Intelligent Web Crawler & Analyzer: Process Documentation

Prepared for User at 10:31 PM EEST, May 21, 2025

May 21, 2025

Contents

1 Overview

The Intelligent Web Crawler & Analyzer is a Python-based tool designed to analyze website crawlability, extract metadata (titles, descriptions, links), detect JavaScript-heavy content and APIs/RSS feeds, visualize results via a Streamlit dashboard, and store data in SQLite with CSV export capabilities. This document details the development process, execution attempts on the user's device (`C:\Users\pc\Desktop\web-crawler`), challenges, solutions, findings, and reflections.

2 Development Process

2.1 Project Setup

- Created a single Python file (`crawler.py`) to handle all functionality: crawling, analysis, visualization, and storage.
- Used libraries: `aiohttp` (async HTTP requests), `BeautifulSoup` (HTML parsing), `playwright` (JS rendering), `streamlit` (GUI), `plotly/pandas` (visualization), `sqlite3` (storage), `feedparser` (RSS), and `logging` (error handling).
- Set up `requirements.txt` with dependencies: `aiohttp`, `beautifulsoup4`, `playwright`, `streamlit`, `plotly`, `pandas`, `sqlite3`, `feedparser`, `pylint`, `pytest`.
- Added Gulp automation (`gulpfile.js`, `package.json`) for linting, testing, and building.
- Created a test file (`tests/test_crawler.py`).

2.2 Core Functionality Development

- **Crawlability Analysis**: Implemented `analyze_robots_txt` to parse `robots.txt`, extracting `can_crawl`, `crawl_delay`, `sitemap_urls`, and `disallowed_paths`.
- **Content Extraction**: Developed `extract_content` to extract titles (`<h1>`, `<h2>`, `<h3>`), meta descriptions, and links (up to 50) with retry logic and pagination detection.
- **JS/API Detection**: Built `check_js_and_api` to detect JS-heavy content, API endpoints (`/api`, `/json`), and RSS feeds (`/rss`).
- **Storage**: Used SQLite (`init_db`, `store_data`) to store results in `crawled_data.db`.
- **Visualization and GUI**: Created a Streamlit dashboard with metrics, tabs, Plotly histogram, recommendations, and CSV download.

2.3 Automation and Testing

- Configured Gulp tasks to clean, lint (`pylint`), test (`pytest`), and build.
- Wrote tests in `tests/test_crawler.py` to verify core functions.

2.4 Execution on User's Device

- **Setup**: User set up the project in `C:\Users\pc\Desktop\web-crawler`.
- **Dependency Installation**:
  - Initial errors: `'pytest'` and `'streamlit'` not recognized (21:45 EEST).
  - Resolved by running `pip install -r requirements.txt` and `npm install`.

- **Running**: Launched Streamlit GUI at `http://localhost:8501`, tested with `https://www.cnet.com` and `https://example.com`.

2.5 Challenges and Solutions

- **Playwright Compatibility**:
  - Error: `NotImplementedError` in Playwright (logs at 21:52, 21:58).
  - Cause: Python 3.13 incompatibility.
  - Solution: Recommended downgrading to Python 3.11; added exception handling in `check_js_and_api`.
- **Error Handling**:
  - Error: `'NotImplementedError' object has no attribute 'get'`.
  - Solution: Updated `analyze_website` to handle exceptions with `return_exceptions=True`.
- **Deprecation Warning**: Replaced `text="Next"` with `string="Next"` in `BeautifulSoup`.
- **Duplicate Code**: Removed duplicate `check_js_and_api`.
- **Website Access**: Used `https://example.com` as a fallback for sites blocking crawling.

2.6 Findings

- **Crawlability**: Parsed `robots.txt` for `https://example.com`: `can_crawl=True`, `crawl_delay="Not specified"`.
- **Content Extraction**: Extracted titles, descriptions, links; visualized links with Plotly histogram.
- **JS/API Detection**: Failed due to Playwright errors; used fallback values.
- **Storage**: SQLite stored results, enabling CSV export.
- **Recommendations**: Suggested `aiohttp` and `BeautifulSoup` for static content.

2.7 Deployment

- **Local**: Ran at `http://localhost:8501` after fixes.
- **Streamlit Cloud**: Provided steps to deploy via GitHub and `share.streamlit.io`.

3 Reflections

- **Efficiency**: `asyncio` improved performance for concurrent tasks.
- **Robustness**: Exception handling ensured the app didn't crash on failures.
- **Challenges**: Python 3.13 incompatibility was a major hurdle; earlier validation could have helped.
- **Future Improvements**:
  - Use Python 3.11 from the start.
  - Explore alternative JS rendering libraries (e.g., `selenium`).
  - Add scheduled crawling with `schedule` library.