# Report on the Web Scraping Project

Armin Felahatpisheh

April 2024

# Contents

## 0.1 Introduction

The exponential growth of data on the internet demands effective methods for obtaining pertinent information. Data extraction from websites, or "web scraping," has become an essential tool for data analysis and decision-making. Our goal in this project is to create a reliable and effective web scraping system that follows a number of important guidelines to maximize efficiency and guarantee data integrity.

My method is based on a multi-threaded architecture that maximizes throughput by allowing multiple URLs to be processed simultaneously. We can greatly accelerate the process of scraping data from numerous websites by utilizing concurrency. Concurrency, however, increases the possibility of race conditions, which may contaminate data. In order to mitigate this issue, we utilize synchronization techniques like locking to guarantee the secure management of data access and modification among various threads.

Finally, we will go over the web page being scraped and count the number of unique words.

**Key features of the scraper include:**

- **Single Processing of Each Page:** Each web page is processed exactly once to avoid redundancy and reduce workload.
- **Efficient Queue Management:** Pages to be scraped are managed using an efficient queue system that supports fast retrieval and insertion, critical for maintaining high scraping speeds.
- **Comprehensive Content Scraping:** The system is designed to extract all content from the pages, ensuring a thorough data collection.
- **Link Extraction:** After the scraping, the system extracts further pages (links) from the scraped content, expanding the scope of data collection continuously.
- **Distinct Word Counting:** For each page, the number of unique words is determined using the Flajolet-Martin[1] algorithm, a probabilistic approach that efficiently estimates the number of distinct elements in a stream.
- **Data Storage:** All scraped pages along with their corresponding unique words count are stored systematically for further analysis.

This structured and methodical approach not only enhances the scraping efficiency but also ensures that the data extracted is accurate and useful for analytical purposes. The subsequent sections will delve deeper into the implementation details and the performance evaluation of the scraper.

## 0.2 Related Work

Web scraping, the process of extracting data from websites, has seen numerous improvements in efficiency and speed through various technological advancements. The primary focus has been on improving the concurrency of scraping processes, reducing the latency in data retrieval, and enhancing the parsing mechanisms. Concurrency can be significantly enhanced by using multithreading or asynchronous programming models, which allow a scraper to handle multiple pages simultaneously rather than processing them sequentially. Libraries such as `Scrapy` in Python leverage asynchronous operations to manage multiple website requests at once, considerably speeding up the data collection process.

Another approach to accelerate web scraping is through the use of distributed systems. Tools like Apache Nutch and StormCrawler enable scaling across multiple machines, distributing the workload and thus, reducing the time required to scrape large volumes of data. Moreover, the efficiency of parsing the HTML content can be improved by using optimized libraries like `BeautifulSoup` and `lxml`, which provide fast parsing capabilities and can handle poorly formed HTML code effectively.

Moreover, reducing overheads related to network requests is crucial for speeding up web scraping. Techniques such as setting up local caching mechanisms to avoid redundant fetches and using data compression to reduce the amount of data transferred over the network are commonly employed. Additionally, careful management of request headers and session handling can minimize the chance of being blocked by target websites, thus maintaining a steady flow of data.

Overall, the field of web scraping continues to evolve with an emphasis on enhancing speed and efficiency through better use of modern programming practices and architectures. These improvements not only facilitate quicker data retrieval but also ensure that scrapers can operate at a large scale without compromising on performance.

## 0.3 Implementation

### 0.3.1 Algorithm

**Algorithm 1** Descriptive Pseudocode for Page Scraping Algorithm

**procedure** StartPageScraping
    **while** counter is less than limit and queue is not empty and cursor is less than queue length **do**
        Determine the number of pages to be processed concurrently.
        Initialize a thread pool executor with the number of determined pages as the maximum number of workers.
        Submit scraping tasks for each page to the thread pool.
        **for** each future result from the thread pool **do**
            Receive the results of the scraping including parsed HTML
            Mark the page as visited.
            Increment the counter for processed pages.
            Store the results of the scraped page.
            Extract and add new links from the scraped page to the queue.
        **end for**
        Close the thread pool.
    **end while**
**end procedure**

