WEB – CRAWLER

A web crawler is a program that automatically traverses the web by following hyperlinks from one webpage to another, collecting data along the way.

The main purpose of this bot is to learn about the different web pages on the internet. This kind of bot is mostly operated by search engines. By applying the search algorithm to the data collected by the web crawlers, search engines can provide the relevant links as a response for the request requested by the user.

The idea is that the whole internet can be represented by a directed graph:

- with vertices -> Domains/ URLs/ Websites.
- edges -> Connections.

Approach: The idea behind the working of this algorithm is to parse the raw HTML of the website and look for other URL in the obtained data. If there is a URL, then add it to the queue and visit them in breadth-first search manner.

Step 1: Requirements and Goals of the System

- **Scalability:** Our service needs to be scalable such that it can crawl the entire Web and can be used to fetch hundreds of millions of Web documents.
- **Extensibility:** Our service should be designed in a modular way, with the expectation that new functionality will be added to it. There could be newer document types that need to be downloaded and processed in the future.

Step 2: Design Considerations

→ Crawling the web is a complex task, and there are many ways to go about it.
→ We should be asking a few questions before going any further:

Crawler for HTML pages:

- We must write a general-purpose crawler to download different web documents for which we must write parsing module for HTML.

What are the required protocols:

- HTTP protocol is required to exchange resources between crawler and internet.

What is the expected number of pages we will crawl?

- Assuming we need to crawl one billion website links.
- Since a website can contain many, many URLs, let's assume we have 15 billion different web pages that will be reached by our crawler.
- We can use maximum depth for defining the number of pages in search.

Step3: Capacity Estimation and Constraints

Traffic Estimates:

- If we **want to crawl 15 billion pages in 4 weeks**, then **pages need to fetch per second:** 15B/(4*7*86400 sec)~= **6200 pages/sec**.

Storage Estimates:

- Page sizes vary a lot, but as mentioned above since we will be dealing with **HTML text only**, let's assume an average page size be 100KB.
- With **each page** if we are storing **500 bytes of metadata**, then **total storage we need**: 15B * (100KB + 500B) ~= **1.5 petabytes.**

Step 4: High Level Design

Basic Algorithm

- The basic algorithm executed by any Web crawler is to take a list of seed URLs as its input.
- And then repeatedly execute the following steps:

i. Pick a URL from the unvisited URL list.
ii. Determine the IP Address of its hostname.
iii. Establishing a connection to the host to download the corresponding document.
iv. Parse the document contents to look for new URLs.
v. Add the new URLs to the list of unvisited URLs.

vi. Process the downloaded document, e.g., store it or index its contents, etc.
vii. Go back to step 1.

How To Crawl?

Breadth First Search

- Breadth-first search (BFS) is usually used.
- *The **Breadth First Search (BFS)** algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.*

Difficulties in implementing efficient web crawler

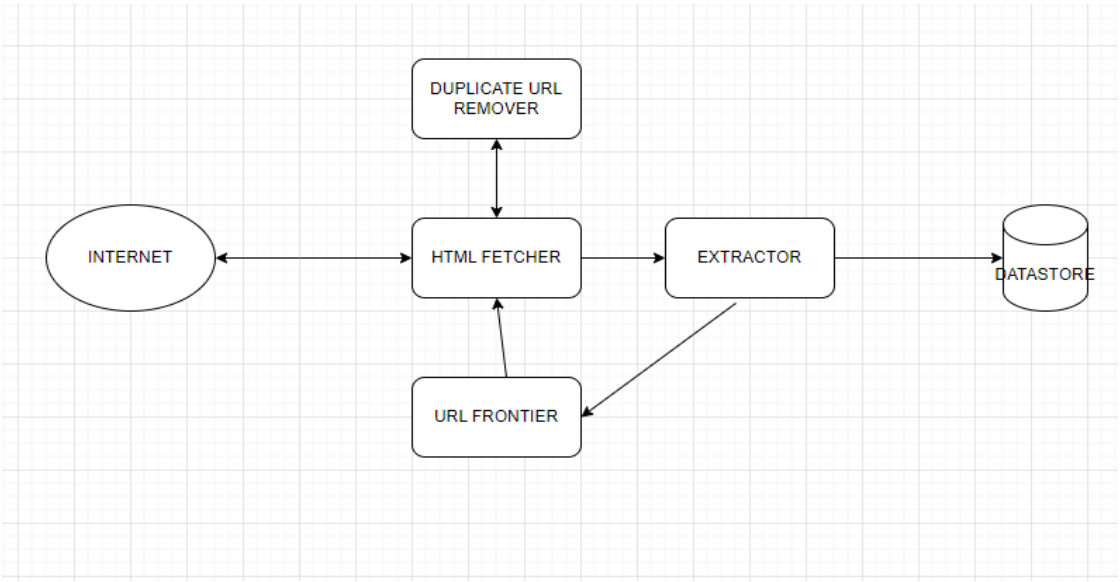
- There are two important characteristics of the Web that makes Web crawling a very difficult task:

1. **Large volume of web pages:** A large volume of web page implies that web crawler can only download a fraction of the web pages at any time and hence it is critical that web crawler should be intelligent enough to prioritize download.

