**Project Documentation: Learning Assistant video transcriber, summarizer, mcq generator and notes generator**

**Overview**

This project is an automated video processing system designed to perform the following tasks:

1. **Download YouTube videos**
2. **Transcribe video audio**
3. **Summarize transcriptions**
4. **Generate notes**
5. **Create multiple-choice questions (MCQs)**

The system utilizes various Python libraries and frameworks and is built with a FastAPI backend and a React frontend, orchestrated using Docker.

**Project Structure**

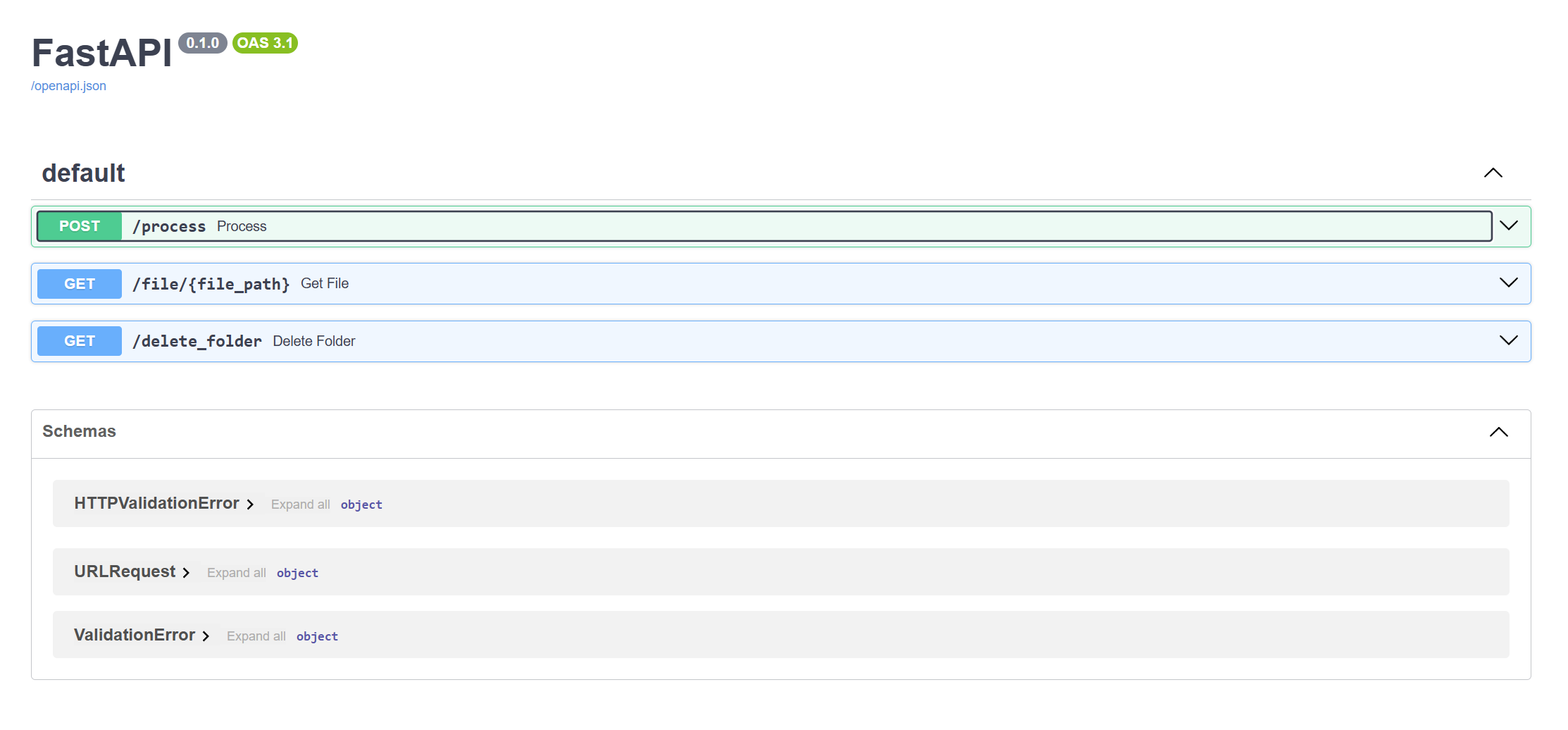
**Backend (server/)**

* main.py: The main FastAPI application.
* Dockerfile: Docker configuration for the FastAPI backend.
* requirements.txt: Python dependencies for the backend.
* component/Youtube\_reader/youtube\_reader2.py: YouTube video downloader.
* component/Video\_Transcriber2/video\_transcriber.py: Video transcriber.
* component/Summarizer2/summarize.py: Text summarizer.
* component/Note\_Generator2/note\_gen.py: Note generator.
* component/Mcq\_Generator2/mcq\_gen.py: MCQ generator.