### 0.3.2 Initialization

The initialization phase sets up the scraping process by configuring the initial variables and structures needed for web scraping. The scraper class is initialized with a list of domains we want to scrape and extract links from, a limit on the number of pages to scrape, and the number of threads to use.

```
lock = threading.RLock()

class Scraper:
    def __init__(self, domains: List[str], limit: int,
    threads_num_used: int):
        self._limit = limit
        self._threads_num_used = threads_num_used
        self._counter = 0
        self._visited_pages = set()
        self._queue = [d for d in domains]
        self._cursor = 0
        self._results = {}
```

**Lock Initialization:**
A `threading.RLock()` (reentrant lock) is instantiated, allowing the same thread to acquire the lock multiple times. This is critical in a multithreaded environment to manage shared resources like the queue of URLs and visited pages for

our case. This lock assures that at any point, only one thread is using the shared data sources, guaranteeing avoiding race conditions. One race condition, for example, which is easily avoided using this lock is when 2 thread receive the same page.

**Constructor (`__init__`):**
Initializes the Scraper object with a list of domains, a limit to the number of pages to scrape, and the number of threads to use. It sets up necessary variables such as counters, queue of pages to visit, and storage for results.

- `self._limit = limit`: This variable stores the maximum number of pages the scraper is allowed to process. The `limit` is passed as a parameter to the constructor of the class, and setting `self._limit` to this value ensures that the scraper does not exceed the predefined limit of pages to scrape.
- `self._threads_num_used = threads_num_used`: This variable determines how many threads the scraper will use simultaneously for scraping. It is set based on the `threads_num_used` parameter provided during initialization. This is important for managing the level of concurrency in scraping operations.
- `self._counter = 0`: This counter is used to track the number of pages that have been successfully scraped. It starts at 0 and increments as pages are processed, helping to ensure that scraping stops when the limit is reached.
- `self._visited_pages = set()`: This set holds the URLs of pages that have already been visited to avoid re-scraping the same pages. It ensures that each page is processed only once, optimizing the scraping efficiency.
- `self._queue = [d for d in domains]`: This list acts as a queue of domains or URLs to be scraped. It is initialized with the list of domains passed to the constructor. The queue supports the breadth-first (BFS[2])scraping strategy by storing and managing upcoming pages to visit.
- `self._cursor = 0`: The cursor is an index that helps in accessing pages from the `self._queue`. It indicates the current position in the queue from which the next page to be scraped is selected.
- `self._results = {}`: This dictionary is used to store the results of the scraping operations. Typically, it could hold data like the count of distinct words on each page or other relevant information extracted during scraping.

**Helper Methods:**

- `_store_scraped_page_result`: This method processes a parsed HTML page, counts distinct words, and stores the result. It uses a lock to ensure thread-safe operations when updating the results and writing to a file.
- `_get_pages_for_threads_num`: This method retrieves a list of pages equal to the number of threads available. It ensures that pages are not revisited using the lock to manage the cursor and visited pages set.
- `_add_page_to_visited`: Adds a page to the set of visited pages, using a lock to ensure thread safety.
- `_extract_and_add_links_from_page`: Extracts all links from a parsed page and adds them to the queue of pages to be visited if they haven't been visited yet, using a lock to manage updates to the queue and visited pages set.

**How do we guarantee that each page is scraped only once?**

In the `Scraper` class, the mechanism to ensure that each page is scraped only once is primarily facilitated by using a set called `self._visited_pages`. This set is initialized as an empty set in the constructor of the class. In Python, sets are collections that do not allow duplicate elements, a property that is used here to store URLs of pages that have already been visited. When a new page is about to be processed, the scraper checks whether the URL of this page is already present in the `self._visited_pages` set. If the page is not in the set, it is added to this set during the processing, ensuring that no page is processed more than once, thus optimizing the efficiency of the scraping process and preventing redundant work.

**What is the time complexity of adding a page URL into `self._visited_pages` and checking if the page has been added already?**

The `self._visited_pages` is implemented as a set in Python, which is an abstract data type that supports highly efficient operations for adding elements and checking membership. Adding an element to a set (`set.add()`) and checking if an element is in the set (`in`) both typically operate in average time complexity of **O(1)**. This efficiency is due to the underlying hash table implementation, which allows it to quickly determine if an element is present without having to check each element in the set. However, in the worst case, such as when rehashing occurs due to a large number of hash collisions, these operations can degrade to **O(n)**.

**What is the time complexity of adding and getting a page URL from `self._queue` using `self._cursor`?**

1. **Adding a Page URL to `self._queue`:** The operation of adding an element to the end of a list in Python (`list.append()`) is an **O(1)** amortized [3] operation. This is because lists in Python are dynamic arrays,

and appending to them does not typically require shifting of other elements (except when resizing the underlying array, which is an infrequent amortized operation).

