VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

NGUYEN HUU LOC

CRAWLER SYSTEM BASED ON RECORDING USER ACTIVITIES

MASTER THESIS

Major: Computer Science

HA NOI - 2022

VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Nguyen Huu Loc

CRAWLER SYSTEM BASED ON RECORDING USER ACTIVITIES

MASTER THESIS

Major: Computer Science

Supervisor: PhD. Hoang Xuan Tung

HA NOI - 2022

Abstract

In this emerging world of the internet, there is lots of data present and retrieving this data becomes very complicated. As a result, web crawling is one of the important methods of data gathering.

Traditionally, web crawlers usually use CSS selectors, XPath which can be found in the client HTML source code and scrape data from pages with ease.

Today, with the development of state-of-the-art technology, most websites are dynamically created content to prevent web scrapers from getting the HTML source code directly. Additionally, most websites (especially e-commerce websites) are dynamic and hence, a need arises to constantly update the web pages structure which can cause the prior mentioned selectors or XPaths used by scrapers to not work properly.

In this thesis, through the study and research of the Puppeteer framework, a web-based headless crawler system is designed and implemented to adapt to any website changes by recording users' activities to extract HTML data from many E-Commerce websites.

Keywords: *Crawlers, XPaths, CSS Selectors, Dynamic Pages, Website changes, Puppeteer Framework, Record user activities.*

Acknowledgements

Firstly, I would like to thank my advisor, PhD. Hoang Xuan Tung for his support, care, and patience. He has provided helpful comments on various aspects of my thesis. These comments made my thesis much better, and I learned a lot from his comments.

Secondly, I would like to express my deep gratitude to the teachers in the faculty of Information Technology and teachers of University of Engineering and Technology as a whole who have taught me a lot of knowledge and experience during the past 4 years in college.

Finally, I would like to thank my friends and family whose love, support and encouragement helped me improve a lot during my time at the university.

Student

Nguyen Huu Loc

Declaration

I declare that the results presented in this thesis have been carried out by myself under the supervision of PhD. Hoang Xuan Tung.

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institution of tertiary education. I confirm that the work submitted is my own.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices.

Student

Nguyen Huu Loc

Table of Contents

Acronyms

AJAX	Asynchronous JavaScript And XML
API	Application Programming Interface
CRUD	Create Read Update Delete
CSS	Cascading Style Sheets
DBMS	Database Management System
DOM	Document Object Model
ECommerce	Electronic Commerce
HTML	Hyper Text Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
ML	Machine Learning
RESTful	Representational state transfer
SVM	Support Vector Machine
TF-IDF	Term Frequency – Inverse Document Frequency
UI	User Interface

URL	Uniform Resource Locator
XML	Extensible Markup Language
XPath	XML Path Language

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In the modern era of big data and the need for data and information of many varieties, people and organizations alike are going great lengths to gather as much relevant data and information as possible. Therefore, people use web crawling as the solution for data extraction as many individuals and organizations need to consume large amounts of data on the web.

Although the first generation of crawlers could only collect data, modern-day web crawlers are becoming much more powerful apart from data collection, they're also capable of keeping track of web applications for vulnerability and accessibility. There are many other applications in web crawling such as creating large datasets for statistical analysis, data engineering, and machine learning purposes.

Crawlers are also largely used in the field of online shopping, also known as e-commerce. There are many large e-commerce websites for people to choose to buy products from such as Tiki, Shopee, Lazada, etc. As a result, people are at a loss to search and buy the desired product from different websites. In this case, many people use web crawling services to collect and store data which can get a better comparison of the same product or multiple products from the same category.

As today's e-commerce websites are always updated regularly, crawlers have many difficulties in extracting data, namely the problem of structurally changing interfaces. Crawlers use selectors from the website's source code to scrape the data, but the constantly changing interface makes the selectors no longer usable after a certain time. So in this thesis, a crawler system is designed to crawl the details of products by collecting

selectors with the help of user interaction from the interface of the website so that the selector is never outdated.

1.2 Problem statement

In this paper [15], B. Roberts illustrate the development process of a simple crawler. The development process is shown in detail with a figure below.

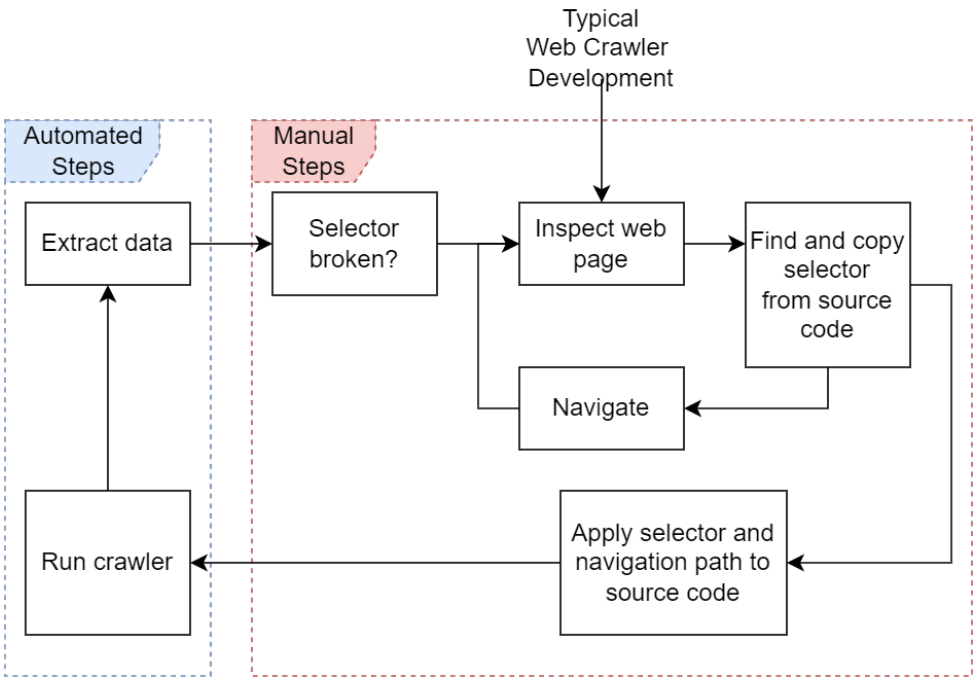

Figure 1.1: Typical Crawler Development Process

First of all, developers must inspect the web page and then choose which data they want to crawl. The Chrome Developers Tool allow developers to look at the HTML element and copy selectors from the source code. Finally, the copied selectors and navigation page are then programmed into a crawler which allow them to run the program automatically and crawl data. Since selectors are hard coded into the source code, typical crawlers built this way are mostly failed after some times whenever selectors are updated . Another disadvantage of the typical crawler development process is every time the website updates that cause selectors to change, developers must update the crawler in the source code since crawlers are hard coded.

