to point to the Tesseract executable**: You need to specify the path to the Tesseract executable if it's not added to your system's PATH or if it's installed in a custom location.

Example:

import pytesseract

# Specify the path if needed

pytesseract.pytesseract.tesseract\_cmd = r"C:\Program Files\Tesseract-OCR\tesseract.exe"

Adjust the path based on where you installed Tesseract (e.g., C:\Program Files\Tesseract-OCR\tesseract.exe).

**After running you will get this error ; need to install the poppler**

**Poppler Installation**

**1. Windows:**

To install Poppler on Windows and resolve the issue, follow these steps:

**Step 1: Download Poppler**

1. Go to the Poppler for Windows download page:  
   Poppler for Windows
2. Download the latest release. It will be in a **zip** format (e.g., poppler-xx.x.x\_x86.zip).
3. Extract the zip file to a folder, for example: C:\poppler

**Step 2: Add Poppler to the PATH**

1. After extracting Poppler, the folder should contain subfolders like bin, lib, and include. The bin folder contains the executables like pdftoppm and pdfinfo that pdf2image needs.
2. Now, you need to add the **bin** folder to your system's PATH:
   * Open **Environment Variables** (Search "Environment Variables" in the Start Menu and click "Edit the system environment variables").
   * Under **System variables**, find and select **Path** and click **Edit**.
   * In the **Edit Environment Variable** window, click **New** and add the full path to the bin folder, for example: C:\poppler\bin,
   * in your system the path is C:\Program Files\poppler-24.08.0\Library\bin
3. add both paths Click **OK** to save the changes.

**Step 3: Verify the Installation**

After installing Poppler and adding it to the PATH, try running your script again. The pdf2image library should now be able to use pdfinfo and pdftoppm to process PDFs.

1. Open a new **Command Prompt** (CMD) window.
2. Type: pdfinfo

If Poppler is installed correctly, it will show the version of pdfinfo and other details.

1. You can check if you already have it installed by calling pdftoppm -h in your terminal/cmd.

--------------------------------------------------------------------------------------------

**What If the PDF Has Images of Text?**

If your PDF contains **scanned images of text** (i.e., the text is not selectable but is embedded in an image), then you'll need **OCR (Optical Character Recognition)** to extract text from those images. You can do this using **pytesseract** (as you were trying before), but this requires converting the PDF to images first.

**How to Extract Text from a Scanned PDF using OCR**

You already seem to be on the right track with using **Tesseract** OCR for this purpose, but you’ll need to **convert the scanned pages of the PDF into images** first and then pass those images to Tesseract to extract the text.

Here’s how to do it step by step:

**Step-by-Step Process**

**1. Convert Scanned PDF Pages to Images**

First, you need to convert each page of the scanned PDF into an image. You can use **pdf2image** for this.

**2. Use Tesseract to Perform OCR on the Images**

Once the pages are converted to images, you can use **Tesseract OCR** to extract the text from those images.

**Full Example Using pdf2image and pytesseract:**

1. **Install Dependencies**:
   * pdf2image: To convert PDF pages to images.
   * pytesseract: To perform OCR (text extraction) on the images.
   * **Poppler**: (Required by pdf2image) to help with PDF-to-image conversion.

You can install these libraries via pip: pip install pdf2image pytesseract

1. **Set up Tesseract**: Make sure Tesseract is installed and properly set up in your system (as we discussed in previous messages). You can download it from [Tesseract GitHub](https://github.com/tesseract-ocr/tesseract).
2. **Code to Convert PDF to Images and Extract Text**:

from pdf2image import convert\_from\_path

import pytesseract

# Set the path to your Tesseract executable (if not already added to the PATH)

pytesseract.pytesseract.tesseract\_cmd = r"C:\Program Files\Tesseract-OCR\tesseract.exe"

def extract\_text\_from\_scanned\_pdf(pdf\_path):

# Convert PDF pages to images

images = convert\_from\_path(pdf\_path)

# Initialize an empty string to collect all extracted text

full\_text = ""

# Loop through each image (representing a page)

for img in images:

# Use Tesseract to extract text from each image

text = pytesseract.image\_to\_string(img)

full\_text += text # Add the extracted text to the full text

return full\_text

# Provide the path to your scanned PDF

pdf\_path = r"C:\Users\jsril\Downloads\Python\_Complete\_Roadmap.pdf"

extracted\_text = extract\_text\_from\_scanned\_pdf(pdf\_path)

print(extracted\_text)

**Explanation:**

1. **convert\_from\_path(pdf\_path)**:
   * Converts each page of the PDF into an image (one image per page).
2. **pytesseract.image\_to\_string(img)**:
   * Uses Tesseract OCR to extract text from the image (img), which contains the scanned text.
3. **Loop through all pages**:
   * The loop iterates over all pages (images) generated from the PDF and extracts text from each one, appending it to the full\_text variable.

**Handling Large PDFs:**

If your scanned PDF is large, you may want to process a subset of pages or optimize it to save time and memory. You can specify the first\_page and last\_page parameters in convert\_from\_path() to limit the pages you want to process:

images = convert\_from\_path(pdf\_path, first\_page=1, last\_page=5) # Only process pages 1 to 5

**Other Considerations:**

* **OCR Accuracy**: The accuracy of Tesseract depends on the quality of the scanned images. Clean scans with clear text will yield better results. You can improve OCR accuracy by pre-processing the images (e.g., binarization, resizing) before passing them to Tesseract.
* **Languages**: If your PDF is not in English, you can specify a different language model for Tesseract by setting the lang parameter in image\_to\_string().

text = pytesseract.image\_to\_string(img, lang="spa") # For Spanish, for example

* **Handling Noise or Low-Quality Images**: If the scanned images are of poor quality (e.g., low resolution or background noise), you might need to clean up the images (e.g., using OpenCV) before performing OCR.

**Conclusion:**

For a **scanned PDF**, you **must convert it to images** and then use **OCR** to extract the text. The pdf2image and pytesseract libraries together make this process straightforward. Just make sure to have Tesseract installed and properly set up.

--------------------------

After setting path in env variable also your python code not detecting means

Set in python file, or dot-env file

# Set Poppler path in the environment variables dynamically

os.environ["PATH"] += ";C:\\Program Files\\poppler-24.08.0\\Library\\bin;C:\\poppler\\bin"

# Set the path to Tesseract executable (if not added to PATH globally) pytesseract.pytesseract.tesseract\_cmd = r"C:\Program Files\Tesseract-OCR\tesseract.exe"

.env file

POPPLER\_PATH=C:\Program Files\poppler-24.08.0\Library\bin;C:\poppler\bin

# Set the Poppler path from the .env file (if specified)

os.environ["PATH"] += ";" + os.getenv("POPPLER\_PATH")

----------------------------------------------------------

**More to know about fast api:**

Certainly! This code snippet is a simple **FastAPI** application that allows users to upload PDF files, extracts text from the uploaded PDF, and returns a summary of the extracted text.

Let's break down each part of this code:

**1. FastAPI Imports:**

from fastapi import FastAPI, UploadFile, File

import shutil

* **FastAPI**: This is the main class that creates the FastAPI app.
* **UploadFile**: This is a special class provided by FastAPI to handle file uploads. It allows you to easily access and process uploaded files.
* **File**: This is used to declare that a parameter in a function is going to be a file upload.
* **shutil**: This is a built-in Python library for file operations. Here, it’s used to copy the contents of the uploaded file to the local file system.

**2. FastAPI Application Setup:**

app = FastAPI()

* This creates an instance of the **FastAPI** application. This object is used to define routes and the overall API configuration.

**3. The upload\_pdf Endpoint:**

@app.post("/upload/")

async def upload\_pdf(file: UploadFile = File(...)):

* **@app.post("/upload/")**: This is a **FastAPI decorator** that binds this function to the POST HTTP method and associates it with the /upload/ endpoint.
  + In RESTful APIs, POST is typically used for uploading data (in this case, files).
  + When the user sends a POST request to http://<your-domain>/upload/, this function will be executed.
* **async def upload\_pdf(file: UploadFile = File(...)):**: This is an **asynchronous function** (hence async def). FastAPI allows for asynchronous route handlers, which makes it more efficient when dealing with I/O operations, like file uploads or database interactions.
  + file: UploadFile: This indicates that the function expects a file upload as part of the request body.
  + File(...): This tells FastAPI that the parameter file is required in the request.