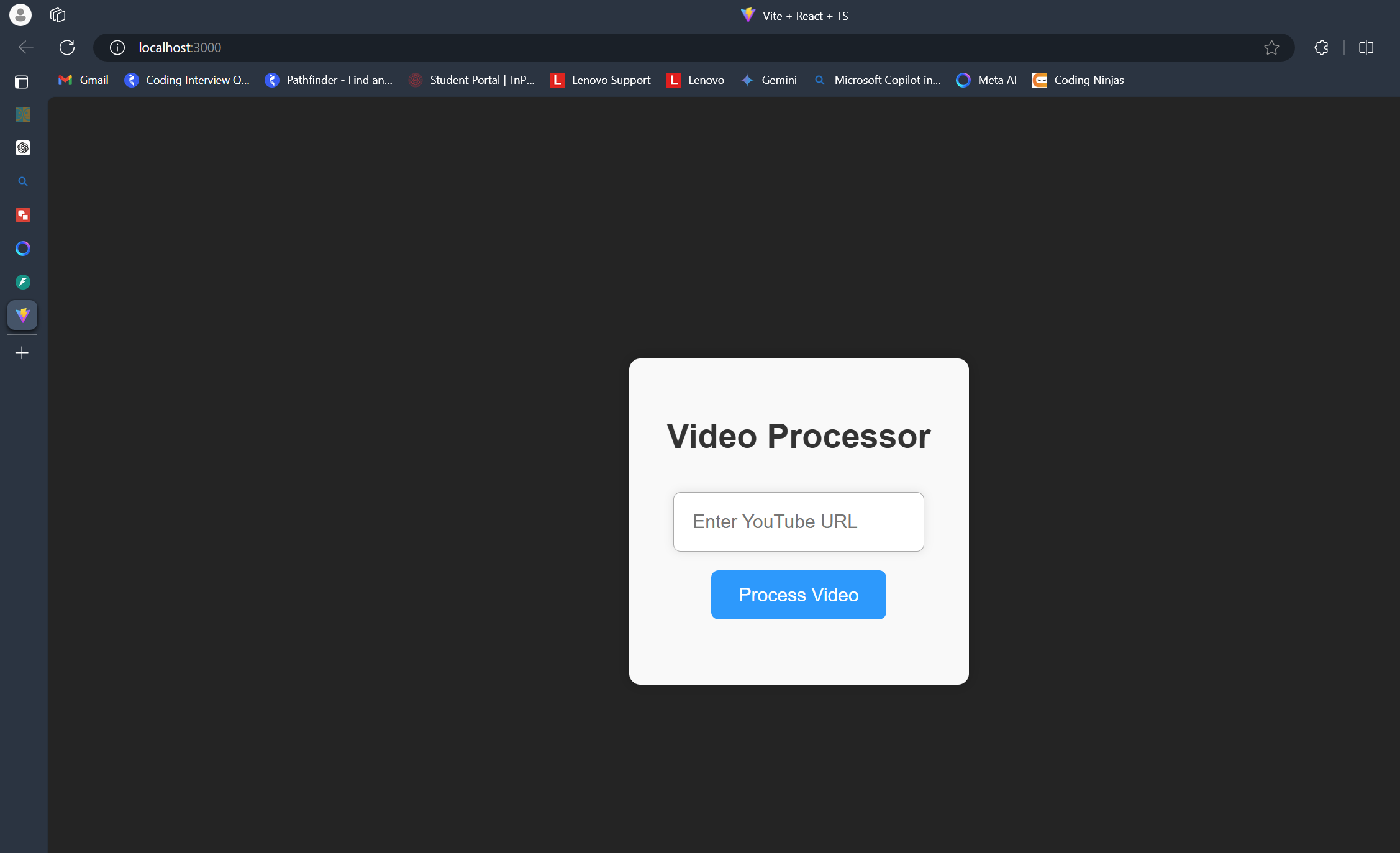
I have reduced endpoints as most of them were ‘GET’ request. Folder ‘fastapi’ will all endpoints

But Folder ‘frontend’ I have reduce endpoint because it was easier work will 3 endpoint.



**Frontend (frontend/)**

* src/VideoProcessor.tsx: React component for video processing.
* Dockerfile: Docker configuration for the React frontend.
* package.json: Node.js dependencies for the frontend.



**Methodology**

**Architecture**

The system follows a **microservices architecture**, with distinct components for each processing step. Each component has a specific responsibility, promoting separation of concerns and scalability.

