Gemini
My Stuff

Gems

Chats

Building AI Document Chat Application
AI Drug Discovery Platform Poster
Setting Up Neovim on Arch Linux
LibreOffice for Students: Pros and Cons
Tauri Android SSH Client Development
Applying for Chinese Scholarship from Pakistan
Blender Reel Caption Ideas
PC Power Consumption and Electricity Usage
Cute Futuristic Robot Character Creation
Correcting Refund Process Sentence
Cover Letter Draft and Tailoring
create an animated image of a mother breastfeeding her baby holding in right position for placing at the title panel of my brochure that i'm making on brreastfeding
Cover Letter for AI/ML Internship
CLI Download Managers for Arch Linux
Email Application for Senior Video Editor
Fix Reversed Earphone Channels on Android
Crafting a Hybrid Role Message
Modeling Own Assets vs. Buying
Arch Linux Network Folder Sharing
Oak Wood Texture Creation
Adding M.2 SSD via PCIe Adapter
Free Download Manager Qt Plugin Error
Empathy in Management: Strengthening Relationships
Decision-Making Under Uncertainty Strategies
Font Recommendations for Westudios Logo
YouTube Link Extraction
Image Generation Capability Inquiry

Arabic Love Poem Translation
Instagram Bio Translation: Child Rehabilitation
China Unicom Smart Home Translation
Unable to Translate App Link
China Telecom Device Login Translation
Arabic Text Translation Request

Conversation with Gemini

How could I make a website that accepts a document "pdf, image or any format" then extracts text from it and sends to an AI model via API to generate context for it.

Then the user could choose between voice or text chat to ask questions about the given document.

That's a fantastic and increasingly common application! Building a website that handles document parsing, text extraction, AI context generation, and multi-modal chat involves several interconnected steps. This structure is often referred to as a **Retrieval-Augmented Generation (RAG)** system.

Here is a breakdown of the process and the technologies you'd need to consider:

1. 🌐 Frontend and Backend Setup

This is the foundation for your web application.

- **Frontend (User Interface):** A responsive web interface (e.g., built with **React, Vue.js, or Angular**) for file uploads, displaying the chat, and managing the voice/text chat options.
- **Backend (Application Logic):** A server (e.g., **Python/Flask/Django**, **Node.js/Express**, or **Go**) to handle file uploads, manage the processing pipeline, and route API calls.

---

2. 📄 Document Processing and Text Extraction

This is the most complex step, as you need to handle various file formats robustly.

- **File Upload and Format Handling:** The backend must accept common formats (PDF, DOCX, JPG, PNG). A server-side library will handle the file upload.
- **Text Extraction:**

  - **PDF/DOCX/Text:** Use libraries like **`PyMuPDF`** or **`pdfplumber`** (Python) for PDFs, or **`python-docx`** for Word documents.
  - **Image (OCR):** For images and scanned PDFs, you'll need **Optical Character Recognition (OCR)**. Libraries like **`Tesseract-OCR`** (via **`pytesseract`** in Python) or cloud services like **Google Cloud Document AI** or **Amazon Textract** are essential for accurate text extraction from images.
  - **Structured Data:** For documents with tables or forms, you might need more advanced, AI-powered document extraction (like the tools mentioned in the search results) to preserve structure.
- **Text Chunking:** Large documents (like a 100-page PDF) must be broken down into smaller, manageable pieces (or "chunks"). This is crucial because AI models have an **input token limit** for their API calls.

---

3. 🧠 AI Context Generation (RAG System)

After extraction, you need to prepare the text to be used as context for the user's questions.

- **Embedding Model:** Pass each text chunk through an **Embedding Model** (e.g., a model accessible via an API from OpenAI, Google, Cohere, etc.). This model converts the text into a list of numbers, or a **vector**, which represents the meaning of the text.
- **Vector Database:** Store these vectors, along with a pointer back to the original text chunk, in a **Vector Database** (e.g., **Pinecone, Weaviate, ChromaDB**). This allows for extremely fast and relevant semantic search.
  - **Context Generation:** When a user asks a question, the system first converts the question into a vector using the *same embedding model*. It then searches the Vector Database for the most **semantically similar** text chunks from the original document. These relevant chunks become the **context** for the AI model.