To deal with this problem, B. Roberts (2019) proposed an automated crawler system using Machine Learning [15]. The same problem is mention in this paper [14] (2009) where R. Penman presented a custom advance algorithm to deal with multiple aspects of this problem. Website structure change is one of the common problems of

crawlers when scraping data from large sites. From time to time, some websites are subject to structural changes or modifications to provide a better user experience. This may be a real challenge for crawlers, who may have been initially set up for certain designs. Hence, some changes will not allow them to work properly. Even in the case of a minor change, web crawlers need to be reconfigured along with the web page changes.

1.3 Difficulties and challenges

Firstly, this thesis has encountered certain difficulties such as performance issues. Due to a large amount of info on a product page of a large e-commerce website, it took a lot of time for the crawler system to find the selectors and crawl each selector in a single product detail page. As a result, crawling on a large-scale website requires a lot of time.

Secondly, another problem when dealing with crawlers is sometimes selectors have a unique name which makes it easy to differentiate them from the rest. For example, a product detail page in tiki labels an HTML element class name as ".product-title" while some other HTML elements class name is randomly generated such as ".pXksd2". This is one of the strategies to prevent web scrapers mentioned in this paper [8]. This method is usually implemented in many large scale websites where companies want to secure their data.

Thirdly, E-commerce websites, and web pages in general, increasingly rely on dynamically created content using Javascript. Dynamic websites are websites that can update the content after the first HTML load, different from static websites. For example, different from the traditional methods to load a website one page at a time, some websites use dynamically loaded content that can update as the user scroll the website. Since common crawler can only extract static HTML data, dynamically loaded website is a major challenge to crawlers in terms of complexity.

Even though the problems mentioned above are among the common crawler problems, there are many other difficulties when developing a web crawler. However, specifically in the e-commerce domain and in this thesis scope, the above problems are more likely to occur.

1.4 Contribution and thesis structure

This thesis focuses on introducing a new crawler system that can deal with web interface changes. This crawler can only satisfy these conditions by being implementable in

real-world web crawler applications, so they must have a fast crawling time and good performance.

- In this thesis, implementation of a crawler system is introduced and described in detail as another approach for dealing with web interface changes.
- In this thesis, this crawler system is compare to other approaches to determine the quality of the designed crawler system.

The content of the thesis is divided into four chapters and a conclusion:

Chapter 1: **Introduction.** This chapter focuses on the introduction of the background of the problem. It describes the problem in a general way, and it also describes the proposed approaches along with the problems and difficulties of the web crawling problem and the methods for the problem.

Chapter 2: **Related work.** This section focuses on the research directions or approaches related to the web crawling problem, web crawling concepts, softwares and application related to web crawling.

Chapter 3: **Proposed System.** This section introduces and describes the basic theory and design structure of the proposed method in detail. Firstly, it describe the system architecture. Secondly, we will analyze about the system basic flows, its use cases and the system database design. Finally, we will describe in detail about the system crawling algorithm.

Chapter 4: **Implementation and result.** This section shows the implementation of the web crawler such as the user interface and the application programming interface. Moreover, we will present the result and compare it with other approaches.

Chapter 5: **Conclusion.** This section summarizes all the issues of the problem, the proposed methods in the thesis and suggests the future development directions for the problem.

Chapter 2

Related work

2.1 Web Crawling Concepts

In order to solve the problem mentioned in this thesis, we need to understand web crawling concepts. Firstly, definitions will be provided to understand the difference between a web crawler and a web scraper. Secondly, we will discuss about a general website's components such as HTML, CSS, JS, etc. In related to web crawler, some concepts are also provided to have a better understand of web crawling ethics.

2.1.1 Web Crawlers

Web crawler is an automated program that discover URLs and categorize web pages on the internet. In order to categorize web pages, A web crawler will start from some known web pages (also known as seed URLs) and follow hyperlinks from those pages to other pages. Since a website can contains many URLs, this process usually takes a lot of time.

2.1.2 Web Scrapers

In a data extraction problems, we need to combine both web crawling and web scraping. While web crawlers focus on discovering URLs, web scraper are used for extracting data from websites.Web scraping is becoming a popular technique as it allows new startups to quickly obtain large amounts of data. Web scraping can be used for weather reports, products comparison, financial analytics.

2.1.3 Web Components

There are different components and structures in a website. However, a general website always consist of these 4 components: HTML, CSS, Javascript and multimedia resources. There are also DOM, CSS selectors and XPath to provide methods to control the web content.

HTML (Hypertext Markup Language) is a markup language which provide contents and define the structure of a website. Combined with CSS (Cascading Style Sheet), developers can control the way the website is presented such as font colors, elements width, animations, etc. Javascript is a programming language that defines how the website operates from login validation, interactions with elements or API calls to the backend server. Multi-media resource refers to the file, video and images embedded in the website as links which provides more appealing to the website.

2.1.4 CSS Selectors and XPath

CSS Selector [11] is a pattern which allow developers to match with any elements satisfying its requirements. CSS Selector can contain an element's class name, element's id or tag. XPath [12] is a query language for matching elements in an XML Document. XPath contains the path to the exact elements so XPath is very accurate when matching any element on the website.