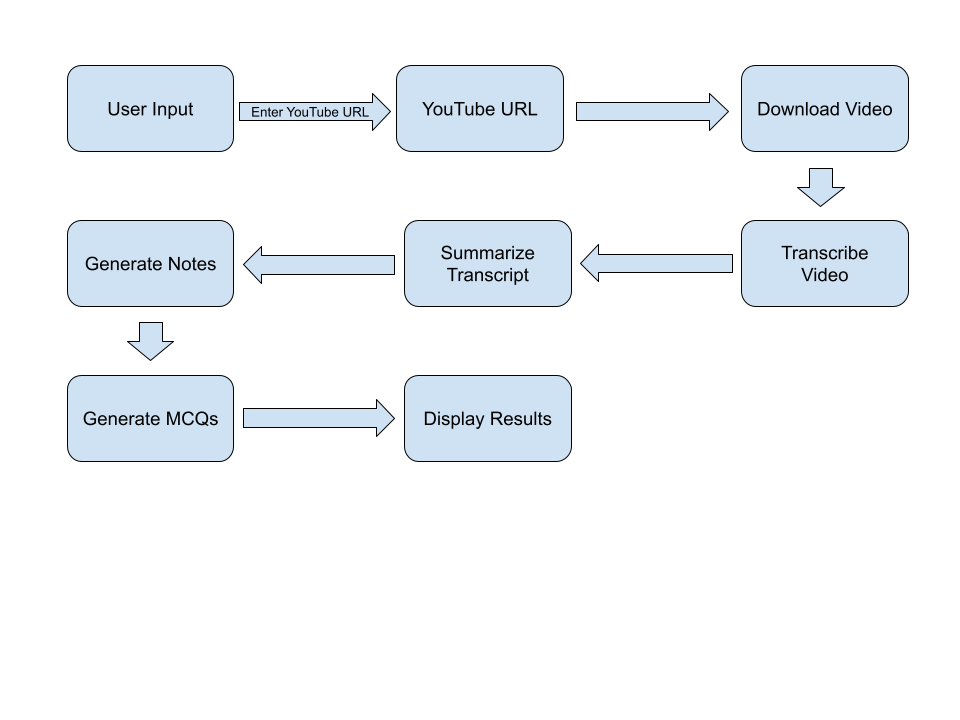
**Backend Workflow**

1. **Download Video**: Utilizes pytube to download the video and requests with BeautifulSoup to fetch video details.
2. **Transcribe Video**: Uses moviepy to extract audio and speech\_recognition to transcribe it.
3. **Summarize Transcript**: Utilizes the transformers library with the BART model to summarize the text.
4. **Generate Notes**: Summarizes the text in smaller chunks using the BART model.
5. **Generate MCQs**: Uses NLTK to create MCQs from the transcript text.

**Frontend Workflow**

1. **User Input**: Users enter a YouTube URL.
2. **Trigger Processing**: The URL is sent to the backend via a POST request.
3. **Display Results**: The frontend fetches and displays the processed files (transcript, summary, notes, and MCQs).

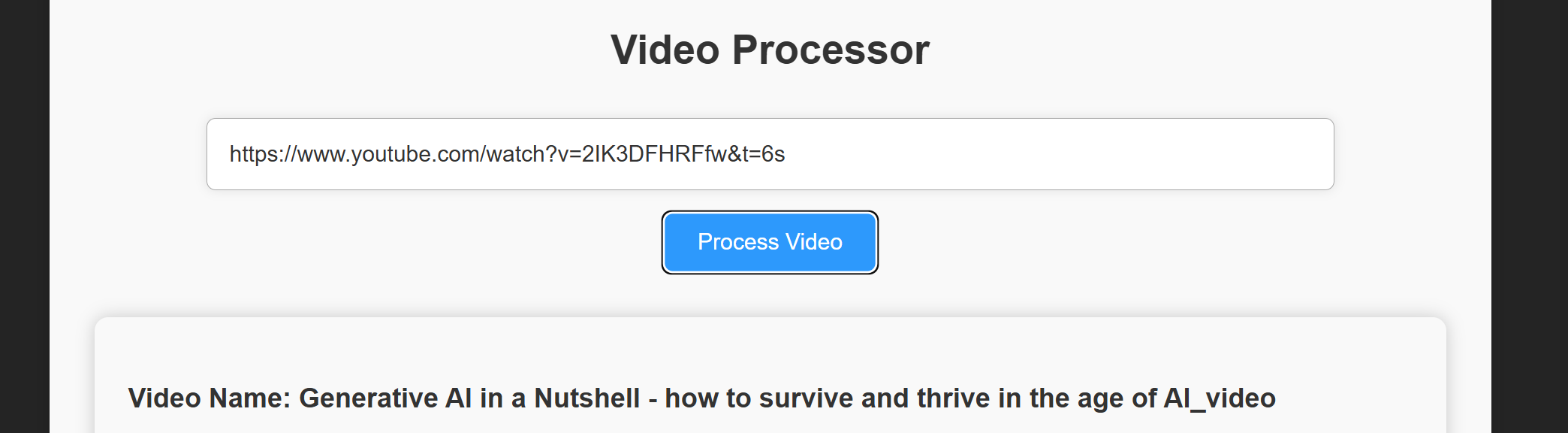
The frontend uses the axios library to fetch data from the backend service. Here is an example of how the axios library is used in the VideoProcessor component