---

4. 🗣️ AI Chat Interaction (Text and Voice)

This final step connects the user's query to the AI model.

A. Text Chat

1. **Construct Prompt:** The backend constructs the final prompt by combining the original user question with the retrieved **context** chunks.

2. **API Call:** Send the context-aware prompt to a **Large Language Model (LLM) API** (e.g., Gemini API, OpenAI GPT, Claude).
3. **Response:** The LLM generates the answer, and the backend sends it back to the frontend for display.

B. Voice Chat

To enable voice chat, you need two additional components:

1. **Speech-to-Text (STT):**
    - The frontend records the user's voice.
    - The backend uses an STT API (e.g., **Google Speech-to-Text** or **Rev AI**) to convert the audio into a text query.
2. **Text-to-Speech (TTS):**
    - Once the AI model provides a text response, the backend uses a TTS API (e.g., **Amazon Polly** or **ElevenLabs**) to convert the text back into an audio file.
    - The backend streams this audio file to the frontend, allowing the user to listen to the answer.

This video provides an excellent walkthrough of creating a document extraction and chat application using an AI pipeline. How to Get Your Data Ready for AI Agents (Docs, PDFs, Websites)

Would you like to focus on a specific part of this pipeline, such as **Text Extraction** libraries or **Vector Database** options?

How to Get Your Data Ready for AI Agents (Docs, PDFs, Websites) - YouTube

Dave Ebbelaar · 158K views

Object 1

Lets start by building frontend in react

That's a great starting point! React is an excellent choice for building a dynamic and responsive user interface for this kind of application.

The frontend needs to handle three main responsibilities: **File Upload**, **Chat Interface**, and **Voice Recording/Playback**.

Here is a conceptual structure and the key React components you'll need.

🏗️ React Component Structure

We can break the application into three main components:

1. `App.js` (Container)

This is the main component that manages the application state (e.g., whether a document has been processed, the chat history, and the current input mode).

2. `DocumentUpload.js`

Handles the user selecting and uploading the document.

Element	Functionality
Input (`<input type="file" />`)	Accepts files with allowed extensions (`.pdf`, `.jpg`, `.png`, etc.).
Drag-and-Drop Area	A dedicated zone for users to drag documents directly.
Upload Button	Triggers the file upload to the backend (e.g., using `fetch` or `axios`).
State Indicator	Shows the status: "Uploading...", "Processing...", "Ready to Chat."

3. `ChatInterface.js`

This is the core component where the interaction happens.

Element	Functionality
`ChatHistory`	Displays the conversation history, distinguishing between user questions and AI answers.
`InputBox`	A text area for the user to type questions.
`ModeToggle`	A button or switch to toggle between **Text Mode** and **Voice Mode**.
`VoiceRecorder`	A button to start/stop recording in Voice Mode.

⚛️ Key React Code Concepts

1. Handling File Upload

You'll need to use the `FormData` interface to send the file to your backend API endpoint (e.g., `/api/upload`).

JavaScript

```
// DocumentUpload.js (Simplified)
import React, { useState } from 'react';
import axios from 'axios';

const DocumentUpload = ({ onDocumentReady }) => {
  const [file, setFile] = useState(null);
  const [status, setStatus] = useState('');

  const handleFileChange = (event) => {
    setFile(event.target.files[0]);
  };

  const handleUpload = async () => {
    if (!file) {
      alert("Please select a file first.");
      return;
    }

    const formData = new FormData();
    formData.append('document', file);

    setStatus('Uploading and processing...');

    try {
      // POST request to your backend API
      const response = await axios.post('/api/process-document', formData, {
        headers: {
          'Content-Type': 'multipart/form-data',
        },
      });

      setStatus('Document ready! Ask a question below.');
      onDocumentReady(response.data.documentId); // Pass ID to the chat component
    } catch (error) {
      setStatus('Error processing document.');
      console.error('Upload error:', error);
    }
  };

  return (
    <div>
      <input type="file" onChange={handleFileChange}
accept=".pdf,.jpg,.png,.docx" />
      <button onClick={handleUpload} disabled={!file ||
status.includes('processing')}>
        Process Document
```

```
      </button>
      <p>{status}</p>
    </div>
  );
};

export default DocumentUpload;
```

2. Implementing Voice Recording

For the voice feature, you will utilize the browser's **MediaRecorder API**.