2. **Rate of change on web pages:** Another problem with today's dynamic world is that web pages on the internet change very frequently, as a result, by the time the crawler is downloading the last page from a site, the page may change, or a new page has been added to the site.

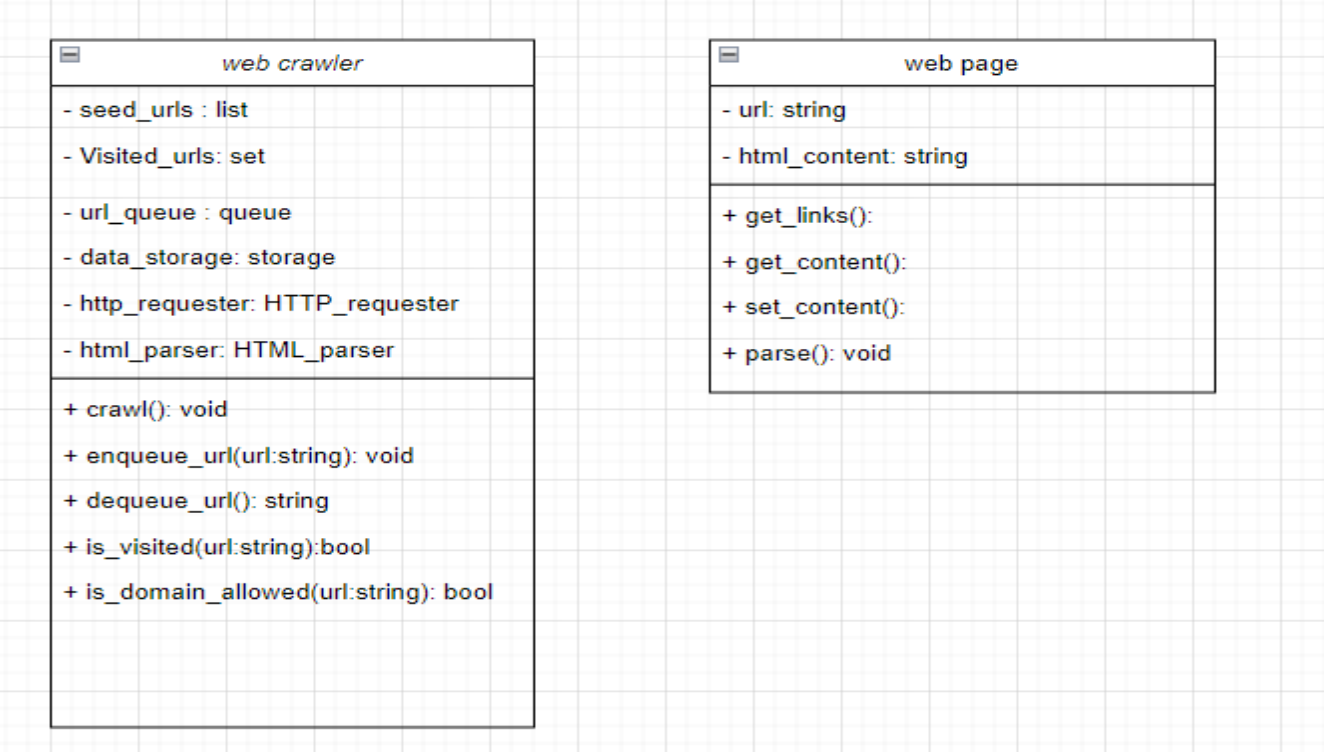
A bare minimum crawler needs at least these components:

1. **URL frontier:** To store the list of URLs to download and prioritize which URLs should be crawled first.
2. **HTTP Fetcher:** To retrieve a web page from the server.
3. **Extractor:** To extract links from HTML documents.
4. **Duplicate Eliminator:** To make sure same content is not extracted twice unintentionally.
5. **Datastore:** To store retrieve pages and URL and other metadata.



Component Diagram for web-crawler

Step 5: Class required for designing web-crawler:



Class Diagram for the web-crawler

WebCrawler:

- This is the main class that orchestrates the crawling process.
- **Attributes:**

1. **seed_urls**: A list of initial URLs to start the crawl.
2. **visited_urls**: A set to keep track of visited URLs.
3. **url_queue**: A queue data structure to manage URLs.
4. **data_storage**: An instance of a class responsible for storing crawled data.
5. **http_requester**: An instance of a class responsible for making HTTP requests.
6. **html_parser**: An instance of a class responsible for parsing HTML content.

- **Methods:**

1. **crawl()**: Initiates the crawling process.
2. **enqueue_url(url: String)**: Adds a URL to the URL queue.
3. **dequeue_url(): String**: Removes and returns a URL from the queue.
4. **is_visited(url: String): bool**: Checks if a URL has been visited.
5. **is_domain_allowed(url: String): bool**: Checks if the URL's domain is allowed according to **robots.txt**.