**Detailed Explanation of Components**

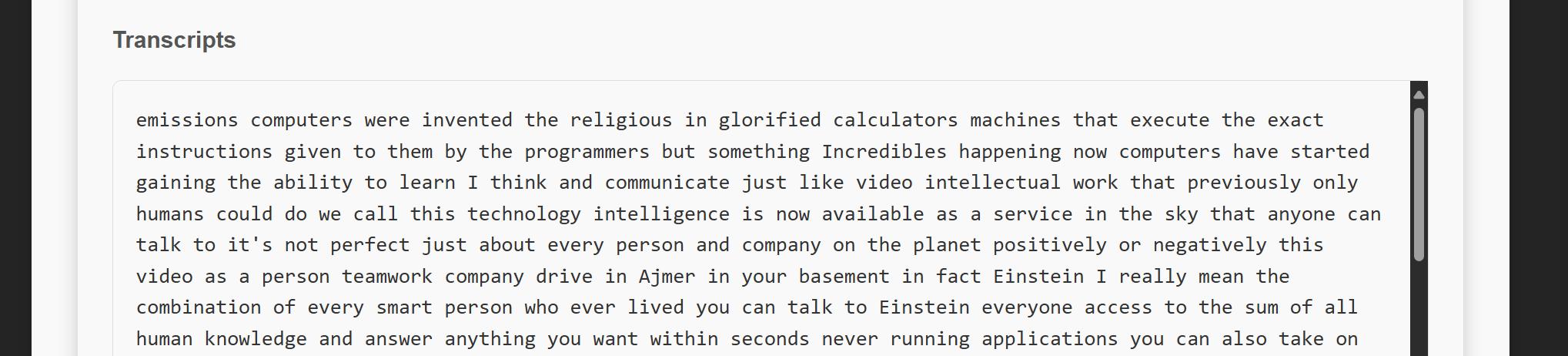
**YouTubeDownloader**

* The YouTubeDownloader component is responsible for handling the downloading of YouTube videos and extracting relevant video details. It uses the pytube library to download videos and requests with BeautifulSoup to fetch additional metadata such as title, description, views, and likes. This component ensures that videos are fetched securely and efficiently for further processing.