Figure 2.1: A Website with HTML and CSS

For example, in this HTML source code in **Figure 2.1** from quotes.toscrape website, there are 2 ways to match the element with class "author" with "small" tag in CSS

Selector and XPath:

XPath: /html/body/div[1]/div[2]/div[1]/div[1]/span[2]/small[1]

CSS Selector: div.quote span small.author

Both methods return the same result "Albert Einstein" in this case. While XPath usually give faster results compare to CSS selector, CSS Selector is more human-readable.

2.1.5 Document Object Model (DOM)

DOM [19] is a programming interface which represent an objects comprise the web structure and content of a document. With DOM, developers can interact with pages through nodes and elements. DOM allow programming languages such as Javascript to query elements or nodes using XPath or CSS selector, to change the web page structure or fetch data from those nodes.

2.1.6 Data Extraction Ethics

In this paper [7], V. Krotov et al. address the issues of web scraping posing a serious, legal and ethical challenges. V. Krotov mentioned that while collecting data from the web remains a legal and ethical "grey area", developers needs to understand the process of web scraping like whether the website allow extracting data or how crawling data would affect the website.

R. Mitchell discuss in his paper [10] about various aspect of web scraping legalities like trademarks, copyright laws, trespass to chattels, robots.txt and terms of service, etc.

Trademarks and copyright laws are the laws that protects the website owner's data from being used by others. Even though you can store data for your purpose, the scraped content should not be published without permission. It is also possible to publish statistic of the website content or some example data. When it comes to publishing the website's content, developers need to make sure that there is no violation of this law.

Trespass to Chattels is criteria that makes a web scraper cause damage to the website. For example, web scraper that cause the server to slow down or shut down such as by making too many request is violating this criteria. Essentially, causing harm to the website is forbidden for web scrapers.

Robots.txt and Terms of Services are the website standards that inform which area of the web site can be accessed by automated robots. For example, the figure 2.2 below is the section of Wikipedia 's robots.txt.

```
#
# Friendly, low-speed bots are welcome viewing article pages, but not
# dynamically-generated pages please.
#
# Inktomi's "Slurp" can read a minimum delay between hits; if your
# bot supports such a thing using the 'Crawl-delay' or another
# instruction, please let us know.
#
# There is a special exception for API mobileview to allow dynamic
# mobile web & app views to load section content.
# These views aren't HTTP-cached but use parser cache aggressively
# and don't expose special: pages etc.
#
# Another exception is for REST API documentation, located at
# /api/rest_v1/?doc.
#
User-agent: *
Allow: /w/api.php?action=mobileview&
Allow: /w/load.php?
Allow: /api/rest_v1/?doc
Disallow: /w/
Disallow: /api/
Disallow: /trap/
Disallow: /wiki/Special:
Disallow: /wiki/Spezial:
Disallow: /wiki/Spesial:
Disallow: /wiki/Special%3A
Disallow: /wiki/Spezial%3A
Disallow: /wiki/Spesial%3A
```

Figure 2.2: Wikipedia robots.txt

As in the 2.2 shown, there are basic terms in a robots.txt file: User-agent is the name of the robots, spiders in mentioned (GoogleBot), Allow and Disallow contains the specific parts of the website which the robots have permission to access to. There is also another term which is not mentioned in the figure is Crawl-delay, informing web crawlers how long each time a request must be sent so as not to overload the website server.

2.2 Different Approaches

As mentioned in 1.2, It is common for sites to change their web pages structure. This include changes to the HTML elements containing the data to be extracted which prevent web scrapers from locating data. This problem have resulted in an interest in developing more advance scraping algorithms. Many approaches have been used in this situation such as machine learning algorithms, other advance algorithms.

2.2.1 Advance Algorithm Approaches

In this paper [14], Penman et al. (2009) have identified three ways for a web page can be change: the content changes and structure stays the same, the content stays the same and the structure changes, and both content and structure changes. To deal with these problems, SiteScraper presented a pipeline architecture which comprised of three-step such as parsing, searching, and generalizing data to train models for handling website changes.

SiteScraper also requires a seed HTML document associating with the text content the user wishes to scrape from. After parsing the HTML content, SiteScraper calculate the similarity between parsed content and the seed document using LCS (Longest Common Substring) and build a scoring algorithm base on the output of LCS. The result of the algorithm in [14] has shown great result, which allows SiteScraper to scrape data even after structural changes by automatically retrain models.

Guojun et al. (2017) proposed an intelligent dynamic crawler which extracted data based on XPath selectors and detect website changes based on TF-IDF method to calculate relevance [5]. The result of the algorithm is relatively high when detecting website structural changes, which reduced the maintenance cost for the system in the future.

Khan et al. (2020) presented a web scraper with autocorrection template functionality [6]. When dealing with a website changes, his approach is to identify the same data value in the website and form a new unique identifier which surround the data value.

2.2.2 Machine Learning Approaches

In this paper [1], Victor Carle (2020) presented a scraping algorithm using SVM (Support Vector Machine) which focus on training the model to recognize the HTML source code surrounding the data and its metadata. To extract data from HTML code, both the XPath of the block and the unique name of the node are needed. Its limitations lies at the need for the unique name of HTML node since some websites label HTML nodes as unique and understandable name like "price-container" while others use common name such as "text".

Like Victor Carle (2020), Ujwal et al. (2017) attempt to solve the problem of mutating site structure [16]. Ujwal designed a ML-based scraper system using a Random Forest classifier and relevant data extracted with the use of XPath. His proposed approach is constructing a tree representing an HTML tag in the DOM tree and detect website structural changes in the tree whenever crawled.

2.3 Softwares and Tools

Headless Recorder [3] (2021) is a Chrome extension which allow recording browser interaction such as click or scroll events and then generate Puppeteer and Playwright scripts. The user can see preview of the selectors for elements on hover. Generated Playwright and Puppeteer scripts which later can be rewritten to run scraping algorithms.

Unfortunately, Headless Recorder is partially a scraping application, it does not solve all of our problem. It also have some limitations such as recording on hover because clicking may trigger some unwanted events. On the other hand, our approach is similar to this application by recording user click event to extract element selectors.

WebScraper[1] is a Chrome extension which allow scraping by using point-and-click to scrape thousand of records from a website. By creating a sitemap, it can transverse the website by instructing the scraper to extract data based on its location. The result data is available in JSON, CSV format and can be export to an Excel file. Scrapers can be configured for non-programmers to understand just with a few clicks.

