REPORT ON ASYNC WEB CRAWLER

1. PROBLEM STATEMENT

The task was to build an asynchronous web crawler that was capable of crawling websites while respecting domain boundaries and robots.txt rules. The crawler must be capable of handling modern JavaScript-heavy platforms like WordPress, React, and Next.js, and output structured data for analysis.

Let's get real, web crawling the modern web is like herding cats, dynamic, unstable, and perhaps likely to throw CAPTCHAs at you.

2. PROJECT SCOPE

The crawler was required to:

- Crawl a list of seed URLs, following internal links up to a configurable depth.
- Respect robots.txt and politely handle rate limiting.
- Detect and log failed or blocked pages, including CAPTCHA-protected ones.
- Support dynamic content rendered via JavaScript.
- Produce outputs in Markdown and JSON formats for easy indexing and retrieval.
- Be deployable cross-platform on Windows, macOS, and Linux.

3. PROJECT WORKFLOW

A. Environment Setup

Virtual environments were created for Python 3.11 to isolate dependencies. Packages needed were all recorded for reproducibility. Docker containerisation was optionally included to ensure platform behaviour consistency.

B. Initial Crawler Development

A basic HTML crawler was created for single seed URLs. Static HTML parsing was accomplished with BeautifulSoup. Initial testing revealed CAPTCHA pages blocked automated access and required special handling.

C. Advanced Crawling Capabilities

Selenium and Pyppeteer were introduced to support JavaScript-heavy pages. CAPTCHAs were detected and logged, and optional HTML snapshots of blocked pages were recorded. Duplicate URL and domain limitation checking enabled efficient crawling.

D. Output Processing

Pages crawled were converted in Markdown format with descriptive filenames. A JSONL index was maintained for all successfully crawled pages. Failed URLs and exceptions were logged in a dedicated file.

4. FEATURES

- Asynchronous Crawling: Utilised asyncio and aiohttp for high performance.
- Domain & Depth Control: Configurable depth and domain restriction.
- Error & Exception Handling: Managed timeouts, retries, SSL issues.
- CAPTCHA Handling: Detection, HTML snapshotting, and logging.
- Cross-Platform Deployment: Virtual environments and optional Docker.
- Structured Outputs: Markdown pages and JSONL index for each crawled URL.

5. PROJECT STRUCTURE

```
/scraper/
├── crawler.py          # Main crawler script
├── crawlLog.txt        # Detailed logging output
├── seeds.txt           # Priority URLs to crawl
├── .gitignore          # Ignored files/folders
├── requirements.txt    # Python dependencies
└── output/
    ├── MDs/            # Markdown files for each crawled page
    ├── index.jsonl     # JSON index of all crawled pages
    └── failed_urls.txt # URLs that failed to crawl
```

6. CHALLENGES AND SOLUTIONS

- CAPTCHAs and IP Blocking: Manual intervention when automated detection failed.
- Javascript Rendering: Selenium and Pyppeteer solved dynamic content problems.

- Cross-Platform Execution: Virtual environments allowed local compatibility; Docker enabled reproducibility.
- SSL Certificates and Security Errors: Proper logging and exception handling were implemented.

It is like trying to walk a tightrope while juggling burning torches – difficult, but ultimately satisfying when nothing catches on fire.

7. KEY OBSERVATIONS

- The crawler managed to handle static and dynamic sites.
- CAPTCHAs remain the primary limitation, though detection and logging minimize their impact.
- Docker simplifies reproducibility and deployment to collaborators or end-users.
- Structured output facilitates downstream analysis or further processing.

8. CONCLUSION

The project demonstrates that a well-structured asynchronous web crawler can manage modern websites, respect polite crawling rules, and produce clean, structured outputs. Cross-platform deployment is achievable using Python virtual environments or containerisation.