Midterm Project Report

Advanced Computer Programming

Student Name : Wirandito Sarwono

Student ID : 113021198

Teacher : DINH-TRUNG VU

2024-04

Chapter 1 Introduction

1.1 Github

1) **Personal Github Account**: 113021198, DitoNewJeans
2) **Group Project Repository**: https://github.com/113021198/ACP

1.2 Overview

The Scrapy framework, regular expressions, and JSON and XML handling were used to create a web crawler that extracts GitHub repository information. The spider navigates to a user's repository page (specifically https://github.com/DitoNewJeans?tab=repositories) and extracts detailed information about each repository, including URL, description, last updated date, programming languages used, and number of commits. The program handles various GitHub page layouts through multiple CSS selectors and gracefully manages empty repositories.

Implementation

1.1 Class 1: GithubSpider

The main class that extends Scrapy's Spider class to crawl GitHub

1.1.1 Fields

- name: 'github' - the identifier for the spider
- allowed_domains: ['github.com'] - only for the github domains
- start-urls - Initial URL to start the crawl ('https://github.com/DitoNewJeans?tab=repositories')
- custom_settings -
  - 'ROBOTSTXT_OBEY': False - Ignores robots.txt restrictions
  - 'USER_AGENT': Sets a browser user agent to avoid being blocked
  - 'LOG_LEVEL': 'DEBUG' - Sets detailed logging for debugging

1.1.2 Methods

- **start_requests**

  Initiates the crawling process by sending requests to the starting URLs.

```
def start_requests(self):
    self.logger.info(f"Starting crawl with URL:
{self.start_urls[0]}")
    for url in self.start_urls:
        yield scrapy.Request(url=url, callback=self.parse)
```

- **parse**

  Processes the response from the initial request, extracting repository information and following links to repository detail pages.

  - Uses multiple CSS selector strategies to handle different GitHub page layouts
  - Extracts basic repository information (name, URL, description, last updated date)
  - Determines if repositories are empty
  - Request each repository's detailed page for further processing

    - Handles pagination by following "next page" links
- parse_repository_details

  Processes the individual repository pages to extract additional information

    - Extracts programming languages used in the repository
    - Extracts the number of commits
    - Handles empty repositories appropriately
    - Standardizes the output format for all repositories

1.1.3 Functions

The code doesn't include standalone functions outside of class methods, but it makes use of Python's built-in functions and Scrapy's framework functions.

1.2 Method/Function 1: parse

This method is the main parser for the repository list page:

```
def parse(self, response):
    # Debug information
    self.logger.info(f"Response URL: {response.url}")
    self.logger.info(f"Response status: {response.status}")

    repos_found = False

    #sel 1
    repos = response.css('div[id="user-repositories-list"] li')
    if repos:
        self.logger.info(f"Found {len(repos)} repositories with
selector 1")
        repos_found = True

        for repo in repos:
            name_element = repo.css('h3 a::text').get()
            if name_element:
                name = name_element.strip()
                url = response.urljoin(repo.css('h3
a::attr(href)').get())
                about =
repo.css('p.color-fg-muted.mb-0::text').get()
                about = about.strip() if about else "No
description"
```

```
                last_updated =
repo.css('relative-time::attr(datetime)').get()
                if last_updated:
                    last_updated =
datetime.datetime.fromisoformat(last_updated.replace('Z',
'+00:00')).strftime('%Y-%m-%d')
                else:
                    last_updated = "Not specified"

                is_empty = 'This repository is empty.' in
repo.get()

                self.logger.info(f"Found repository: {name}")

                #page details
                repo_data = {
                    'name': name,
                    'url': url,
                    'about': about,
                    'last_updated': last_updated,
                    'is_empty': is_empty
                }

                yield scrapy.Request(
                    url=url,
                    callback=self.parse_repository_details,
                    meta={'repo_data': repo_data}
                )
```

The method is designed with the following key features:

- Comprehensive logging for debugging
- Multiple CSS selector strategies to handle different GitHub page layouts
- Graceful handling of missing information with default values
- Proper date formatting for timestamp information
- Metadata passing between requests using Scrapy's meta parameter

1.3 Method/Function 2: parse_repository_details

```
def parse_repository_details(self, response):
    repo_data = response.meta['repo_data']

    #check repo
    is_empty = 'This repository is empty' in response.text
    repo_data['is_empty'] = is_empty

    if is_empty:
        repo_data['languages'] = None
        repo_data['commits'] = None
        self.logger.info(f"Empty repository:
{repo_data['name']}")

        standardized_output = {
            'url': repo_data['url'],
            'about': repo_data['about'],
            'last_updated': repo_data['last_updated'],
            'languages': None,
            'commits': None
        }

        self.logger.info(f"Repository {repo_data['name']} is
empty - settings languages and commits to None")
        yield standardized_output
        return

    #langs format
    langs = []
    language_elements =
response.css('div.repository-lang-stats-graph
span.language-color::attr(aria-label)').getall()

    for lang_text in language_elements:
        if lang_text and " " in lang_text:
            lang_name = lang_text.split(" ")[0]
            langs.append(lang_name)

    #lg
    if not langs:
```

```
        lang_elements = response.css('a[href*="search?l="]
span::text').getall()
        langs = [lang.strip() for lang in lang_elements if
lang.strip()]

    languages = ", ".join(langs) if langs else "None"
    repo_data['languages'] = languages

    #commit
    commits = None
    commit_texts = response.css('a[href*="commits"]
span::text').getall()
    for text in commit_texts:
        match = re.search(r'\d+', text)
        if match:
            commits = match.group(0)
            break

    repo_data['commits'] = commits if commits else '0'
```

Key features of this method include:

- Fallback mechanisms for language detection using multiple selector strategies
- Regular expression usage to extract commit counts
- Proper handling of empty repositories
- Standardized output formatting

Chapter 2 Results

1.1 Result 1: Expected Output

The spider produces JSON output with the following fields for each repository:

```
{
  "url": "https://github.com/username/repository-name",
  "about": "Repository description or 'No description'",
  "last_updated": "YYYY-MM-DD or 'Not specified'",
  "languages": "Language1, Language2, Language3 or None",
  "commits": "Number of commits or None"
}
```

For empty repositories, the spider sets languages and commits to None.

Chapter 2 Conclusions

The GitHub spider showcases a sophisticated implementation of web scraping techniques using the Scrapy framework. By using Scrapy's powerful features, the spider efficiently navigates through GitHub's repository pages and extracts valuable information in a structured format. The implementation demonstrates strong proficiency in using regular expressions to parse text data, particularly for extracting numeric information like commit counts, while exhibiting careful attention to date handling and defensive coding approaches that allow the spider to adapt to different page layouts. The spider's ability to produce standardized output regardless of input variations demonstrates good data engineering principles, with thoughtful consideration for edge cases like empty repositories.

For future enhancements, the spider could be extended to support authentication for accessing private repositories, implement rate limiting to avoid API restrictions, extract additional metrics like stars and forks, and add support for organization pages. Overall, this GitHub spider effectively demonstrates how to build a robust web crawler that can handle the complexities of web applications with dynamically generated content while maintaining clean, well-structured, and maintainable code.