2. **Getting a Page URL using `self._cursor`:** Accessing an element from a list by index, such as `self._queue[self._cursor]`, is also an **O(1)** operation. Lists in Python support direct access via indices, allowing for constant time retrieval, which is efficient and crucial for the operation of a web scraper handling numerous URLs.

### 0.3.3 Multithreading Setup

The scraper uses Python's `concurrent.futures` module to manage a pool of threads. Each thread is tasked with processing a different web page simultaneously, which significantly speeds up the scraping process.

```
def _start(self):
    with concurrent.futures.ThreadPoolExecutor(max_workers=self._threads_num_used) as executor:
        pages_extracted = self._get_pages_for_threads_num()
        futures = [executor.submit(playwright_scraper, page) for page in pages_extracted]
```

### 0.3.4 Page Processing

Each thread processes a page by invoking the `playwright_scraper` function, which handles the downloading and rendering of the page content using the Playwright tool. The Algorithm 2 is implemented to ensures all dynamic content is loaded.

**Algorithm 2** Scroll to the Bottom of a Web Page Until No New Data is Loaded

```
procedure ScrollUntilNoData
    Initialize previousHeight to 0
    Define function checkAndScroll
    Start function checkAndScroll

    function checkAndScroll
        currentHeight ← maximum of document's scrollHeight or offsetHeight
        if currentHeight > previousHeight then
            Scroll to position (0, currentHeight)
            Update previousHeight to currentHeight
            Pause execution for 2000 milliseconds
            Call checkAndScroll again
        else
            End the function
        end if
    end function
end procedure
```

### 0.3.5 The Visited Pages and The Queue of the Pages

Management of visited pages and the page queue is crucial to avoid reprocessing the same page and to ensure that the scraper efficiently traverses through all available links.

```python
def _add_page_to_visited(self, page):
    global lock
    with lock:
        self._visited_pages.add(page)
```

### 0.3.6 Data Extraction and Storage

Extracted data is processed to count unique words using a custom algorithm and stored in a structured format using `PrettyTable` for easy analysis and reporting.

```python
def _store_scraped_page_result(self, parsed_html: BeautifulSoup, page: str):
    text = parsed_html.get_text(separator=' ')
    words = text.split()
    with lock:
        self._results[page] = count_number_of_distinct_words(words)
        ... # Writing the result to a .txt file.or sending it to a database
```

### 0.3.7 Link Extraction and Queue Management

New links are extracted from each processed page and added to the queue for future scraping, ensuring continuous operation. The main question is what are the 'valid' links we should extract.

The following numbered items delineate the conditions under which a link is considered valid in the given web scraping context:

1. **Normalization:** The URL must not end with a slash ('/'). If it does, the slash is removed.
2. **Presence of "www":** If the URL does not contain "www", it is normalized to include "www" after the scheme and before the network location.
3. **Scheme Inclusion:** The URL must have a scheme (e.g., 'http', 'https'). If missing, it is assumed based on context.
4. **No Invalid Characters:** The URL must not contain any of the following characters: '[# $ [ ] ?]'. These characters are typically used in URLs for navigation within a page, which might not be relevant for web scraping.
5. **Standard Protocols Only:** URLs starting with "javascript:", "mailto:", or "tel:" are excluded as they are not valid web pages but are rather script handlers or communication links.
6. **Domain Limitation:** The URL should not contain more than two dots ('.'), preventing deep subdomains.
7. **Length Restriction:** The URL cannot exceed 512 characters, facilitating manageability and avoiding excessively long URLs which might be erroneous or malicious.
8. **Valid URL Format:** The URL must pass a standard URL validation check, ensuring that it is syntactically correct according to general URL formatting rules.

### 0.3.8 Finding insights from the results saved

Analyzing the number of distinct words on a webpage can serve as a indicator of its authenticity. Webpages that host genuine content typically employ a diverse vocabulary to show ideas comprehensively and accurately. Conversely, fraudulent or "fake" pages often lack substance and may repetitively use a limited set of keywords aimed at search engine optimization rather than offering quality information. This repetitiveness results in a low count of unique words. Moreover, these pages might use excessive jargon, nonsensical phrases which further skews the natural distribution of language seen in legitimate articles. Therefore, by examining the diversity of vocabulary through the count of distinct words, it's possible to discern patterns that differentiate credible content from deceptive or low-quality web pages. This metric, when combined with other analytical tools,

enhances the effectiveness of algorithms designed to detect and flag potentially fake content on the internet.