- **Start Recording:** Request access to the user's microphone using `navigator.mediaDevices.getUserMedia({ audio: true })`.
- **Data Collection:** Use `MediaRecorder` to capture audio chunks.
- **Stop/Send:** On stop, assemble the chunks into a final audio Blob (e.g., an OGG or WAV file) and send it to your backend.

💡 Next Steps

To make this functional, we need to define the API contract between the frontend (React) and the backend server.

Would you like to start defining the **API endpoints** that your React application will use for file processing and chat interaction?

Lets just complete the frontend first

You're right, let's keep the focus strictly on the frontend. We will build a functional React interface that mimics the final experience, even without a backend connected yet.

We will build this in three parts:

1. **Project Setup & Libraries**
2. **`DocumentUpload.js`** (Handling the file input)
3. **`ChatInterface.js`** (Handling text/voice input and displaying messages)

1. Project Setup

Open your terminal and create a new React app. We will also install `lucide-react` for some nice icons (like microphone, upload, send).

Bash

```
npx create-react-app doc-chat-app
cd doc-chat-app
npm install lucide-react
npm start
```

Now, create a folder named `components` inside `src` (`src/components`).

---

2. The File Upload Component (`src/components/DocumentUpload.js`)

This component handles file selection. It simulates an "uploading" state so you can see how the UI reacts.

JavaScript

```
import React, { useState } from 'react';
import { Upload, FileText, CheckCircle, AlertCircle } from 'lucide-react';

const DocumentUpload = ({ onUploadComplete }) => {
  const [file, setFile] = useState(null);
  const [uploadStatus, setUploadStatus] = useState('idle'); // idle, uploading, 
success, error

  const handleFileChange = (e) => {
    if (e.target.files && e.target.files[0]) {
      setFile(e.target.files[0]);
      setUploadStatus('idle');
    }
  };

  const handleUpload = () => {
    if (!file) return;

    setUploadStatus('uploading');

    // SIMULATION: Simulate a 2-second backend upload process
    setTimeout(() => {
      setUploadStatus('success');
      // Pass the filename back to parent to switch views
      onUploadComplete(file.name);
    }, 2000);
  };

  return (
    <div className="upload-container">
      <div className="upload-box">
        <div className="icon-area">
          {uploadStatus === 'success' ? (
            <CheckCircle size={48} color="#10B981" />
          ) : (
            <Upload size={48} color="#6366f1" />
          )}
        </div>
```

```
        <h2>Upload your Document</h2>
        <p>PDF, Images, or DOCX</p>

        <input
          type="file"
          id="file-upload"
          className="hidden-input"
          onChange={handleFileChange}
          accept=".pdf,.jpg,.jpeg,.png,.docx"
        />

        <label htmlFor="file-upload" className="file-label">
          {file ? file.name : "Choose File"}
        </label>

        {file && (
          <button
            className="upload-btn"
            onClick={handleUpload}
            disabled={uploadStatus === 'uploading' || uploadStatus === 'success'}
          >
            {uploadStatus === 'uploading' ? 'Processing...' : 'Start Analysis'}
          </button>
        )}

        {uploadStatus === 'error' && (
          <p className="error-msg"><AlertCircle size={16}/> Upload failed</p>
        )}
      </div>
    </div>
  );
};

export default DocumentUpload;
```

---

3. The Chat Interface (`src/components/ChatInterface.js`)

This component needs to handle switching between text and voice.

- **Text:** Standard chat bubbles.
- **Voice:** We use the browser's native `MediaRecorder` API to capture audio.