**4. Save the Uploaded File:**

file\_path = f"uploads/{file.filename}"

with open(file\_path, "wb") as buffer:

shutil.copyfileobj(file.file, buffer)

* **file.filename**: This is the original filename of the uploaded file.
* **file\_path = f"uploads/{file.filename}"**: This constructs a file path to store the uploaded file in the uploads/ directory (ensure this directory exists or create it).
* **with open(file\_path, "wb") as buffer:**: This opens the destination file (in write-binary mode) where the uploaded file will be saved.
* **shutil.copyfileobj(file.file, buffer)**: This copies the contents of the uploaded file (file.file) to the destination file (buffer).

**5. Text Extraction and Summarization:**

text = extract\_text\_pymupdf(file\_path)

summary = summarize\_text(text)

* **extract\_text\_pymupdf(file\_path)**: This is a placeholder for a function that should extract text from the PDF file (file\_path). This could be implemented using a library like PyMuPDF or pdfminer to extract text from the PDF file.
* **summarize\_text(text)**: This is a placeholder for a function that should take the extracted text and summarize it. This could be done using an NLP model or a simple algorithm to condense the text.

**6. Return the Summary:**

return {"summary": summary}

* The function returns a JSON object with the key "summary" and the value as the summarized text.
* FastAPI automatically converts this Python dictionary into a proper JSON response, which will be sent back to the client.

**Full Flow of the Code:**

1. **User Uploads a PDF**:
   * The user sends a POST request to /upload/ with a PDF file as part of the request body.
   * FastAPI handles the file upload and stores the file in the uploads/ directory.
2. **Extract Text from PDF**:
   * After saving the PDF file, the application calls extract\_text\_pymupdf(file\_path) to extract the text from the PDF file.
   * extract\_text\_pymupdf is assumed to use a PDF parsing library (like PyMuPDF, pdfminer, etc.) to convert the PDF into plain text.
3. **Summarize the Text**:
   * The extracted text is passed to the summarize\_text(text) function, which condenses the text into a summary.
4. **Return the Summary**:
   * The summary is returned to the client in the form of a JSON response.

**Additional FastAPI Concepts:**

Here are a few things to know about **FastAPI** in general that might help:

1. **Request Validation**:
   * FastAPI automatically validates incoming request data based on the types of parameters in your route functions. For example, if you specify file: UploadFile, FastAPI ensures that the incoming request contains a file.
2. **Asynchronous Support**:
   * FastAPI supports asynchronous routes, making it efficient when handling I/O-bound tasks (e.g., file uploads, database queries). You can use async def for functions that involve waiting for external operations.
   * Asynchronous functions help you improve performance by not blocking the server while waiting for external resources.
3. **Dependency Injection**:
   * FastAPI supports dependency injection, which allows you to structure your application in a modular way by easily sharing common functionality like database connections or configuration settings across different routes.
4. **Automatic Documentation**:
   * FastAPI automatically generates **interactive API documentation** using **Swagger** and **ReDoc** by just running your application. You can access it at /docs or /redoc (e.g., http://127.0.0.1:8000/docs).
   * This documentation will include all the endpoints, request/response schemas, and much more, based on the Python code you write.
5. **Error Handling**:
   * FastAPI makes it easy to handle and raise custom exceptions with appropriate HTTP status codes. If something goes wrong in your route handler (like file upload errors), you can raise HTTPException with the relevant status code and message.
6. **Run the Application**:
   * To run a FastAPI application, you would typically use **Uvicorn**:
7. uvicorn main:app --reload
   * This command runs the FastAPI app (main.py in this case) with live reloading, which is useful during development.

