WebCrawlerApp
Chris Sifeng Chen
April 17th, 2024

Overview and purpose:

This is a Web Crawler application written in Java using two external libraries. Apache HTTP Client mainly to send a GET request. Jsoup to parse Response Entity String value. The goal of this application is to scrape all reviews/publications filtered by topics that we see at the entry URL link, (https://www.cochranelibrary.com/cdsr/reviews/topics) and output each review onto a text file with a specific format.

Generic program design reasoning:

This application is built with the OOP paradigm, by making sure each component has its own responsibility, ensures code reusability, and maintainability, and allows us to test each class independently.

As an entry point, we have a WebCrawlerApp class with a main method. This class mainly takes care of creating instances as needed such as (Topic, Review, Crawler) and passing references to utility classes as needed (CrawlerUtils, Parser).

The main method in WebCrawlerApp starts by calling a CrawlerUtils static function, called CrawlHomePage(), this method will go ahead and scrape the home page and create a list of Topic instances with their URL, and return them back to the main method. From here, for each topic, the program will create a separate Crawler instance (extends the thread class) to make an HTTP Get request for each Topic instance. Within each thread, what it does is scrape for the total number of pages that exist for each topic simultaneously. The program then waits until all Crawler Threads have finished running, now we have a list of Topic instances with their corresponding topic title and pagination URLs.

Now we have our second multi-threading logic, for each topic, we first take the Topic's list of pagination URLs, and then the program creates a Crawler instance for each page URL of that list of URLs (so the total number of threads for each topic is the total number of pages of a topic search). Passing in an Enum argument for the parameter of the Crawler's constructor, this will then go ahead and initialize the type of running logic for the thread. (Enum: CRAWL_PAGE_URL). Furthermore, scraping the 25 reviews of all pages simultaneously and append them into the topic's Review List.

The above paragraph logic repeats for each Topic instance of the Topic list we created earlier. As soon as all pages of each topic are done scraping, the program iterates through the topic list again and writes their Review list data onto a txt file.

Justification of multi-threading decision:

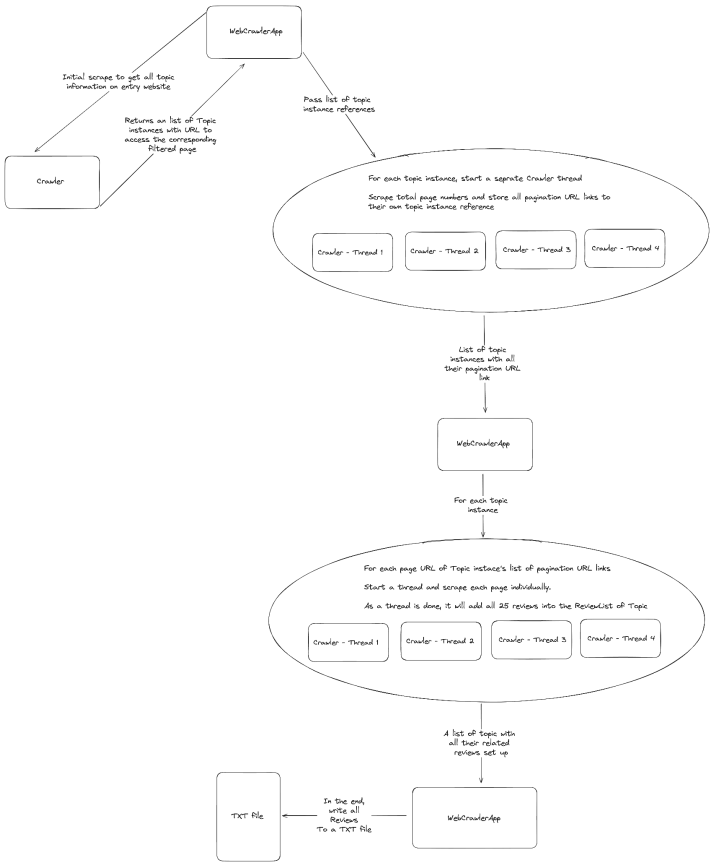
Originally I thought of starting a separate thread for each topic that we are trying to scrape. However, I quickly realized each topic's number of pages is very different, some are hundreds and some are single digits. Therefore, by starting only X number of threads, of which X is the total number of topics, this decision misses out on the opportunity to create more threads that can work simultaneously and make separate GET requests. On the other hand (the proposed design in the above section), if we iterate through the topics one by one, and for each topic, we create Y number of threads, for which Y is the number of pages of a single topic, we can speed our scraping process for each topic and move on to the next topic and do the same. This is way more flexible in terms of maximizing the efficiency of scraping each topic. Rather than having potentially one thread scrape 5 pages itself and another thread scraping 200 pages itself.

Usage of external libraries:

The usage of Jsoup was not required, but I wanted to use this external library because of its HTML parsing capabilities.

The Apache HTTP client was needed for its functionality to create an HTTP client that allowed me to send GET requests to the external server. However, I believe Java's built-in HttpClient could have achieved the same result.

High-level flow: (also in the GitHub repository)



Additional considerations:

The multi-threading code implementation is not very consistent. I believe there are no logic errors in my implementation but there are some improvements to be made relating to thread management and how to better control thread creation. Each thread is indeed making GET requests appropriately and doing the processing logic appropriately. However, some responses received from the GET request return have no HTML result and are then not parsable. This issue should be further investigated and debugged. Therefore, the default method that the program's main method calls will only prompt the user to select a single topic from a list of topics. It will then scrape the reviews of each page, which will then be outputted to a txt file as soon as all pages of the topic are scraped.

Future ideas to investigate further:

1) The url accessing each topic has several redirections to other API calls. For example, if you send a **curl** POST request (not a get this time), curl "https://www.cochranelibrary.com/en/c/portal/render_portlet?p_l_id=20759&p_p_id=scolarissearchresultsportlet_WAR_scolarissearchresults&p_p_lifecycle=0&p_t_lifecycle=0&p_p_state=normal&p_p_mode=view&p_p_col_id=column-1&p_p_col_pos=0&p_p_col_count=1&p_p_isolated=1&currentURL=%2Fsearch' \

-X 'POST' \", you can get the codes of all articles of a topic based on the parameter or filtering you pass into the POST request URL parameters. There can potentially be a way to get all codes of articles and avoid having to go through all pages of a topic and scrape reviews.

2) There is another potential method to scrape each page, since you are allowed to select how many results you wish to see on each page, there is potentially a way to use the total number of reviews existing for a topic and query for all reviews to show up on a single HTML page. That way we can minimize the number of GET requests sent through our own client and potentially scrape all reviews at once. This will not only minimize the efficiency but also the code readability of our program.