JavaScript

```
import React, { useState, useRef, useEffect } from 'react';
import { Mic, Send, StopCircle, FileText } from 'lucide-react';

const ChatInterface = ({ fileName }) => {
  const [messages, setMessages] = useState([
    { id: 1, sender: 'ai', text: `I've analyzed ${fileName}. What would you like to
know?` }
  ]);
```

```
const [inputText, setInputText] = useState('');
const [isRecording, setIsRecording] = useState(false);
const [isAiTyping, setIsAiTyping] = useState(false);

const mediaRecorderRef = useRef(null);
const audioChunksRef = useRef([]);

// --- HANDLE SENDING MESSAGES ---
const handleSendMessage = (text) => {
  if (!text.trim()) return;

  // 1. Add User Message
  const newMessage = { id: Date.now(), sender: 'user', text: text };
  setMessages((prev) => [...prev, newMessage]);
  setInputText('');

  // 2. Simulate AI Response
  setIsAiTyping(true);
  setTimeout(() => {
    const aiResponse = {
      id: Date.now() + 1,
      sender: 'ai',
      text: "This is a simulated response based on the document context."
    };
    setMessages((prev) => [...prev, aiResponse]);
    setIsAiTyping(false);
  }, 1500);
};

// --- VOICE RECORDING LOGIC ---
const startRecording = async () => {
  try {
    const stream = await navigator.mediaDevices.getUserMedia({ audio: true });
    mediaRecorderRef.current = new MediaRecorder(stream);
    audioChunksRef.current = [];

    mediaRecorderRef.current.ondataavailable = (event) => {
      if (event.data.size > 0) {
        audioChunksRef.current.push(event.data);
      }
    };

    mediaRecorderRef.current.onstop = () => {
      // Create audio blob (In real app, send this to backend)
      const audioBlob = new Blob(audioChunksRef.current, { type: 'audio/wav' });
      const audioUrl = URL.createObjectURL(audioBlob);

      // Simulate sending audio and getting text back
      handleSendMessage("🎤 [Voice Audio Sent]");
    };

    mediaRecorderRef.current.start();
    setIsRecording(true);
  } catch (err) {
    console.error("Error accessing microphone:", err);
    alert("Could not access microphone.");
```

```
    }
  };

  const stopRecording = () => {
    if (mediaRecorderRef.current) {
      mediaRecorderRef.current.stop();
      setIsRecording(false);
    }
  };

  return (
    <div className="chat-container">
      {/* Header */}
      <div className="chat-header">
        <FileText size={20} />
        <span>Context: {fileName}</span>
      </div>

      {/* Messages Area */}
      <div className="messages-area">
        {messages.map((msg) => (
          <div key={msg.id} className={`message-bubble ${msg.sender}`}>
            {msg.text}
          </div>
        ))}
        {isAiTyping && <div className="message-bubble ai typing">Typing...</div>}
      </div>

      {/* Input Area */}
      <div className="input-area">
        <input
          type="text"
          placeholder="Ask a question..."
          value={inputText}
          onChange={(e) => setInputText(e.target.value)}
          onKeyPress={(e) => e.key === 'Enter' && handleSendMessage(inputText)}
          disabled={isRecording}
        />

        {/* Toggle between Mic and Send based on input */}
        {inputText ? (
          <button className="icon-btn send" onClick={() =>
handleSendMessage(inputText)}>
            <Send size={20} />
          </button>
        ) : (
          <button
            className={`icon-btn mic ${isRecording ? 'recording' : ''}`}
            onClick={isRecording ? stopRecording : startRecording}
          >
            {isRecording ? <StopCircle size={20} /> : <Mic size={20} />}
          </button>
        )}
      </div>
    </div>
  );
```

```
};

export default ChatInterface;
```

---

4. Connect it all (`src/App.js` & `src/App.css`)

`App.js`

JavaScript

```
import React, { useState } from 'react';
import './App.css';
import DocumentUpload from './components/DocumentUpload';
import ChatInterface from './components/ChatInterface';

function App() {
  const [currentView, setCurrentView] = useState('upload'); // 'upload' or 'chat'
  const [fileName, setFileName] = useState('');

  const handleUploadComplete = (name) => {
    setFileName(name);
    setCurrentView('chat');
  };

  return (
    <div className="App">
      <header className="app-header">
        <h1>DocuChat AI</h1>
      </header>
      <main className="app-main">
        {currentView === 'upload' ? (
          <DocumentUpload onUploadComplete={handleUploadComplete} />
        ) : (
          <ChatInterface fileName={fileName} />
        )}
      </main>
    </div>
  );
}

export default App;
```

`App.css` (Add this to make it look decent)