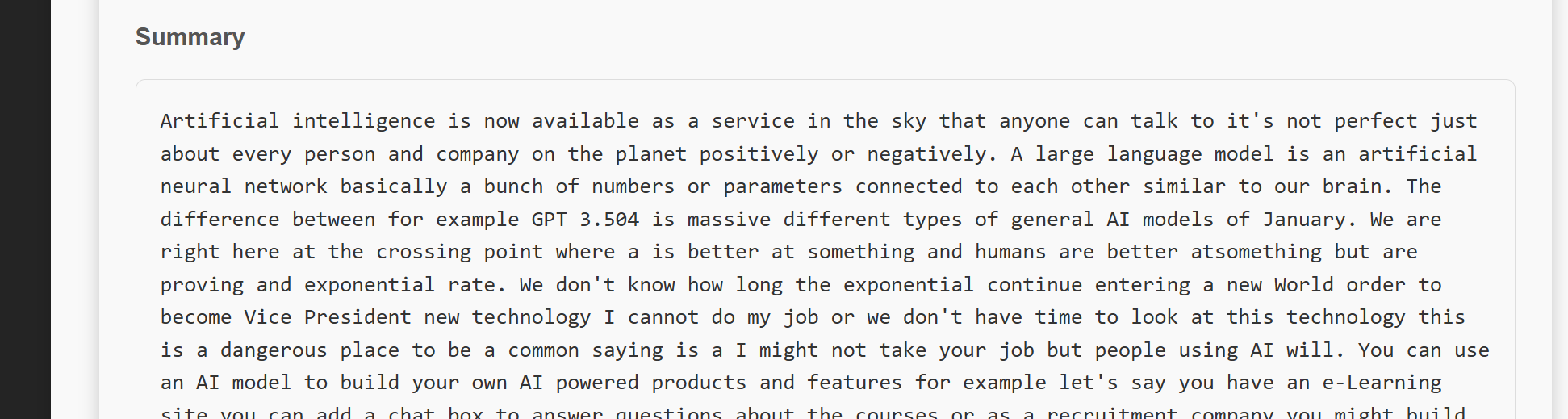
**VideoTranscriber**

* The VideoTranscriber component extracts audio from downloaded videos and transcribes this audio into text format. It utilizes the moviepy library to extract the audio stream from video files and speech\_recognition to perform the transcription. This component is crucial for converting spoken content from videos into textual data, which is essential for subsequent analysis and processing.



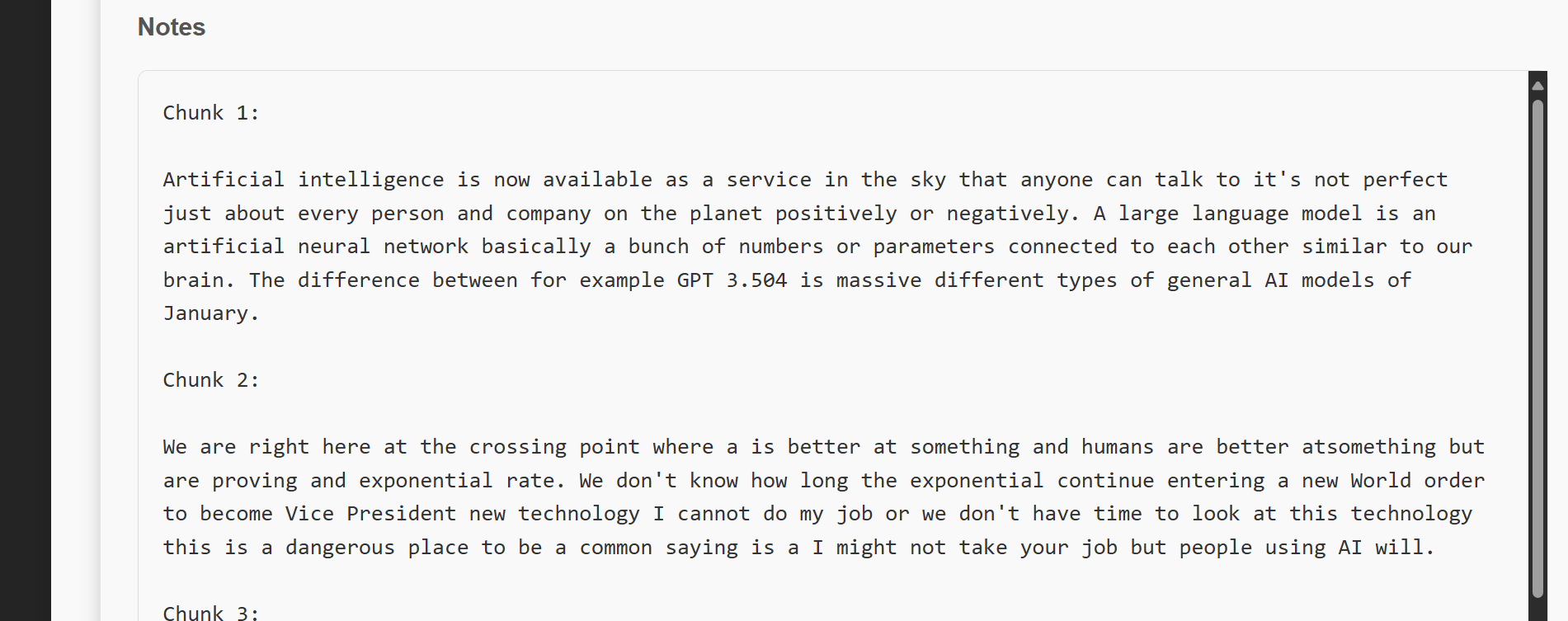
**TextSummarizer**

* The TextSummarizer component employs transformer-based models, such as BART from the transformers library, to summarize the transcribed text. It condenses lengthy transcripts into concise summaries, enhancing readability and digestibility of the content. This summarization process helps in generating high-level overviews of the video content, facilitating quicker understanding and review.