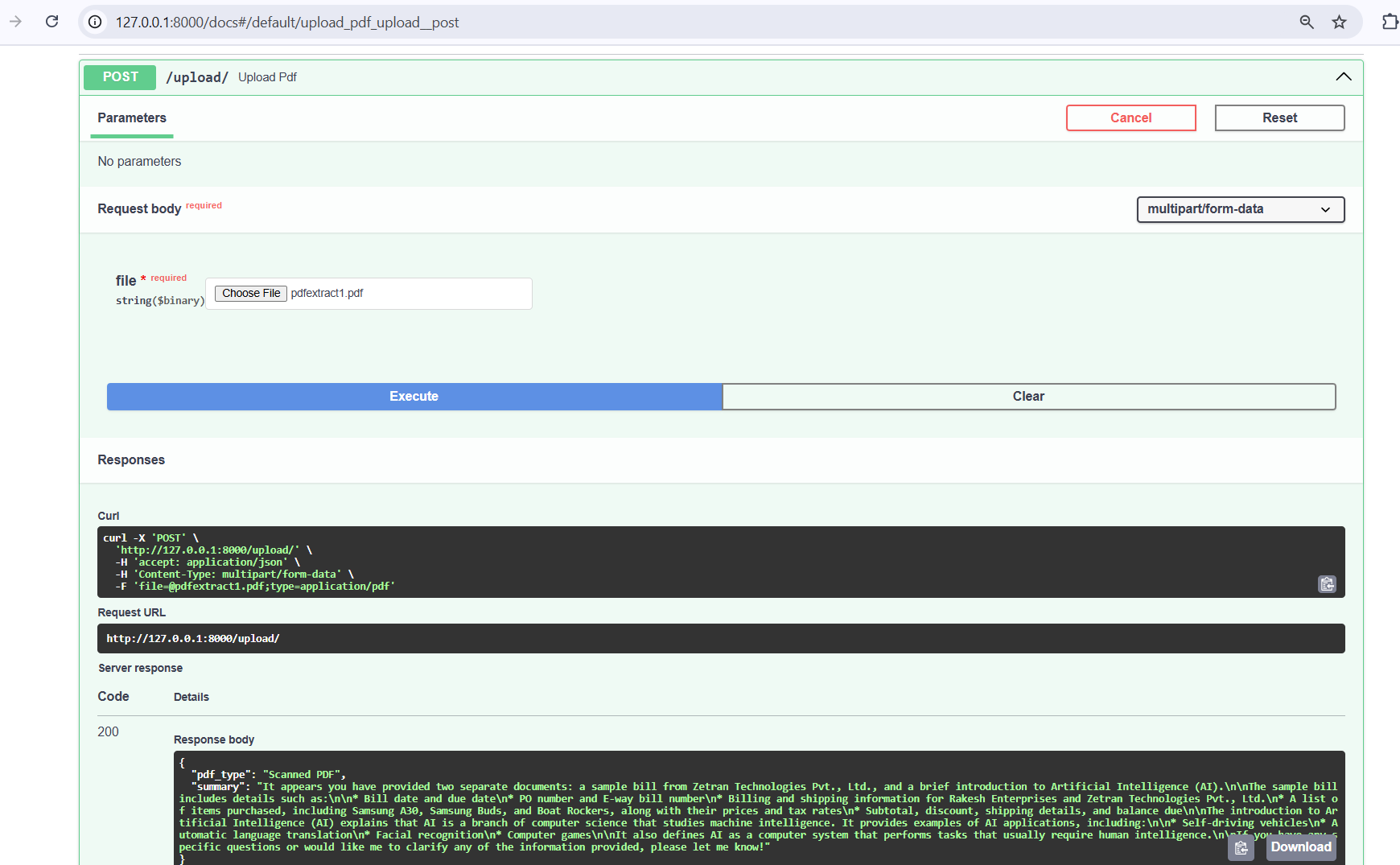
**Summary of FastAPI Features:**

* **FastAPI** is an easy-to-use web framework for building APIs with **automatic validation** and **interactive documentation**.
* It uses **Pydantic models** for automatic data validation and supports **asynchronous** programming.
* You can easily handle **file uploads**, use **dependency injection**, and make use of automatic **documentation** with Swagger UI.

If you want to learn more, you can check out the official FastAPI documentation here: [FastAPI Documentation](https://fastapi.tiangolo.com/).

Let me know if you have more specific questions or need further explanations!

Output: for scanned pdf



Output : text based pdf

A screenshot of a computer

AI-generated content may be incorrect.