CSS

```
/* Basic Reset & Layout */
body { font-family: 'Inter', sans-serif; background-color: #f3f4f6; margin: 0; }
.App { display: flex; flex-direction: column; height: 100vh; max-width: 600px;
margin: 0 auto; background: white; box-shadow: 0 0 20px rgba(0,0,0,0.05); }

/* Header */
.app-header { padding: 1rem; border-bottom: 1px solid #e5e7eb; text-align:
center; }
```

```
.app-header h1 { margin: 0; font-size: 1.25rem; color: #1f2937; }
.app-main { flex: 1; display: flex; flex-direction: column; overflow: hidden; }

/* Upload Component */
.upload-container { flex: 1; display: flex; align-items: center; justify-content:
center; padding: 2rem; }
.upload-box { text-align: center; border: 2px dashed #cbd5e1; padding: 2rem;
border-radius: 12px; width: 100%; }
.hidden-input { display: none; }
.file-label { display: block; margin: 1rem auto; padding: 0.5rem 1rem; background:
#f1f5f9; border-radius: 6px; cursor: pointer; max-width: 200px; white-space:
nowrap; overflow: hidden; text-overflow: ellipsis;}
.upload-btn { background: #6366f1; color: white; border: none; padding: 0.75rem
1.5rem; border-radius: 6px; cursor: pointer; font-weight: 600; margin-top: 1rem; }
.upload-btn:disabled { background: #a5b4fc; cursor: not-allowed; }

/* Chat Component */
.chat-container { display: flex; flex-direction: column; height: 100%; }
.chat-header { padding: 0.75rem; background: #f8fafc; border-bottom: 1px solid
#e2e8f0; display: flex; align-items: center; gap: 0.5rem; font-size: 0.9rem; color:
#64748b; }

.messages-area { flex: 1; overflow-y: auto; padding: 1rem; display: flex; flex-
direction: column; gap: 1rem; }
.message-bubble { max-width: 80%; padding: 0.75rem 1rem; border-radius: 12px; line-
height: 1.4; }
.message-bubble.user { align-self: flex-end; background: #6366f1; color: white;
border-bottom-right-radius: 2px; }
.message-bubble.ai { align-self: flex-start; background: #f1f5f9; color: #1f2937;
border-bottom-left-radius: 2px; }
.message-bubble.typing { font-style: italic; opacity: 0.7; }

.input-area { padding: 1rem; border-top: 1px solid #e2e8f0; display: flex; gap:
0.5rem; align-items: center; }
.input-area input { flex: 1; padding: 0.75rem; border: 1px solid #cbd5e1; border-
radius: 24px; outline: none; }
.input-area input:focus { border-color: #6366f1; }
.icon-btn { width: 40px; height: 40px; border-radius: 50%; border: none; display:
flex; align-items: center; justify-content: center; cursor: pointer; transition:
all 0.2s; }
.icon-btn.send { background: #6366f1; color: white; }
.icon-btn.mic { background: #f1f5f9; color: #475569; }
.icon-btn.mic.recording { background: #ef4444; color: white; animation: pulse 1.5s
infinite; }

@keyframes pulse { 0% { transform: scale(1); } 50% { transform: scale(1.1); } 100%
{ transform: scale(1); } }
```

How to Run This

1. Copy the code into the respective files.
2. Make sure your terminal is in the project folder.
3. Run `npm start`.

This will give you a fully interactive mockup. You can "upload" a file, see the processing state, enter the chat, type messages, and even click the microphone (which will actually ask for permission and record chunks, though it just sends a placeholder text for now).

Would you like to move on to setting up the **Backend (Python/Flask)** to actually process these files?

Yes let's setup a backend and let's start with processing different file uploads

Great choice. We will use **Python** with **Flask** because it has excellent libraries for document processing (PDFs, Images, Word docs) and is very easy to set up.

🛠️ Prerequisites

Before we write code, you need two things installed on your computer:

1. **Python** (3.8 or higher)
2. **Tesseract OCR:** This is the engine that reads text from images.
   - *Windows:* Download the installer from UB-Mannheim/tesseract/wiki.
   - *Mac:* `brew install tesseract`
   - *Linux:* `sudo apt-get install tesseract-ocr`

---

1. Project Structure

Create a new folder called `backend` next to your frontend folder.