DataMiner[2] is a Google Chrome and Microsoft Edge browser extension that helps scrape data from web pages and into a CSV file or Excel spreadsheet. It provide users with a simple UI which helps execute data extraction and web crawling.

2.4 Frameworks and Libraries

Puppeteer [4] is a popular and powerful Node library which provides a high-level API to control headless browser such as Chrome or Chromium via Chrome DevTools. Puppeteer allows developers to scrape websites by opening a Chromium browser and extracting data with CSS selectors or XPath. Puppeteer also have some limitations such as it only supports on Chrome browser. Since it is a browser control tool which primarily focus on a particular set of control structure, it can supports only Javascript. Within this project, using Puppeteer is preferred because it can supports both AJAX and dynamic pages in order to extract data from the web.

[1] `https://webscraper.io/`
[2] `https://dataminer.io/`

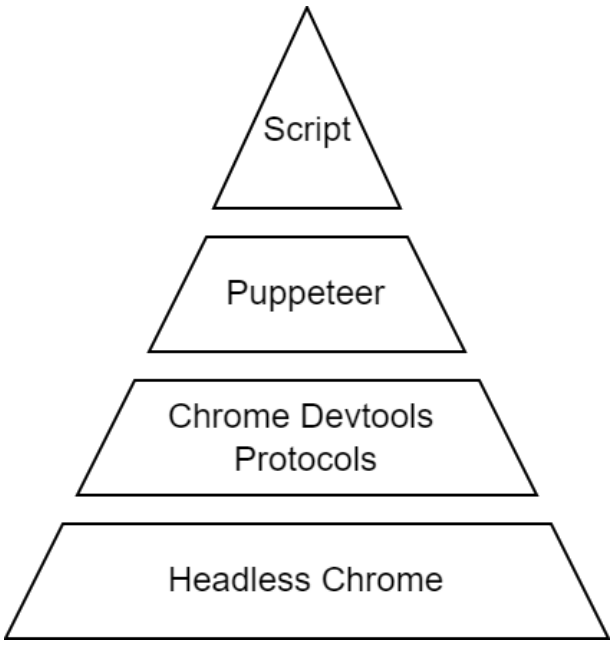

Figure 2.3: Puppeteer Architecture

This **Figure 2.3** presents the Puppeteer's architecture which is consist of 4 layers:

- Headless Chrome: is the browser level where the actions is performed in the browser engine.
- Chrome Devtools Protocols: allows external user to interact with the browser.
- Puppeteer: is the node module package which provide Javascript API to interact with the browser using Chrome Devtools.
- Scripts: is the NodeJS script where developers can program to interact with browser through Puppeteer.

Another way to automate browser or handle javascript generated content without using Puppeteer is by using Selenium. Selenium [18] is a set of tools which has capabilities to control on different browsers such as FireFox, Chrome, Edge, etc. It also supports multiple programming languages such as Javascript, Python, etc. In order to create a scraper from Selenium, it is required to download the latest webdriver on the web from which you want to extract data. Its core difference from Puppeteer is that Selenium is a web-UI testing software and Puppeteer is a browser-handling framework.

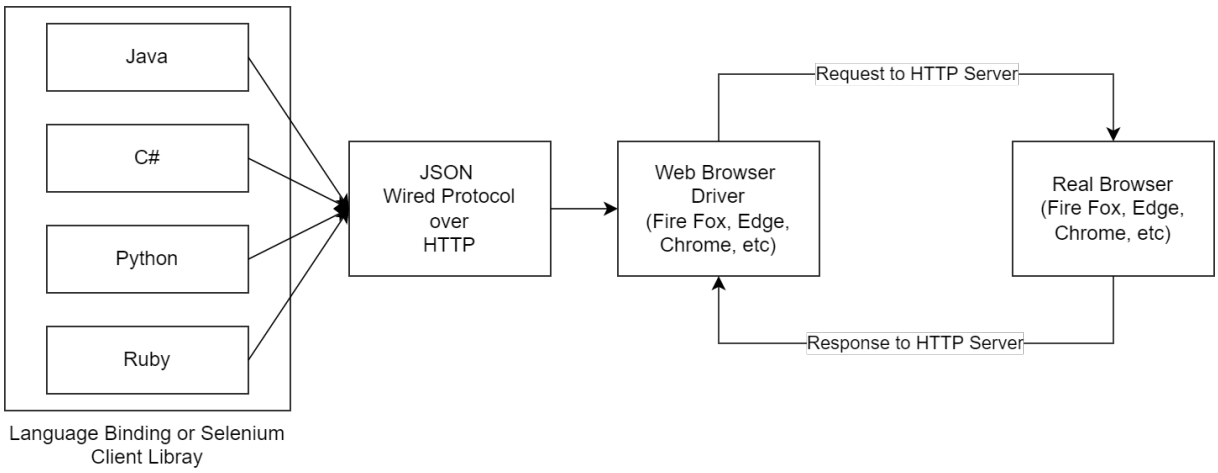

Figure 2.4: Selenium Architecture

Selenium architecture consists of 4 layers: the Selenium Client Library, JSON Wire Protocol, Web Browser Drivers and Browsers.

- Selenium Client Library: is Selenium Library that can be used in many programming languages such as Python, C#, Java, etc. The Selenium code will be converted into JSON format.
- JSON Wire Protocol: allows the transfer of data between HTTP servers.
- Each browser have their own web driver. Web drivers translate JSON received from the client library and then execute the commands on the browser. The response is returned through HTTP response.

Scrapy [17] is a popular Python web-scraping framework which has many variety of applications such as data mining, information processing. Even though its original purpose is to be a web scraping framework, it can also be used for crawling data through API. When it comes to extracting data, it has the fastest performance compare to other frameworks due to its ability to handle asynchronous request. Since Scrapy is not a browser-handling framework, it cannot deal with dynamic generated content or execute javascript.

