# Technical Overview

## Overview of logic

When I first began this assignment, I thought back to previous similar programs that I made. I knew that in order to scrape information from a website you have to use an API. In this case, I must scrape raw html returned from the API. The only thing that made this challenging was that the website used a lot of JavaScript to dynamically call endpoints and display data. Once I knew that I had to scrape HTML elements and filter info from the page I knew that I could either use browser automation or parse raw HTML. I wanted to make this program as fast as possible so I opted to parse the HTML rather than using a browser because they take up too many resources on the computer. Initial fetch requests using Java libraries didn't pan out because the website uses Cloudflare, which uses TLS fingerprinting to block scraping bots.

I had to get creative. After some research online, I came across a Python library which bypasses this detection somehow. The Python library is called CloudScraper and it worked beautifully! I then had to find a way to work this into my Java code. The easiest option is by far using the Jython library, but I wanted to keep the program to vanilla Java without using external dependencies, so instead, I opted to use Process Builder for this.

The way that this website handles requests is very convoluded because the reviews are loaded inside their own front-end JavaScript logic. I had to spend a lot of time inside the network debugger before finding a useful endpoint to call that would render a proper response. However my struggle was not over, this would only render out the external page and not the reviews because they are loaded using Javascript. Luckily, I stumbled across a POST request to a 'render_portlet' endpoint in the network debugger and realized it made a post request with parameters that I could provide. Finding this helped me significantly because it would return the results as pure HTML which would in turn, allow me to parse for reviews.

From the very beginning, I knew that I had to implement threading into the program because when the assignment stated that I had to scrape every article and I saw that some had thousands of articles, scraping a single page at a time simply would not cut it. I was going to use a random access file to help with the synchronous writing, however, I embraced the challenge of writing a synchronization function myself. In its worst case the program runs in a linear runtime because we have to iterate over N number of pages. Sadly even with the implementation of multithreading, it does not have much bearing on time complexity, it merely serves to improve the performance and speed slightly.

## Program flow walkthrough

The entire program can be summarized in three simple steps: configure parameters, make a web request, and manipulate returned data.

From the beginning of the program, we start by launching into a function that is responsible for grabbing the available topics. This function's goal is to launch an instance of the topics_loader python script where it will make a request to the site and grab all the elements that are present showing available topics. The python script then prints to the standard out stream; the Java function then cleans up the output and stores it. Once this is complete, we ask the user to choose between one of these topics and we have to check to see if this is present in the list.

Our next step is to begin the actual web scraping. The first step was to figure out the number of pages the website yields for reviews because each thread of the program will process each page independently. This is essential because if the topic has a large number of reviews, making a call to each page consecutively would take forever.

From there we jump into taking the chosen topic and encoding special characters because it will be supplanted into a URL dynamically. We then launch the external python script in the same manner that we did with the topics, however this time the python script performs a little differently. This time it will configure a POST request instead of a GET request. This is because this is how the website returns the reviews. If we were to make a GET request we would simply be given a shell of HTML that doesn't contain the reviews that we want, but rather Javascript.

Once we determine how many reviews there are we calculate how many pages there are based on 25 reviews per page. Next, we set all the variables that the threads will need later on such as the number of pages, the path to our python script, the name of the topic, and the standard URL for fetching each page.

With that setup performed, we can now jump into launching our threads for each page. Firstly, the program creates an output file and a byte stream in order to write data to it. The threads then call the external python script. The python script makes the same POST request for each unique page and prints it. Each thread will then read the reviews in a properly formatted manner and write it to our output file but this time they use a synchronized function to write to an output file.

## Additional improvements

There are a couple of improvements that I noticed. Based on the options on the website, we could offer support for multiple languages, offer the option to store files into a database, and download them locally if needed. Another good idea is to fetch a list of keywords that are found in the ariticle's abstract and associate them with each review as additional metadata.