Plaintext

```
/backend
  ├── uploads/          # Folder where we save incoming files temporarily
  ├── app.py            # The main Flask server
  ├── processor.py      # The logic to extract text from files
  └── requirements.txt  # List of python libraries
```

2. Install Libraries

Create `requirements.txt` inside the `backend` folder:

Plaintext

```
Flask
flask-cors
werkzeug
PyMuPDF
python-docx
```

```
pytesseract
Pillow
```

Open your terminal in the `backend` folder and run:

Bash

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

---

3. The Processing Logic (`processor.py`)

This file handles the messy work of opening different file types and pulling out the text.

Python

```
import os
import fitz  # PyMuPDF for PDFs
import pytesseract
from PIL import Image
from docx import Document

# NOTE: If on Windows, you might need to point to the tesseract exe manually:
# pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-
OCR\tesseract.exe'

ALLOWED_EXTENSIONS = {'pdf', 'png', 'jpg', 'jpeg', 'docx'}

def is_allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

def extract_text_from_pdf(filepath):
    text = ""
    with fitz.open(filepath) as doc:
        for page in doc:
            text += page.get_text()
    return text

def extract_text_from_image(filepath):
    image = Image.open(filepath)
    text = pytesseract.image_to_string(image)
    return text

def extract_text_from_docx(filepath):
    doc = Document(filepath)
    text = "\n".join([para.text for para in doc.paragraphs])
    return text

def process_file(filepath):
    """Determines file type and extracts text accordingly."""
    ext = filepath.rsplit('.', 1)[1].lower()

    try:
```

```
        if ext == 'pdf':
            return extract_text_from_pdf(filepath)
        elif ext in ['png', 'jpg', 'jpeg']:
            return extract_text_from_image(filepath)
        elif ext == 'docx':
            return extract_text_from_docx(filepath)
        else:
            return "Unsupported file format."
    except Exception as e:
        print(f"Error processing file: {e}")
        return None
```

---

4. The Server (`app.py`)

This file receives the file from your React frontend, saves it, and asks `processor.py` to extract the text.

Python

```
import os
from flask import Flask, request, jsonify
from flask_cors import CORS
from werkzeug.utils import secure_filename
from processor import process_file, is_allowed_file

app = Flask(__name__)
CORS(app)  # Allow React to talk to this server

# Configure upload folder
UPLOAD_FOLDER = 'uploads'
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

@app.route('/api/process-document', methods=['POST'])
def upload_file():
    if 'document' not in request.files:
        return jsonify({'error': 'No file part'}), 400

    file = request.files['document']

    if file.filename == '':
        return jsonify({'error': 'No selected file'}), 400

    if file and is_allowed_file(file.filename):
        filename = secure_filename(file.filename)
        filepath = os.path.join(app.config['UPLOAD_FOLDER'], filename)
        file.save(filepath)

        # Process the file to extract text
        extracted_text = process_file(filepath)

        if extracted_text:
            # In the next step, we will send this text to an AI/Vector DB.
```

```
            # For now, we return it to the frontend to prove it works.
            return jsonify({
                'message': 'File processed successfully',
                'filename': filename,
                'extracted_text_preview': extracted_text[:500] + "..." # First 500 
chars
            })
        else:
            return jsonify({'error': 'Failed to extract text'}), 500

    return jsonify({'error': 'File type not allowed'}), 400

if __name__ == '__main__':
    app.run(debug=True, port=5000)
```

---

5. Connecting Frontend to Backend

Now we need to update your React `DocumentUpload.js` to actually call this Python server instead of using the simulation `setTimeout`.

Update `src/components/DocumentUpload.js` (inside the `handleUpload` function):

JavaScript

```
// Add this import at the top
// Note: You might need to run `npm install axios` if you haven't yet.
// import axios from 'axios';

const handleUpload = async () => {
    if (!file) return;

    setUploadStatus('uploading');

    const formData = new FormData();
    formData.append('document', file);

    try {
        // Ensure your Python server is running on port 5000
        const response = await fetch('http://localhost:5000/api/process-document', 
{
            method: 'POST',
            body: formData,
        });

        const data = await response.json();

        if (response.ok) {
            setUploadStatus('success');
            console.log("Extracted Text:", data.extracted_text_preview); // Check 
console to see result!

            // Wait a moment so user sees the success checkmark
            setTimeout(() => {
```

```
                onUploadComplete(data.filename);
            }, 1000);
        } else {
            setUploadStatus('error');
            console.error(data.error);
        }
    } catch (error) {
        console.error("Upload failed", error);
        setUploadStatus('error');
    }
};
```

🏃 How to Run

1. **Terminal 1 (Backend):**

   Bash

-
```
cd backend
python app.py
```

You should see `Running on http://127.0.0.1:5000`

- **Terminal 2 (Frontend):**

Bash

2.
```
cd doc-chat-app
npm start
```

3. **Test it:** Upload an image or PDF. Watch the Python terminal—it will print logs showing it received the file. Check your browser console to see the extracted text!