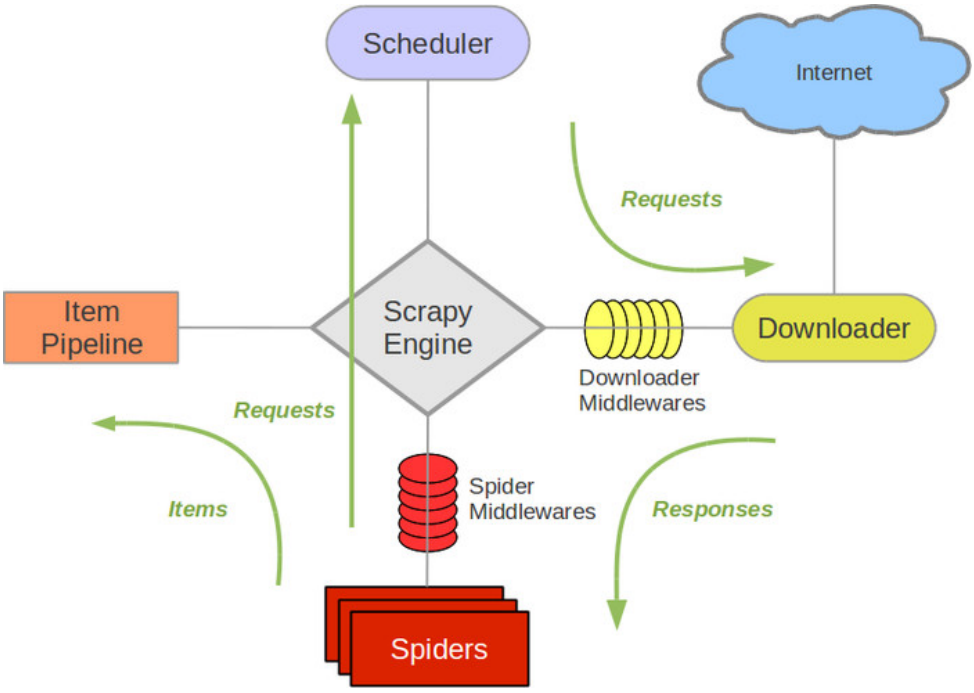

Figure 2.5: Scrapy Architecture

The figure above is the Scrapy architecture [17], consist of 5 components: Pipeline, Scheduler, Downloader, Spiders and Scrapy Engine.

- Scrapy Engine: is the central component which control the data flow in the architecture
- Downloader: is responsible for fetching web pages and send the data to Spiders through the Engine.
- Spiders: are programmed by developers and responsible for extracting content and handling request.
- Item Pipeline: is the component that handle the extracted content from Spiders to validate and check the data before saving to the database.
- Scheduler: handle requests to be saved into a queue and used later whenever receive a request.

Chapter 3

Proposed System

This thesis research on solving the problem of structural changing website in crawlers. In this chapter, we present a design of the system which shows our approach towards dealing with this problem. In section 3.1, an overview of the system is introduced in which we discuss about our proposed system architecture and its components. Section 3.2 will be about its requirements specification. In section 3.3, we will discuss about the crawler system design such as the UI design, API design and Database design.

As mentioned in 1.2, our approach to dealing with problems is first to use the crawler to scrape the selectors containing the data we want to extract and then use those selectors to crawl the data applied on every URLs on the same ECommerce site. In this thesis, we used the framework Puppeteer to create an automated browser in which we can record the user click-and-point events and then extract the selectors from where the user click.

3.1 System Architecture

Our proposed system structure is based on the following component: *a) a VueJS frontend, b) a NodeJs Backend*, *c) Puppeteer Framework* and *d) MongoDB Database*. The whole system structure shows in detail in **Figure 3.1**: a VueJs frontend that display data and make request to the server, a NodeJs API to access to database and run crawling request, Puppeteer framework that initiate Chrome/Chromium browser in which we can apply the Javascript function to control the browser. Finally, choosing the correct DBMS (Database Management System) is important for the crawler performance.