**Algorithm 3** Flajolet-Martin Algorithm for Estimating Distinct Words

```
1: procedure FlajoletMartin(stream)
2:     bitmaps ← array of bit vectors initialized to 0
3:     numHashFunctions ← number of different hash functions
4:     for each item in stream do
5:         for i ← 1 to numHashFunctions do
6:             hashValue ← hash(item, i)
7:             r ← trailingZeros(hashValue)
8:             bitmaps[i][r] ← 1
9:         end for
10:    end for
11:    averages ← array of 0s
12:    for i ← 1 to numHashFunctions do
13:        R ← the smallest index of 0 in bitmaps[i]
14:        averages[i] ← 2^R
15:    end for
16:    result ← median of averages
17:    return result
18: end procedure
```

## 0.4 Results

Now we will go over some results and illustrate some statistics.

**Execution Time vs. Number of Threads** The graph depicting the relationship between execution time and the number of threads reveals an initial decline in execution time as the number of threads increases from 1 to 4. This trend suggests that parallel processing effectively reduces the total execution time, likely due to more tasks being executed simultaneously. However, beyond 4 threads, the execution time plateaus and even slightly increases, indicating a potential overhead associated with managing a larger number of threads. This overhead could be due to factors such as thread synchronization and resource contention, which negate the benefits of additional parallelism. Figure 1

**CPU Usage vs. Number of Threads** The CPU usage as a function of the number of threads demonstrates a steady increase. This is expected as more threads consume more CPU resources. The data points indicate a linear relationship, with CPU usage increasing proportionally with the number of threads. This linear increase suggests that the system efficiently scales with the addition of more threads, without showing signs of CPU saturation or excessive overhead at the higher thread counts examined. This efficiency might indicate that the workload is well-suited to parallel processing, utilizing the CPU effectively across the additional threads. Figure 2

**RAM Usage vs. Number of Threads** In the graph showing RAM usage against the number of threads, there is a gradual increase in RAM consumption as more threads are utilized. The increase is not as steep as CPU usage, which implies that while more memory is required for managing additional threads, the memory overhead is not disproportionately high. Notably, the graph shows slight variations in RAM usage for the same number of threads, possibly reflecting different memory management efficiencies depending on the specific tasks being performed by each thread. Overall, the increase in RAM usage is moderate, suggesting that the application handles memory efficiently across varying thread counts. Figure 3

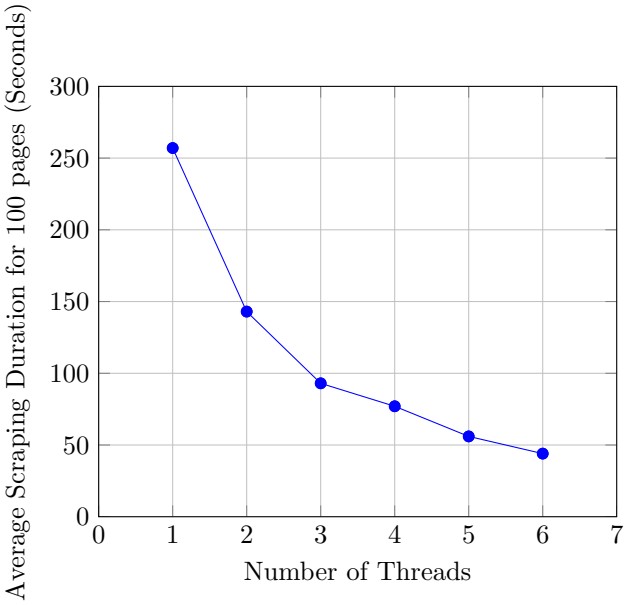


Figure 1: Average CPU usage as a function of the number of threads.