The next step is the "AI Brain": taking that extracted text and **creating Embeddings** so the AI can understand it. Do you want to use **OpenAI** (easier, paid) or a **Local Model** (free, slightly harder setup) for this?

```
[ali@archlinux Backend]$ pip install -r requirements.txt

error: externally-managed-environment


× This environment is externally managed
 ╰─> To install Python packages system-wide, try 'pacman -S
    python-xyz', where xyz is the package you are trying to
    install.
```

```
    If you wish to install a non-Arch-packaged Python package,
    create a virtual environment using 'python -m venv path/to/venv'.
    Then use path/to/venv/bin/python and path/to/venv/bin/pip.

    If you wish to install a non-Arch packaged Python application,
    it may be easiest to use 'pipx install xyz', which will manage a
    virtual environment for you. Make sure you have python-pipx
    installed via pacman.

note: If you believe this is a mistake, please contact your Python installation or OS distribution provider. You can override this, at the risk of breaking your Python installation or OS, by passing --break-system-packages.
hint: See PEP 668 for the detailed specification.
```

This is a standard safety feature on Arch Linux (and other modern distros) to stop `pip` from breaking your system tools. Arch manages Python packages via `pacman`, so installing global packages with `pip` can cause conflicts.

The correct and cleanest way to fix this is to create a **Virtual Environment** for your project. This keeps your project's libraries isolated from your Arch system.

1. Create the Virtual Environment

Run these commands inside your `Backend` folder:

Bash

```
# Create a virtual environment named "venv"
python -m venv venv
```

2. Activate the Environment

You need to "step into" this environment before installing anything.

Bash

```
# Activate the virtual environment
source venv/bin/activate
```

You should see `(venv)` appear at the start of your terminal prompt.

3. Install Requirements

Now that you are inside the virtual environment, you can safely install your packages:

Bash

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

⚠️ Important Notes for Arch Users

1. **System Packages:** You *could* install these using `sudo pacman -S python-flask python-pillow ...` etc., but it is **not recommended** for development projects because versions might mismatch what your code needs.
2. **Future Sessions:** Every time you open a new terminal to run your backend, you must run `source venv/bin/activate` first.
3. **System Dependencies:** `PyMuPDF` and `Pillow` usually install fine, but if you get build errors, you might need base development tools installed on Arch:

   Bash

```
3. sudo pacman -S base-devel python-setuptools
```

Once the installation finishes successfully, you can run `python app.py`.

Would you like to proceed with setting up the AI/Embedding part now, or did you hit any other errors during installation?

Now do I need to copy my whole backend files and directories inside the env dir?

No, **do not** move your files inside the `venv` directory!

That `venv` folder is just a storage box for the libraries (like Flask and PyMuPDF) that you downloaded. It is not meant to hold your own project code.

✅ Correct Structure

Your project files (`app.py`, `processor.py`) should sit **next to** the `venv` folder, not inside it.

Your `Backend` folder should look like this:

Plaintext

```
Backend/
├── venv/                 <-- Leave this alone (contains libraries)
├── app.py                <-- Your code (Keep it here)
```

```
├── processor.py          <-- Your code (Keep it here)
├── requirements.txt      <-- Your list of libraries
└── uploads/              <-- Where files get saved
```

🚀 How to Run It Correctly

1. **Stay in your `Backend` folder** (where `app.py` is).
2. **Activate the environment** (if you haven't already):

   Bash

-
```
source venv/bin/activate
```

(You will see `(venv)` in your terminal prompt)

- **Run your app:**

Bash

```
3. python app.py
```

That's it! Python will automatically look inside the `venv` folder to find the libraries it needs while running the `app.py` file sitting right in front of you.