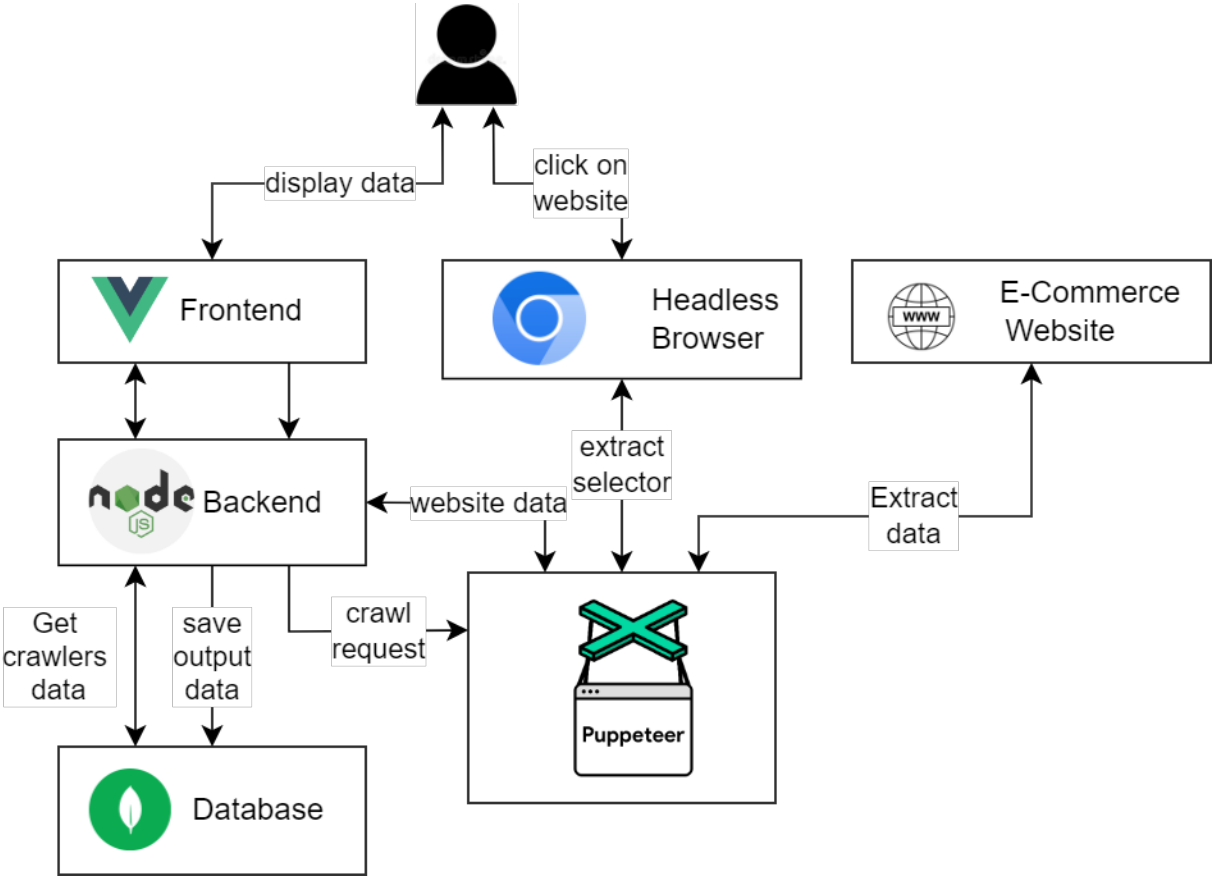

Figure 3.1: Proposed architecture for Crawler System

The benefits of our proposed system is that it provides with a simple UI for users to use. While most crawler system are programmed by hard coding, with this proposed system, users can build a crawler with some simple click from the users. This system's flexibility allow users to make changes to the crawler quickly. With the use of a Headless Browser to record the user click event and extract selectors from it, the accurate data can be extracted from the website. The proposed system main functionality is the "recording user activities", that is inspired from the Headless Recorder system [3]. While the Headless recorder system basic functionality is to record the user activities such as click event, scroll event or hover event to generate a Puppeteer or Playwright script, our approach is to record the user click event and extract selector data from the web page.

3.1.1 Frontend (Client Side)

In the **Figure 3.1**, our proposed system has a simple UI dashboard to display all the crawlers and a detail form of the crawler information. The dashboard allows users to manage the crawler data and observe its performance and statistics.

The system also provides users with a detail crawler form where users can update the detail information of the crawler and save its in the database. Additionally, users can make API request to the backend server that allow them to either extract selectors or extract data from a website.

Before making a crawl request, the users requires a list of URL(s) to extract data from. Considering most website use pagination that contains list of data in separate pages, our proposed UI also provides with a utility page that can generate paginated UI and export in CSV format.

3.1.2 Backend (Server Side)

The backend contains API that deals with CRUD operation for a crawler entity: fetch all crawlers, fetch one crawler, add new crawler, update and delete a crawler. The backend API also validates a crawler information , processes data crawling requests and selectors extraction requests.

After the server receive a request to either crawl data or extract selectors from the detail form, the server will query the crawler that exist in the database and use the information from the crawler to run Puppeteer framework.

When crawling data from the website, API also provide users with an additional parameter that allows crawler to extract data using either XPath or CSS selector. Users can configure the parameter on the crawler form UI by clicking on "XPath" or "CSS".

3.1.3 Puppeteer Framework

The main function of the proposed system is used in this Puppeteer framework section called "recording users activities". The framework used in this approach has 2 operations: Recording and Scraping.

When API receive a selector extraction request from UI, it run the Recording operation and automatically launch an instance of Chrome Browser and navigate to URL. In the URL page, before the document element is created and any script are run, the Browser run a Javascript programs which emits the user click events. Afterwards, each time the user click the website element, we can extract XPath and CSS selectors by transversing the parent of the current node in the DOM.

The Scraping operation is to crawl data using the selectors from the Recording operation. It requires a list of URLs and selectors as input data and then Puppeteer will

run though each URL to extract data. The data output of the Scraping operation will be saved in the MongoDB database.

3.1.4 MongoDB Database

There are many types of web pages such as paginated pages containing links of a product, pages containing detail information of a product or a single page. This leads to the output data of the crawler will be unstructured and disorganized, making it hard to save data in a SQL database.

As a NoSQL database, MongoDB is flexible when used to store data since the data is saved in JSON format. MongoDB makes it easy for developers to store structured or unstructured data. In our proposed system, MongoDB is used to store the crawler's information, extracted selectors and the crawler's output data.

A crawler's information contains a code, name, description and a list of URLs. To be able to query and store data into the crawler, each crawler needs to have its own unique code. When the user makes an API request to fetch a crawler, in addition to crawler information, selectors data and crawler output data are also returned and displayed on the UI.

Extracted selectors contains both XPath and CSS selector result. After extracting selectors are done, users needs to update the metadata of the selectors, meanings the field of data that a selector represent.

Finally, after each Scraping operation, the output data will be saved in the database with the crawler code as a mock foreign key to the crawler's information.

3.2 Requirements Specification

3.2.1 Overview

The system would be developed as a web-based application. Its objective is to create a flexible crawler system so that users create and update their own crawler The crawler is designed to be able extract selectors and data through the user's interaction on the UI. As a web application, the system aims to create an simple UI for users to understand. With this system's main function is to record user click events, the system expects to devise a different solution to the website structure changing problems in web crawlers.

When dealing with the updating website, the system is expected to update the se-

lectors without the need to change the source code. The proposed system approach is to use the crawler to scrape the necessary selectors containing the information needed to extract. Afterward, use those selectors to crawl the details data on the website.

In order to have a better understanding of the system, a basic workflow of the crawler system is presented in the **Figure 3.2** below. The workflow of the proposed system has 3 basic steps: create a crawler, start selector extraction and crawl data.

