TDGPT – GitHub file extraction

**Aim:**

To build a platform that supports Retrieval-Augmented Generation (RAG)-based application usage by extracting, processing, and embedding documents (e.g., .txt, .md, .py, etc.) from uploaded file, web URL or GitHub repositories link.

**Document content:**

Extracting files from GitHub repo from the given link and embedding them based on the file format

**Input:** GitHub repo URL

**Output:** Saved in “output/{owner}/{repo}/” with original filenames.

**Methods Overview**

Method 1: Using GitHub API

Method 2: Cloning the Repository via Git

Method 3: Downloading and Extracting ZIP Archive

Method 4: GitHub GraphQL API

Method 5: Raw Content Download via HTTP

**Method – 1: Using GitHub API**

**Description**

Utilize GitHub's API to programmatically access repository contents, allowing for selective retrieval of files based on specific criteria such as file extensions.

**Use Case**

Ideal for applications requiring targeted access to certain file types without the overhead of downloading the entire repository.

**Packages:**

* requests – To make HTTP requests to GitHub API.
* base64 – To decode base64-encoded file content.

**Core logic:**

*async def* ***fetch\_contents****(client, owner, repo, path=""):*

*url = f"{GITHUB\_API\_BASE}/repos/{owner}/{repo}/contents/{path}"*

*response = await client.get(url)*

*if response.status\_code != 200:*

*raise ValueError(f"GitHub API error: {response.status\_code} - {response.text}")*

*data = response.json()*

*files = []*

*if isinstance(data, dict) and data.get("type") == "file":*

*files.append(data["path"])*

*else:*

*for item in data:*

*if item["type"] == "file":*

*files.append(item["path"])*

*elif item["type"] == "dir":*

*nested\_files = await fetch\_contents(client, owner, repo, item["path"])*

*files.extend(nested\_files)*

*return files*

*async def* ***download\_files****(client, owner, repo, file\_paths):*

*saved\_files = []*

*for path in file\_paths:*

*if path.endswith(".txt"): #required files are copied using its extention*

*url = f"{GITHUB\_API\_BASE}/repos/{owner}/{repo}/contents/{path}"*

*response = await client.get(url)*

*if response.status\_code == 200:*

*data = response.json()*

*if data.get("encoding") == "base64":*

*import base64*

*content = base64.b64decode(data["content"]).decode("utf-8")*

*local\_path = os.path.join(f"output/{owner}/{repo}", os.path.basename(path))*

*with open(local\_path, "w", encoding="utf-8") as f:*

*f.write(content)*

*saved\_files.append(local\_path)*

*return saved\_files*

**Code Explanation**

The `fetch\_contents` function recursively traverses the repository to list all file paths. The `download\_files` function filters for `.txt` files, downloads them, decodes base64 content, and saves them locally.

**Pros**

* Efficient retrieval of specific files without downloading the entire repository.
* Reduces bandwidth and storage usage.
* Suitable for both public and private repositories (with authentication).

**Cons**

* Subject to GitHub API rate limits (60 req/hr).
* Requires handling of API responses and potential errors.
* Not suitable for retrieving large binary files due to size limitations.

**Performance**

High efficiency for selective file access.

**Limitations**

* Cannot retrieve files managed by Git LFS.
* Does not provide repository history or metadata beyond file contents.

**Method – 2: Git Clone**

**Description**

Clones the entire Git repository locally, enabling access to all files and full commit history. Use Python to filter for required file types.

**Use Case**

Best when full access to the repository (including history, all file types, and branches) is required.

**Packages:**

* subprocess – To execute git clone commands from Python.

**Core logic:**

*import subprocess*

*def clone\_repo(git\_url, clone\_dir):*

*try:*

*subprocess.run(["git", "clone", git\_url, clone\_dir], check=True)*

*print("Repository cloned successfully.")*

*except subprocess.CalledProcessError as e:*

*print("Error during cloning:", e)*

**Pros**

* Full access to all branches and history
* No API rate limits
* Works offline after cloning

**Cons**

* Clones the entire repository (high disk usage and time for large repos)
* Requires Git to be installed
* Slower compared to selective methods

**Performance:** Medium

**Limitations**

* Not selective: entire repository is downloaded
* Requires post-processing to filter files

**Method – 3: Download ZIP**

**Description**

Downloads the default branch of a repository as a ZIP archive and extracts the needed files locally.

**Use Case**

Useful for a quick, read-only snapshot of a public repo.

**Packages:**

* zipfile – To extract contents from the ZIP file.
* io – To handle in-memory binary streams.

**Core logic:**

*import requests, zipfile, io, os*

*def download\_and\_extract\_txt(zip\_url, output\_dir="output"):*

*r = requests.get(zip\_url)*

*z = zipfile.ZipFile(io.BytesIO(r.content))*

*z.extractall(output\_dir)*

*txt\_files = []*

*for file in z.namelist():*

*if file.endswith(".txt"):*

*txt\_files.append(os.path.join(output\_dir, file))*

*return txt\_files*

**Pros**

* Fast and simple
* No authentication or Git installation needed
* Easy extraction of .txt, .md, or similar files

**Cons**

* Only works for the default branch
* No access to commit history or private repos

**Performance:** Fast

**Limitations**

* Read-only archive
* Cannot access other branches or commit history

**Method – 4: GitHub GraphQL API**

**Description**

Uses GitHub’s GraphQL API to query specific files or folders in a single, flexible request.

**Use Case**

Ideal for advanced querying and hierarchical filtering

**Pros**

* Highly flexible and efficient
* Fetches only what’s needed
* Reduces the number of requests

**Cons**

* Requires GraphQL knowledge
* Setup is more complex
* Authentication is mandatory

**Performance:** Medium

**Limitations**

* Complex structure for large repos

**Method – 5: Raw GitHub URLs**

**Description**

Directly fetches a known file using the raw content URL from GitHub.

**Use Case**

Perfect for quick access when the file path is known in advance.

**Core logic:**

*import requests*

*def download\_raw\_txt\_file(url, output\_path):*

*r = requests.get(url)*

*if r.status\_code == 200:*

*with open(output\_path, "w", encoding="utf-8") as f:*

*f.write(r.text)*

**Pros**

* Extremely fast and easy
* No setup or API key required

**Cons**

* Requires exact file path
* Cannot discover or list files

**Performance:** Fast

**Limitations**

* No browsing capability
* Only works when file paths are already known

**Comparison:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Method** | **Use Case** | **Auth Needed** | **Performance** | **Private Repos** | **Filters Available** | **Best For** |
| GitHub API | Selective file retrieval | Yes | High | Yes | Yes | Targeted file access |
| Git Clone | Full repository access | Yes/No | Medium | Yes | No | Comprehensive repository access |
| ZIP Archive Download | Quick snapshot of repository contents | No | High | No | Post-download | Simple, public repositories |
| GitHub GraphQL API | Complex queries and hierarchical access | Yes | Medium | Yes | Yes | Advanced querying needs |
| Raw GitHub File URLs | Direct access to known file paths | No | High | No | No | Quick access to specific files |

**Suggested Method:**

**Method 1 (GitHub API)**  
It is optimal for:

* Controlled, filtered file access
* Good balance of speed and flexibility
* Works for public & private repos