Ready to connect the **AI Brain** (Embeddings)? Do you prefer using **OpenAI** (easy, costs money) or a **Local Model** (free, runs on your PC)?

```
Compiled with problems:

×

ERROR

[eslint]

src/components/DocumentUpload.js

Line 9:10: 'file' is not defined no-undef

Line 11:5: 'setUploadStatus' is not defined no-undef

Line 14:33: 'file' is not defined no-undef

Line 26:13: 'setUploadStatus' is not defined no-undef

Line 31:17: 'onUploadComplete' is not defined no-undef

Line 34:13: 'setUploadStatus' is not defined no-undef

Line 39:9: 'setUploadStatus' is not defined no-undef
```

Search for the keywords to learn more about each error.

It looks like when you updated the code, the `handleUpload` function got separated from the variables defined at the top of the component (like `file` and `uploadStatus`).

To fix this, you need to replace the **entire content** of `src/components/DocumentUpload.js` with the code below. This combines the UI we built earlier with the backend connection logic we just added.

📄 Updated `src/components/DocumentUpload.js`

Copy this entire block and overwrite your existing file:

JavaScript

```
import React, { useState } from 'react';
import { Upload, CheckCircle, AlertCircle } from 'lucide-react';

const DocumentUpload = ({ onUploadComplete }) => {
  // 1. State Definitions
  const [file, setFile] = useState(null);
  const [uploadStatus, setUploadStatus] = useState('idle'); // idle, uploading,
success, error

  // 2. Handle File Selection
  const handleFileChange = (e) => {
    if (e.target.files && e.target.files[0]) {
      setFile(e.target.files[0]);
      setUploadStatus('idle');
    }
  };

  // 3. Handle Upload to Backend
  const handleUpload = async () => {
    if (!file) return;

    setUploadStatus('uploading');

    // Prepare the file for sending
    const formData = new FormData();
    formData.append('document', file);

    try {
        // Send to your Python Backend
        const response = await fetch('http://localhost:5000/api/process-document',
{
            method: 'POST',
            body: formData,
        });

        const data = await response.json();
```

```
        if (response.ok) {
            setUploadStatus('success');
            console.log("Server Response:", data); // Check console for extracted
text

            // Wait 1 second so user sees the green checkmark, then move to chat
            setTimeout(() => {
                onUploadComplete(data.filename);
            }, 1000);
        } else {
            setUploadStatus('error');
            console.error("Server Error:", data.error);
        }
    } catch (error) {
        console.error("Network Error:", error);
        setUploadStatus('error');
    }
  };

  return (
    <div className="upload-container">
      <div className="upload-box">
        <div className="icon-area">
          {uploadStatus === 'success' ? (
            <CheckCircle size={48} color="#10B981" />
          ) : (
            <Upload size={48} color="#6366f1" />
          )}
        </div>

        <h2>Upload your Document</h2>
        <p>PDF, Images, or DOCX</p>

        <input
          type="file"
          id="file-upload"
          className="hidden-input"
          onChange={handleFileChange}
          accept=".pdf,.jpg,.jpeg,.png,.docx"
        />

        <label htmlFor="file-upload" className="file-label">
          {file ? file.name : "Choose File"}
        </label>

        {file && (
          <button
            className="upload-btn"
            onClick={handleUpload}
            disabled={uploadStatus === 'uploading' || uploadStatus === 'success'}
          >
            {uploadStatus === 'uploading' ? 'Processing...' : 'Start Analysis'}
          </button>
        )}
```

```
        {uploadStatus === 'error' && (
          <p className="error-msg" style={{color: 'red', marginTop: '10px'}}>
            <AlertCircle size={16} style={{display:'inline',
verticalAlign:'middle'}}/>
            Upload failed. Check backend console.
          </p>
        )}
      </div>
    </div>
  );
};

export default DocumentUpload;
```

🔍 Why did this happen?

In React, functions like `handleUpload` must be **inside** the component function (`const DocumentUpload = ...`) to see variables like `file` and `setUploadStatus`. The errors occurred because the function was likely pasted outside the component's closing bracket or the variables weren't defined in the same scope.

Try running it now! (Make sure your Python backend is running in the other terminal).