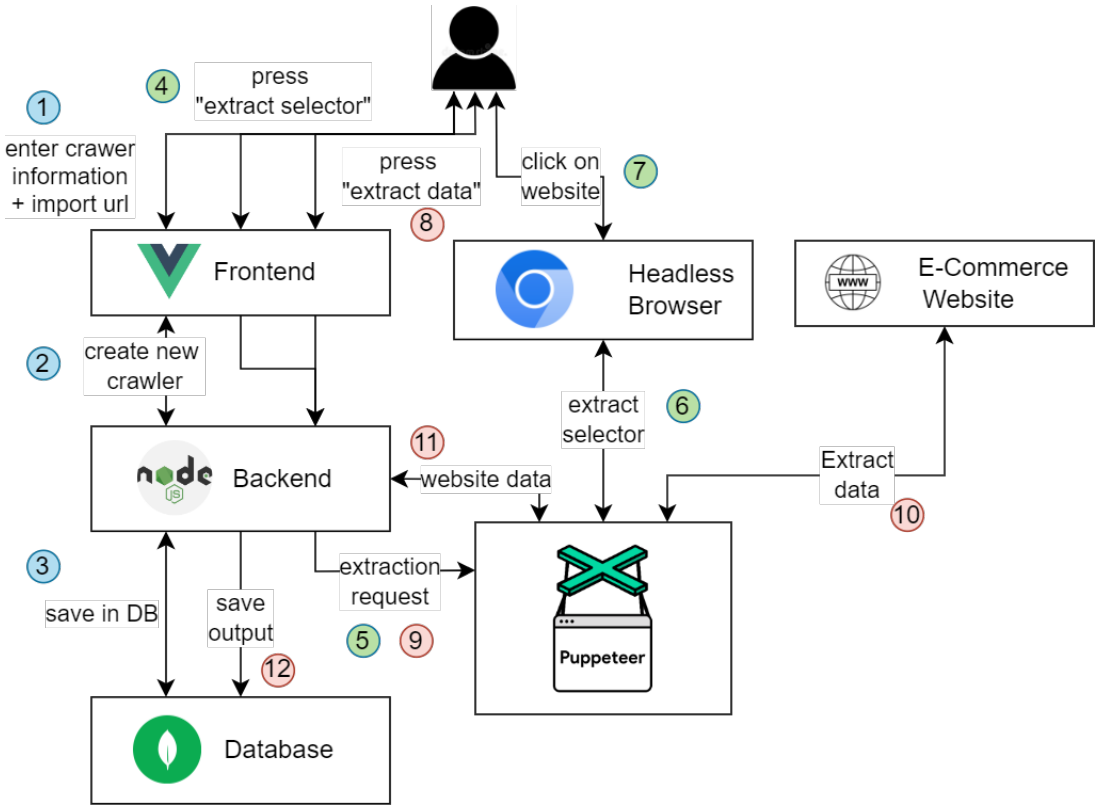

Figure 3.2: Basic workflow of the Crawler System

First of all, user needs to enter its input such as the name of the crawler, crawler code and description. The crawler also needs to have a list or URL(s) as .CSV format from the website as its input. In case we need to crawl from a list of pagination pages, the UI provides user with a utility page to generate URL(s) with pagination. After all the crawler's input are entered, the user save the crawler information into the database.

Secondly, A crawler needs to have a list of selectors to be able to locate data from the website. On the detail form of the crawler UI, users can press a button to start the selector extraction process. User send a request to the API where it will initiate a Puppeteer script. Afterwards, the Puppeteer script will open an instance of the headless browser and inject the event listener script, where it will record the user click event. The recording process is presented in section 3.1.3. Every time the user click on an element

of the browser, the event listener returns the current node in the DOM and generate the CSS and XPath selector from that node. In order to finish the recording process, the user can close the headless browser and the result selectors are displayed on the UI.

Finally, the crawler have both the URLs and the selectors as its base input. Users can run the data extraction process by pressing the button "Extract data" on the UI. The Puppeteer framework will locate the data on the web using the selector and extract data from every URL. The result output of the crawler will be saved in the database and displayed to be exported in CSV file on the UI.

3.2.2 System Requirements

3.2.2.1 Functional Requirements

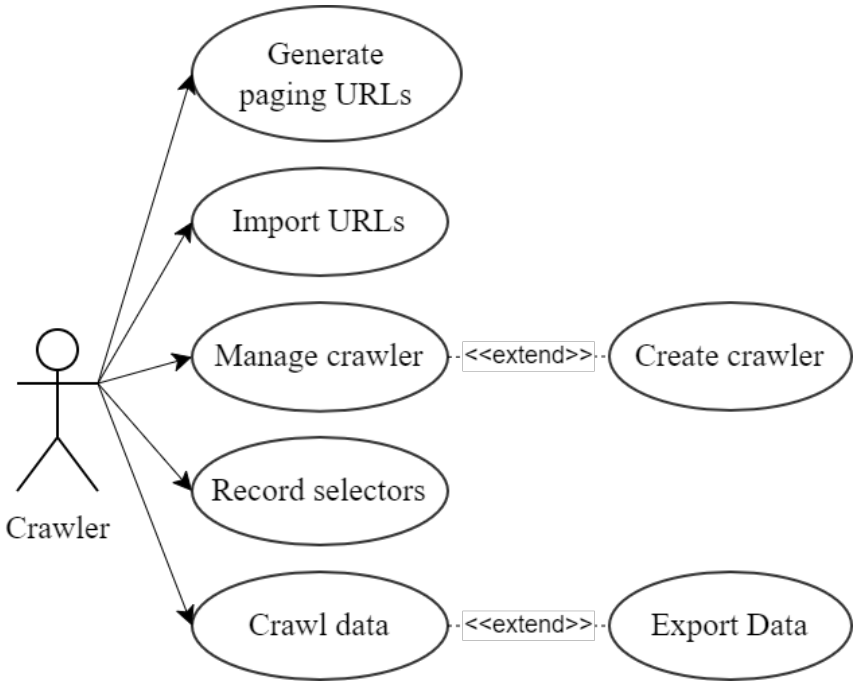

Figure 3.3: Use case diagram of the Crawler System

The following main features are included in the crawler system:

- ***Import URL(s):*** In order to enter a list of URL(s) in the crawler as input, the system allows users to import a list of URL(s) in CSV format.
- ***Generate Paginated URL(s):*** In order to have a set of URL(s) with pagination, the system allows users to generate a list of URLs.