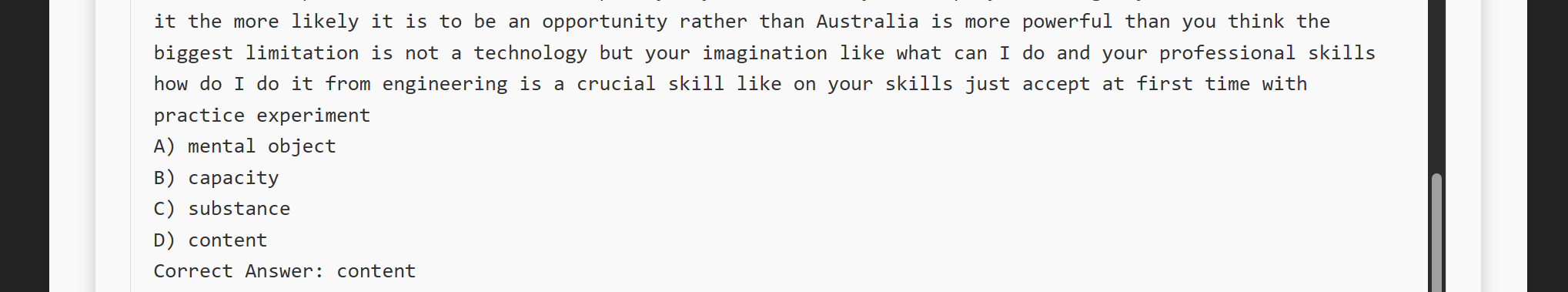
**NoteGenerator**

* The NoteGenerator component builds upon the text summarization process by generating detailed notes from the summarized text. It segments the summarized content into smaller chunks or key points, making it easier for users to review and retain important information. This component is particularly useful for educational purposes, aiding in the creation of structured study materials based on video content.



**MCQGenerator**

* The MCQGenerator component creates multiple-choice questions (MCQs) based on the transcribed and summarized text. It uses natural language processing (NLP) techniques, including part-of-speech tagging and synonym extraction from the NLTK library, to formulate questions and identify potential answer options. This component enhances interactivity by generating quizzes or assessments from video content, promoting active learning and engagement.



**React Frontend**

* Allows users to input a YouTube URL and view the processed files.

**VideoProcessor.tsx**

* The React frontend component, VideoProcessor.tsx, serves as the user interface for interacting with the automated video processing system. It allows users to input a YouTube URL, triggers the backend processing through API requests, and displays the processed results. This frontend ensures a seamless user experience by presenting the generated transcripts, summaries, notes, and MCQs in a structured and accessible format.

**Integration and Workflow**

Each component in the system operates independently but collaboratively to streamline the video processing workflow:

* **Input**: Users provide a YouTube URL via the React frontend.
* **Processing**: The backend components (YouTubeDownloader, VideoTranscriber, TextSummarizer, NoteGenerator, MCQGenerator) sequentially process the video content, from downloading and transcribing to summarizing and generating educational resources.
* **Output**: Processed files (transcript, summary, notes, MCQs) are then presented back to the user through the React frontend, enabling easy access and utilization of the generated content.

**Backend Techniques**

**1. Video Downloading**

**Technique:** Web scraping and downloading using a YouTube downloader library.

**Description:** The YouTubeDownloader class utilizes a library to download the video and audio from a given YouTube URL. This is achieved by:

* Extracting the video URL.
* Downloading the video and audio streams.
* Saving the downloaded content to a specified directory.

python

Copy code

class YouTubeDownloader:

def download\_video\_and\_audio(self, url: str) -> str:

# Implementation details to download video

**2. Video Transcription**

**Technique:** Automatic Speech Recognition (ASR).

**Description:** The VideoTranscriber class uses ASR techniques to convert the audio from the video into text. This process involves:

* Splitting the audio into manageable chunks.
* Using a transcription model to convert each audio chunk into text.
* Combining the text from all chunks into a single transcript.

python

Copy code

class VideoTranscriber:

def transcribe\_video\_in\_chunks(self, video\_path: str):

# Implementation details to transcribe video audio

**3. Text Summarization**

**Technique:** Natural Language Processing (NLP) summarization models.

**Description:** The TextSummarizer class uses an NLP model to generate a concise summary of the transcribed text. This involves:

* Loading the transcript from a text file.
* Using a pre-trained summarization model to generate a summary.
* Saving the summary to a text file.

python

Copy code

class TextSummarizer:

def summarize\_text\_file(self, transcript\_path: str):

# Implementation details to summarize the transcribed text

**4. Note Generation**

**Technique:** NLP text chunking and summarization.

**Description:** The NoteGenerator class creates notes by chunking the transcript into smaller parts and summarizing each part. This process includes:

* Splitting the text into chunks.
* Generating a summary for each chunk.
* Combining the summaries into a single note file.

python

Copy code

class NoteGenerator:

def generate\_notes\_from\_file(self, transcript\_path: str):