Webpage:

- Represents a webpage.
- **Attributes:**

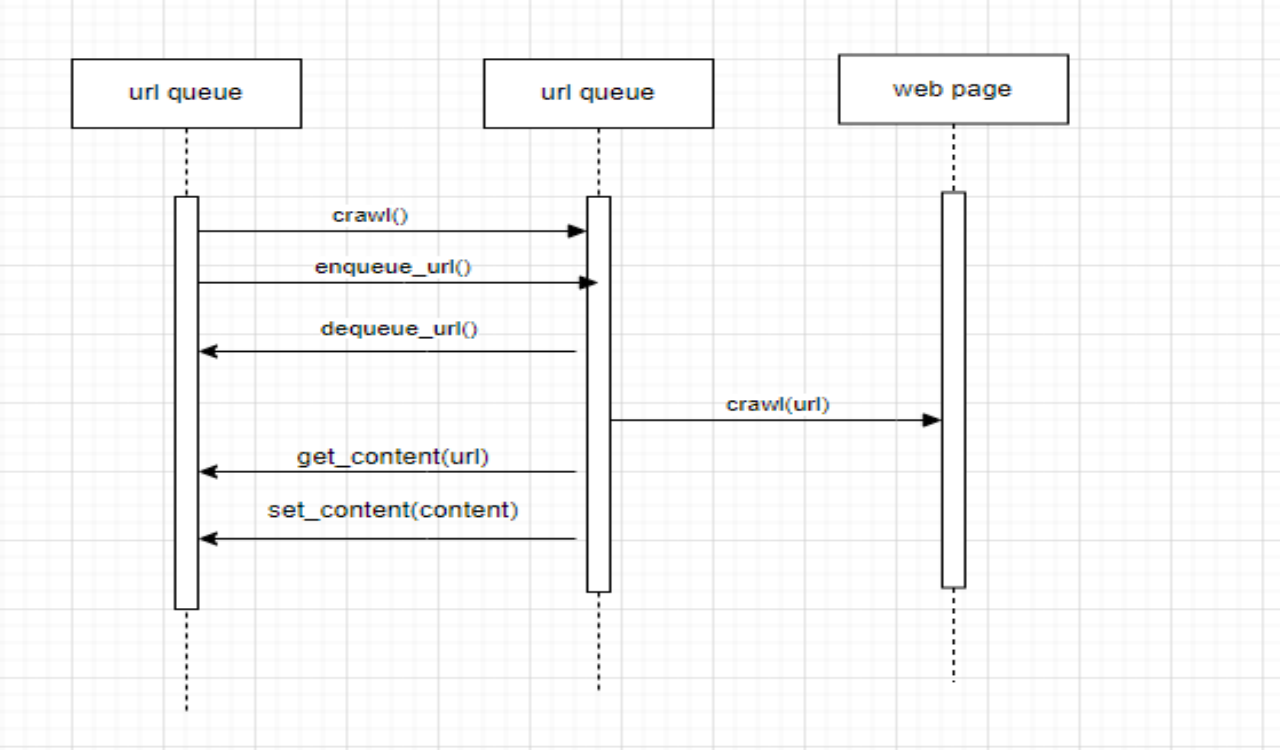
1. **url**: The URL of the webpage.
2. **html_content**: The HTML content of the webpage.

- **Methods:**

1. **get_links()**: Extracts and returns the links from the HTML content.
2. **get_content()**: Retrieves and returns the HTML content.
3. **set_content(content: String)**: Sets the HTML content.

- **HTTPRequester**: Handles sending HTTP requests and receiving responses.
- **HTMLParser**: Parses HTML content to extract information and links.
- **DataStorage**: Handles storing crawled data (e.g., HTML content or structured data).

Step 6: Sequence required for designing web-crawler:



Sequence Diagram for web-crawler

1. The **WebCrawler** object initiates the crawling process by calling the **crawl()** method.
2. The **WebCrawler** interacts with the **URL Queue** by enqueuing and dequeuing URLs using the **enqueue_url(url)** and **dequeue_url()** methods, respectively.
3. The **WebCrawler** checks if a URL has been visited by calling the **is_visited(url)** method.
4. When a URL is dequeued, the **WebCrawler** creates a **WebPage** object by calling **crawl(url)**.

5. The **WebPage** object is responsible for retrieving the HTML content of the webpage by calling **get_content(url)**.
6. The **WebPage** object sets the HTML content using the **set_content(content)** method.

Step 7: Error Handling and Logging:

1. Implement robust error handling to deal with network issues, website changes, and unexpected errors.
2. Log crawl progress, errors, and important events for debugging and analysis.

Step 8: Monitoring and Maintenance:

1. Continuously monitor the crawler's performance
2. Detect and handle issues and update it to adapt to changes in websites' structures.