

****

**Midterm Project Report**

**Advanced Computer Programming**

**Student Name : Hanifi Abrar Setiawan**

**Student ID : 112021224**

**Teacher : DINH-TRUNG VU**

**2024-04**

# Introduction

## Github

1. **Personal Github Account**: [*https://github.com/HanifiSetiawan*](https://github.com/HanifiSetiawan)
2. **Group Project Repository**: [*https://github.com/HanifiSetiawan/ACP-Group-06*](https://github.com/HanifiSetiawan/ACP-Group-06)

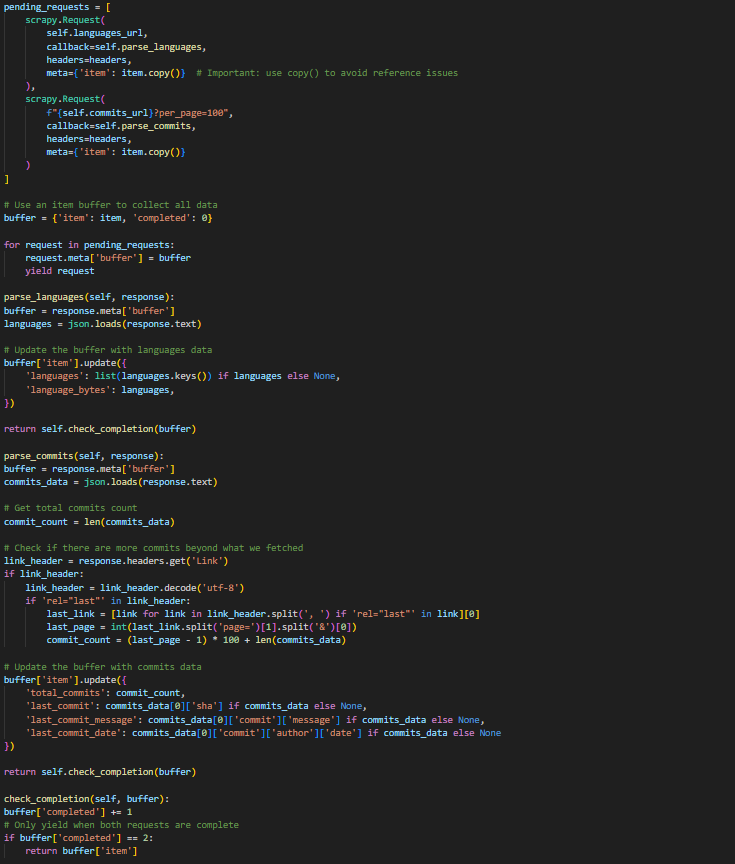
## Overview

The code utilizes several advanced Python features and libraries to scrape GitHub repository data. The ‘scrapy’ library forms the backbone, enabling efficient web crawling and data extraction. The spider class ‘GithubRepoSpider’ leverages asynchronous requests to fetch repository metadata, languages, and commit history from GitHub's API. Advanced features include URL parsing with ‘urllib.parse’, dynamic request headers for authentication (using ‘GITHUB\_TOKEN’), and JSON handling for API responses. The spider employs a buffering system to manage concurrent requests, ensuring data from multiple endpoints is aggregated before yielding the final result. This approach demonstrates sophisticated use of meta-data passing and callback chaining, which are key concepts in Scrapy for handling complex scraping workflows.

The output provides a comprehensive snapshot of the repository, including details like stars, forks, commit history, and programming languages used. The result highlights the spider's ability to extract structured data efficiently, with fields such as ‘last\_commit\_message’ and ‘language\_bytes’ offering deeper insights. The XML format organizes the data hierarchically, making it easy to parse and analyze. This implementation showcases how advanced scraping techniques can transform raw API responses into meaningful, structured information for further use.

# Implementation

## Class GithubRepoSpider

A Scrapy spider designed to scrape GitHub repository metadata, languages, and commit history via GitHub’s REST API. Handles initialization, request handling, and data processing.

### Fields

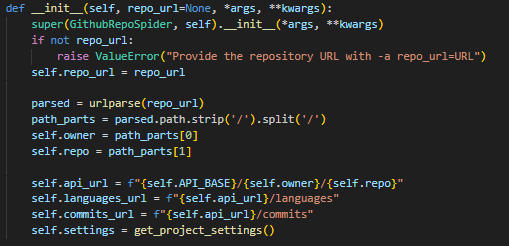


* Identifies the spider in Scrapy commands.