# Implementation details to generate notes from the transcript

**5. MCQ Generation**

**Technique:** NLP for tokenization and synonym extraction.

**Description:** The MCQGenerator class creates multiple-choice questions by:

* Tokenizing the transcript into sentences.
* Identifying key nouns in each sentence.
* Generating distractors (incorrect options) using synonyms.
* Structuring the questions and answers in a readable format.

python

Copy code

class MCQGenerator:

def generate\_and\_save\_mcqs(self, csv\_filename: str, text\_filename: str):

# Implementation details to generate MCQs from the transcript

**FastAPI Endpoints**

**Technique:** RESTful API design.

**Description:** The FastAPI framework is used to create endpoints that handle HTTP requests and interact with the various processing components. Key endpoints include:

* POST /process to initiate the video processing workflow.
* GET /file/{file\_path} to retrieve generated files.
* GET /delete\_folder to clean up temporary files.

python

Copy code

@app.post("/process")

async def process(request: URLRequest):

# Implementation details to handle video processing workflow

@app.get("/file/{file\_path:path}")

async def get\_file(file\_path: str):

# Implementation details to retrieve generated files

@app.get('/delete\_folder')

async def delete\_folder():

# Implementation details to delete temporary processing folders

**Frontend Techniques**

**1. Handling User Input**

**Technique:** React state management.

**Description:** React state management is used to handle user input for the YouTube URL. The state is updated whenever the user types in the input field.

javascript

Copy code

const [url, setUrl] = useState("");

const handleUrlChange = (e: React.ChangeEvent<HTMLInputElement>) => {

setUrl(e.target.value);

};

**2. Making HTTP Requests**

**Technique:** Axios for HTTP requests.

**Description:** Axios is used to send HTTP requests to the FastAPI backend. This includes:

* Sending a POST request to initiate video processing.
* Fetching the contents of generated files.

javascript

Copy code

const handleProcess = async () => {

try {

const res = await axios.post<ProcessResponse>("http://localhost:8000/process", { url\_P: url });

setResponse(res.data);

await fetchFileContents(res.data);

} catch (err) {

setError("An error occurred while processing the video.");

}

};

**3. Displaying Results**

**Technique:** Conditional rendering in React.

**Description:** The results of the processing (transcripts, summaries, notes, MCQs) are displayed using conditional rendering based on the state of the response.

javascript

Copy code

{response && (

<div className="output-container">

<p className="filename">Video Name: {response.video\_filename}</p>

<h2>Generated Files:</h2>

<div className="file-content">

<h3>Transcripts</h3>

<pre className="preformatted">{fileContents["transcripts\_path"]}</pre>

</div>

<div className="file-content">

<h3>Summary</h3>

<pre className="preformatted">{fileContents["summary\_path"]}</pre>

</div>

<div className="file-content">

<h3>Notes</h3>

<pre className="preformatted">{fileContents["notes\_path"]}</pre>

</div>

<div className="file-content">

<h3>MCQs (Text)</h3>

<pre className="preformatted">{fileContents["mcqs\_txt\_path"]}</pre>

</div>

</div>

)}

**Docker Configuration**

**Technique:** Containerization with Docker.

**Description:** Dockerfiles are used to define the environment for the frontend and backend services. This includes specifying the base image, working directory, copying files, installing dependencies, and setting the command to run the application.

**Backend Dockerfile**

FROM python:3.10-slim

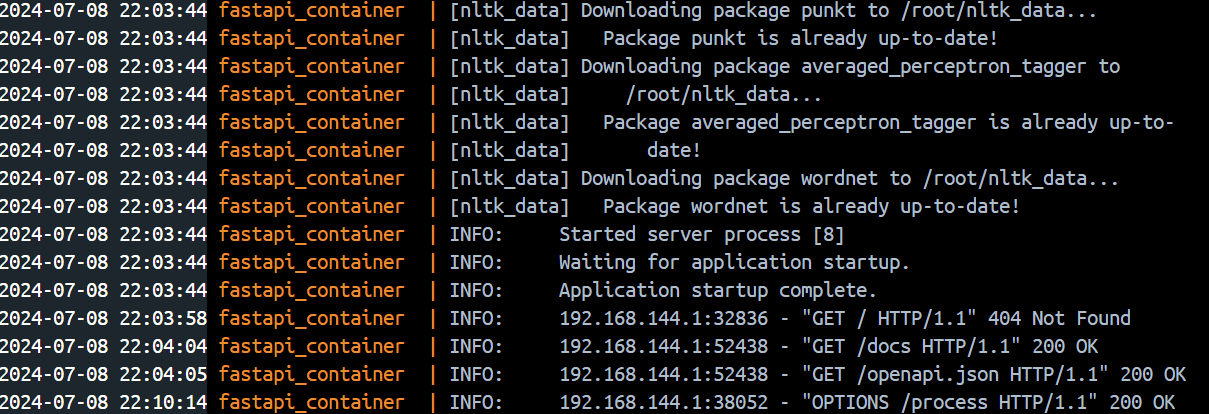
WORKDIR /usr/app

COPY . /usr/app/

RUN pip install fastapi uvicorn -r /usr/app/requirements.txt

EXPOSE 8000

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]