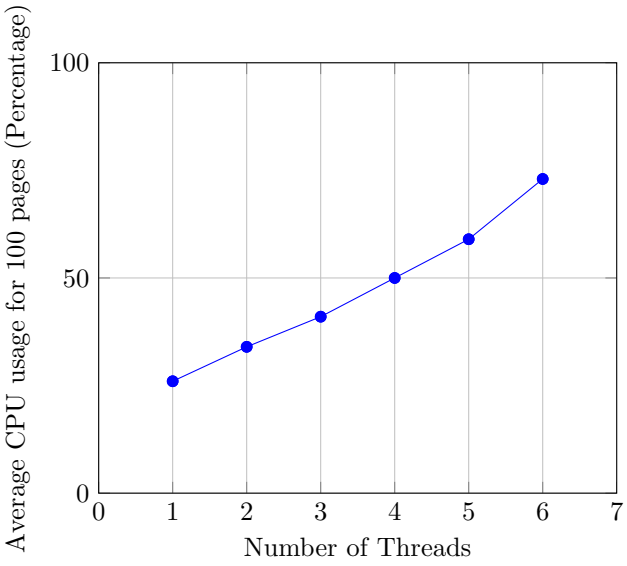


Figure 2: Average scraping duration as a function of the number of threads.

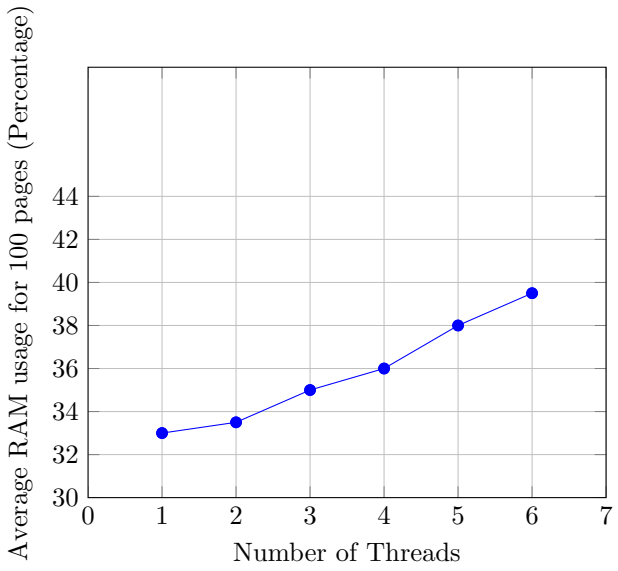


Figure 3: Average RAM Usage as a Function of the Number of Threads.