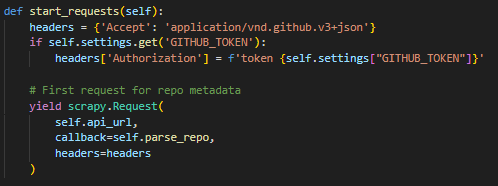


* Base URL for GitHub’s repository API.

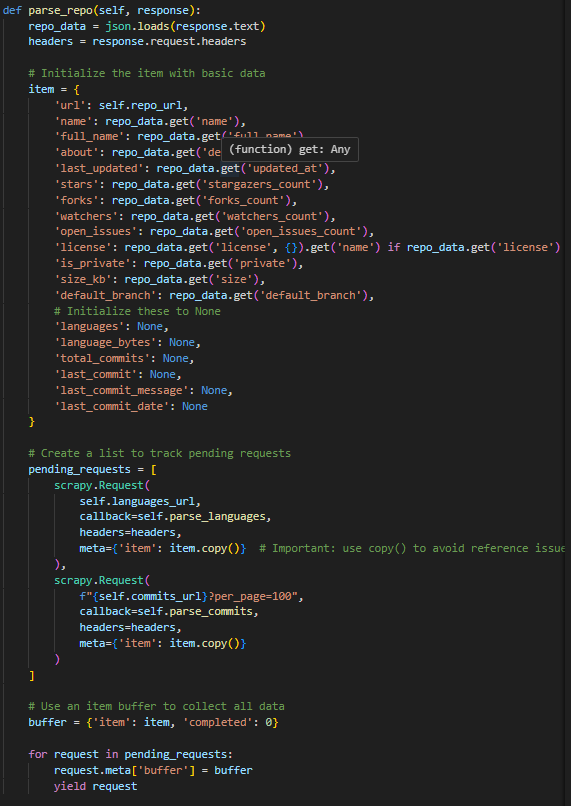
### Methods



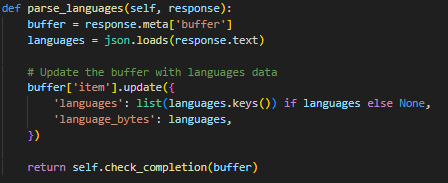
* Initializes the spider, parses the repo URL, and sets up API endpoints.
* Raises ValueError if repo\_url is missing.



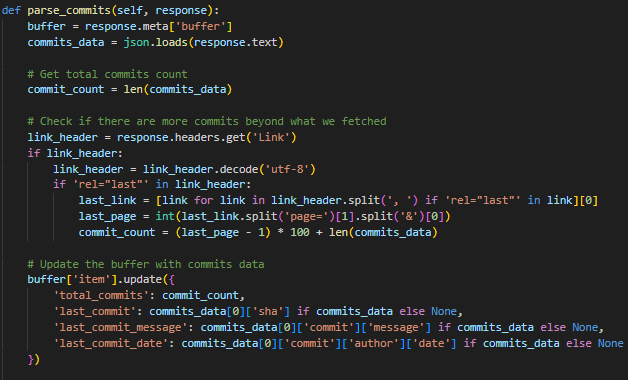
* Generates the first API request for repository metadata.
* Adds authentication headers if GITHUB\_TOKEN is provided in settings.



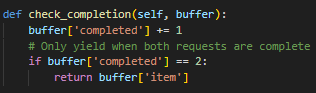
* Processes the repository metadata (e.g., stars, forks, description).
* Triggers parallel requests for **languages** and **commits** data.
* Uses a buffer to track pending requests and merge results.



* Extracts programming languages used in the repo (e.g., Python: 1000 bytes).
* Updates the buffer with language data.



* Fetches commit history (total count, last commit message/date).
* Handles pagination via GitHub’s Link header to count all commits.



* Tracks completed requests (languages + commits).
* Yields the final merged data only after both requests finish.

### Workflow

* + 1. Starts with start\_requests → Fetches repo metadata (parse\_repo).
    2. Concurrently requests languages + commits → Merges results via buffer.
    3. Outputs structured data (as seen in github\_repo.xml).