**Frontend Dockerfile**

FROM node:18

WORKDIR /app

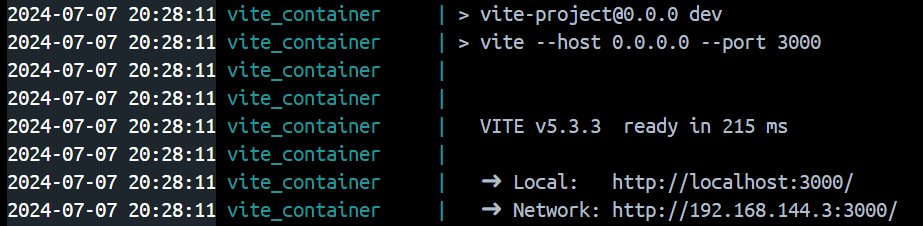
COPY package.json .

RUN npm install

COPY . .

EXPOSE 3000

CMD ["npm", "run", "dev", "--", "--host", "0.0.0.0", "--port", "3000"]



**Docker Compose**

**Technique:** Multi-container orchestration.

**Description:** Docker Compose is used to define and manage multi-container Docker applications. The docker-compose.yml file specifies the services, their build context, ports, volumes, and dependencies.

services:

frontend:

build: ./

container\_name: vite\_container

expose:

- 3000

ports:

- "3000:3000"

volumes:

- /app/node\_modules

- ./:/app

depends\_on:

- backend\_service

backend\_service:

build: ./server

container\_name: fastapi\_container

expose:

- 8000

ports:

- "8000:8000"

volumes:

- /usr/app/\_\_pycache\_\_

- ./server:/usr/app

**Running the Project**

1. **Backend:**
   * Navigate to the server directory and build the Docker image.
   * Run the following commands:
   * docker build -t fastapi-backend .
   * docker run -p 8000:8000 fastapi-backend
2. **Frontend:**
   * Navigate to the frontend directory and build the Docker image.
   * Run the following commands:
   * docker build -t react-frontend .
   * docker run -p 3000:3000 react-frontend
3. **Using Docker Compose:**
   * From the root directory, run:
   * docker-compose up --build

**Key Points and Best Practices:**

1. **Separation of Concerns:**
   * Each component (downloader, transcriber, summarizer, note generator, MCQ generator) has a single responsibility, making the codebase more modular and maintainable.
2. **Error Handling:**
   * Comprehensive error handling is implemented to ensure the system’s robustness.
3. **Scalability:**
   * The microservices architecture allows individual components to be scaled independently.
4. **Asynchronous Processing:**
   * The FastAPI backend is designed to handle asynchronous processing, improving performance and responsiveness.
5. **User Experience:**
   * The React frontend provides a simple and intuitive interface for users to interact with the system.

This setup provides a fully automated workflow for processing YouTube videos, from downloading and transcribing to summarizing and generating educational content.

**Conclusion**

The Video Processing Application effectively demonstrates the integration of various modern web technologies and NLP techniques to provide a comprehensive solution for processing and analyzing YouTube videos. By leveraging FastAPI for the backend, React for the frontend, and Docker for containerization, the application ensures scalability, maintainability, and ease of deployment.

**Future Improvements**

**1. Enhanced Error Handling**

Improve error handling mechanisms in both the backend and frontend to provide more detailed and user-friendly error messages.

**2. Support for Multiple Languages**

Extend the ASR and NLP models to support multiple languages, enabling a broader range of users to utilize the application.

**3. Real-time Processing**

Implement real-time video and audio processing to provide instantaneous feedback and results to users.

**4. Advanced NLP Techniques**

Incorporate more advanced NLP techniques such as sentiment analysis, entity recognition, and topic modeling to provide deeper insights into the video content.

**5. User Authentication**

Add user authentication and authorization to restrict access to the application and provide personalized user experiences.

**6. Frontend Enhancements**

Improve the frontend UI/UX with better design, interactive elements, and real-time updates to enhance the user experience.

By implementing these improvements, the Video Processing Application can become a more robust, versatile, and user-friendly tool, catering to a wider audience and providing more valuable insights from video content.