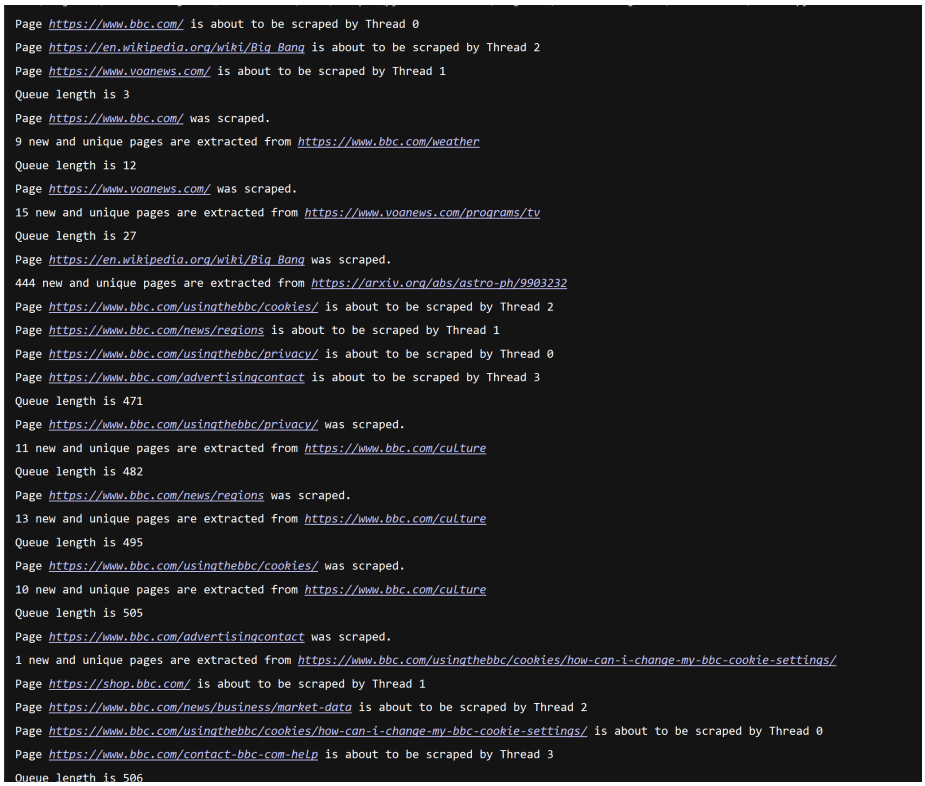


Figure 4: The logs from running the scraper

| Page | Distinct Words Count |
|---|---|
| https://www.bbc.com/ | 8192 |
| https://en.wikipedia.org/wiki/Big_Bang | 8192 |
| https://www.bbc.com/advertisingcontact | 4096 |
| https://www.bbc.com/usingthebbc/cookies/ | 512 |
| https://www.bbc.com/usingthebbc/cookies/how-can-i-change-my-bbc-cookie-settings/ | 512 |
| https://shop.bbc.com/ | 512 |
| https://www.bbc.com/news/regions | 256 |
| https://www.bbc.com/news/business/market-data | 256 |
| https://www.bbc.com/contact-bbc-com-help | 256 |
| https://www.bbc.com/weather | 256 |
| http://pronounce.voanews.com/ | 256 |
| https://www.voanews.com/ | 128 |
| https://www.bbc.com/usingthebbc/privacy/ | 64 |

Figure 5: The results that get sorted in descnding order of unique words count

## 0.5 Future Work

As we continue to enhance the capabilities and performance of our web scraping system, several areas have been identified for future development. These improvements aim to further accelerate the data collection process, improve the robustness of the system, and enhance the quality of the data extracted.

**Using Multiprocessing:** One promising direction is the utilization of multiprocessing in addition to multithreading. While multithreading has provided substantial speed gains by enabling parallel processing of multiple pages, multiprocessing can take this a step further by distributing the workload across multiple CPU cores. This approach can potentially increase the scraping speed by running several processes in parallel, each handling its own set of threads. Future experiments will focus on implementing and benchmarking a multiprocessing model to evaluate its impact on performance, especially in terms of handling I/O-bound and CPU-bound tasks more efficiently.

**Experimentation with Asyncio:** Another area of interest is the integration of asynchronous programming using Python's `asyncio` library. Asynchronous programming can reduce the overhead caused by thread management and improve the efficiency of network operations. By experimenting with `asyncio`, we aim to optimize the handling of simultaneous network requests and I/O operations, thereby reducing the latency and improving the scalability of our scraping operations. Comparative studies will be conducted to measure the performance benefits of asynchronous versus synchronous scraping models.

**Identifying Potentially Fake Pages:** Ensuring the authenticity and reliability of the data collected is paramount. Future work will also include developing methodologies to identify and filter out potentially fake pages. This could involve analyzing the content for signs of misinformation, such as checking the authenticity of the domain, analyzing the quality of the content, and employing machine learning models trained to detect patterns typical of fake information. Implementing these checks will help maintain the integrity of the data collected and protect against the inclusion of misleading or false information.

These future directions will not only aim to improve the speed and efficiency of our web scraping tools but also enhance their reliability and the quality of the data they generate. Each of these initiatives will require careful planning, implementation, and thorough testing to ensure they meet the intended goals.

# Bibliography


[1] Arpit Bhayani. *The Flajolet-Martin Algorithm*. Available at: `https://arpitbhayani.me/blogs/flajolet-martin/` Accessed on: [April 2024].

[2] Khan Academy. *The Breadth-First Search Algorithm*. Available at: `https://www.khanacademy.org/computing/computer-science/algorithms/breadth-first-search/a/the-breadth-first-search-algorithm` Accessed on: [April 2024].

[3] Oliveira, R., *Lecture Notes on Web Scraping*, University of Waterloo, Fall 2020, `https://cs.uwaterloo.ca/~r5olivei/courses/2020-fall-cs466/lecture1.pdf`. Accessed on: [April 2024].

[4] Armin Felahatpisheh, *Hire a Backend Engineer for a Data-Driven Enterprise*, `https://proxify.io/articles/hire-a-backend-engineer-for-a-data-driven-enterprise`.

[5] Playwright Team, *Playwright*, `https://playwright.dev/`.

[6] Richardson, Leonard, *Beautiful Soup Documentation*, `https://beautiful-soup-4.readthedocs.io/en/latest/`.

[7] Google Developers, *Learn JavaScript*, Web.dev, `https://web.dev/learn/javascript/`.

[8] Python Software Foundation, *Threading — Thread-based parallelism*, Python 3.8.5 documentation, `https://docs.python.org/3/library/threading.html`.