## Class XmlExportPipeline

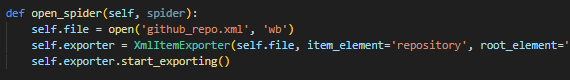


A Scrapy Item Pipeline that exports scraped data to an XML file. It uses Scrapy’s built-in ‘XmlItemExporter’ to serialize items into a structured XML format.

### Methods



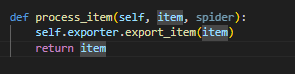
* Constructor initializes file to None.
* Prepares the pipeline but defers file operations until the spider starts.



* Called when the spider starts.
* Opens github\_repo.xml in binary write mode ('wb').
* Initializes the XmlItemExporter with XML structure settings and begins exporting.

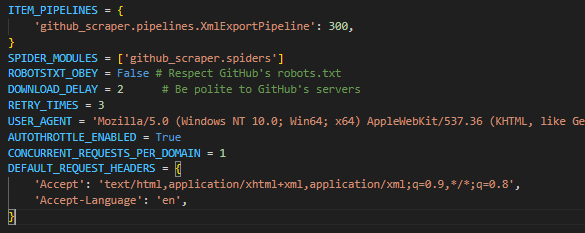


* Called when the spider finishes.
* Finalizes the XML export and closes the file.



* Processes each scraped item (passed from the spider).
* Uses the exporter to write the item as an XML repository element.
* Returns the item to allow chaining with other pipelines.

## Key Settings



1. **ITEM\_PIPELINES**

* 'github\_scraper.pipelines.XmlExportPipeline': 300:  
  Activates the XmlExportPipeline to export scraped data to XML. The value 300 sets its priority (lower = earlier processing).

1. **SPIDER\_MODULES**

* ['github\_scraper.spiders']:  
  Specifies the Python module where Scrapy should look for spiders.

1. **Politeness & Throttling**

* ROBOTSTXT\_OBEY = False:  
  Disables adherence to robots.txt (GitHub’s API allows scraping, but this should be used cautiously).
* DOWNLOAD\_DELAY = 2:  
  Adds a 2-second delay between requests to avoid overloading servers.
* CONCURRENT\_REQUESTS\_PER\_DOMAIN = 1:  
  Limits concurrent requests to GitHub to 1 at a time.
* AUTOTHROTTLE\_ENABLED = True:  
  Automatically adjusts request rates based on server response times.

1. **Retry & Reliability**

* RETRY\_TIMES = 3:  
  Retries failed requests up to 3 times.

1. **Request Headers**

* USER\_AGENT: Mimics a browser (Chrome/91.0) to avoid being blocked.
* DEFAULT\_REQUEST\_HEADERS: Sets default HTTP headers for requests (e.g., Accept and Accept-Language).

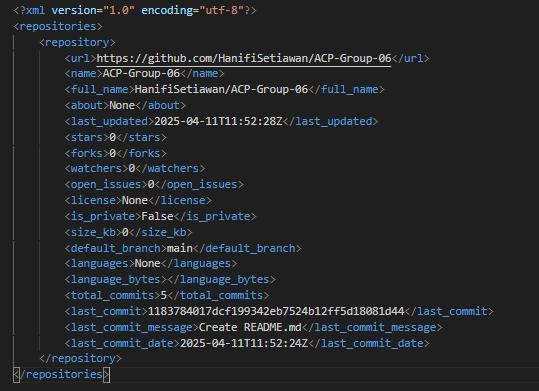
2.4 How to run it



Using basic scrapy command and defineing the ‘repo\_url’

# Results

## Result

. 

.

# Conclusions

The GitHub repository scraper is a well-structured, efficient, and ethical web scraping tool built with Scrapy that extracts key repository data—such as metadata, languages, and commit history—from GitHub’s API and exports it to a structured XML file. The project showcases modular design, separating data extraction (spider) from data export (pipeline), while adhering to best practices like request throttling, retries, and polite delays to avoid overloading servers. With features like pagination handling, dynamic authentication, and concurrent request management, the tool is both robust and scalable, ideal for tracking repository metrics, analyzing language trends, or generating audit reports. Minor enhancements like error handling for rate limits or multi-repo support could further improve its utility, but the current implementation effectively balances functionality, performance, and compliance with GitHub’s guidelines.