- ***Manage Crawler:*** In order to manage the detail of a crawler, the system allows users to update the setting or delete of existing crawlers.
- ***Create Crawler:*** The system allows users to create crawlers in the system.
- ***Extract Selectors:*** In order for the crawler to locate the data, the system allows users to choose which data selectors extracted from the targeted URL.
- ***Crawl Data:*** The system allows users to run crawlers to extract data.
- ***Export Data:*** The system allows users to export the output data of crawlers in CSV format.

3.2.2.2 Non-functional Requirements

In this section, we will define the evaluation method on the crawler performance in different websites. Since the crawler application is mainly used in statistics, analysis or datasets for machine learning projects, it is common for crawlers to extract data in a large scale website. As a result, it is essential for crawlers to be as fast and accurate as possible.

In this paper [13], O. Lloyd et al. (2019) proposed a scraper system using Selenium Webdriver technology. In order to evaluate the system, O. Lloyd compare the accuracy between the data in the website and the extracted data. O. Lloyd also mentioned that evaluating the speed of the crawler in dynamic website also depends on many variables such as the local network capacity, the server network capacity and the distance between the both. Consequently, it is harder to evaluate the time completion of the crawler. Nonetheless, in this thesis, we will evaluate the speed of the crawler based on the average time for a crawler to finish a page. Additionally, we will calculate the accuracy of the data based on the same evaluation method as O. Lloyd in this paper [13].

The system must have the following non-functional requirements: The criteria for the crawler system are performance, data integrity and maintainability. Users usually requires crawler system to have high performance so as not to waste much time trying to get data. Additionally, data extracted by crawlers must be accurate. Finally, The crawler system are also focusing on maintainability since our approach is dealing with structure changes.

Crawling Time Performance			
The average time for a crawler to finish extracting data per URL.			
Time Criteria (s)	Outstanding	Average	Minimum
	0.9s	1.0s - 1.4s	1.5s

Table 3.1: Performance Requirement for the Crawler System

Data Integrity			
The percentage of data that is extracted accurately after crawler are done.			
Percentage Criteria (%)	Outstanding	Average	Minimum
	100%	95%	90%

Table 3.2: Data Integrity Requirement for the Crawler System

3.3 System Design

In this section, we will discuss in detail about the use case mentioned in section 3.2.2.1. Each use case will be described and provided with a sequence diagram to understand the process of the use case.

Moreover, the database design will be shown to give a better understanding to the crawler system data. In the database design, we will present our tables structure in the MongoDB data and its functionality. The entity tables will also be provided to describe the data columns and data types in the system.

Finally, we present our crawling algorithm that allow our application to extract selectors and data from websites.

3.3.1 Use case Analysis

3.3.1.1 Import URLs

Usecase #	UC1
System	Crawler System
Brief Description	The system allows users to import list of URLs in CSV format.
Trigger	Users want to crawl from a list of URL(s)
Basic Flow	- Choose a CSV file from your local computer. - Check if URLs are ok for crawling data. - Save URLs for Crawler configuration.
Precondition	Users are at Crawler Configuration Page.
Postcondition	None.

Table 3.3: Import URL(s) use case table

Sequence Diagram:

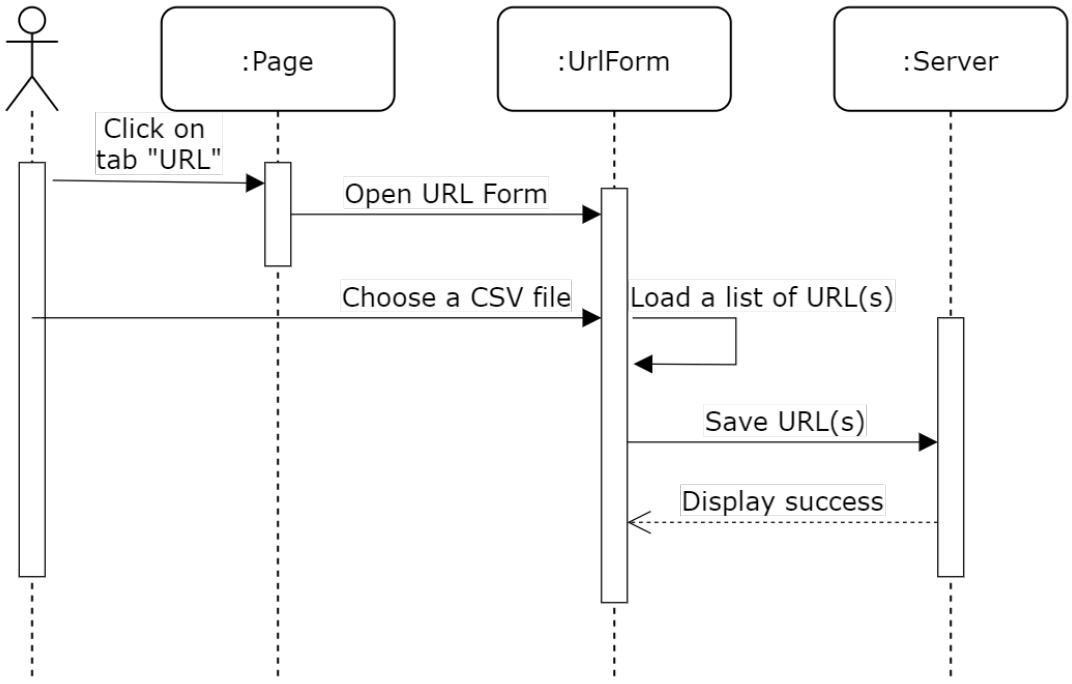

Figure 3.4: Import URL(s) Sequence Diagram

3.3.1.2 Generate Paginated URLs

Usecase #	UC2
System	Crawler System
Brief Description	The system allows users to generate a set of URLs with paging number.
Trigger	Users wants to quickly import a new set of URLs
Basic Flow	- Enter a seed URL (Ex: https://quotes.toscrape.com). - Enter the paging parameter, first page and last page number (Ex: /page/number - 1 - 10) . - Generate URLs and export as CSV file.
Precondition	Users are at URL Tool Page.
Postcondition	None.

Table 3.4: Generate Paginated URL(s) use case table

Sequence Diagram:

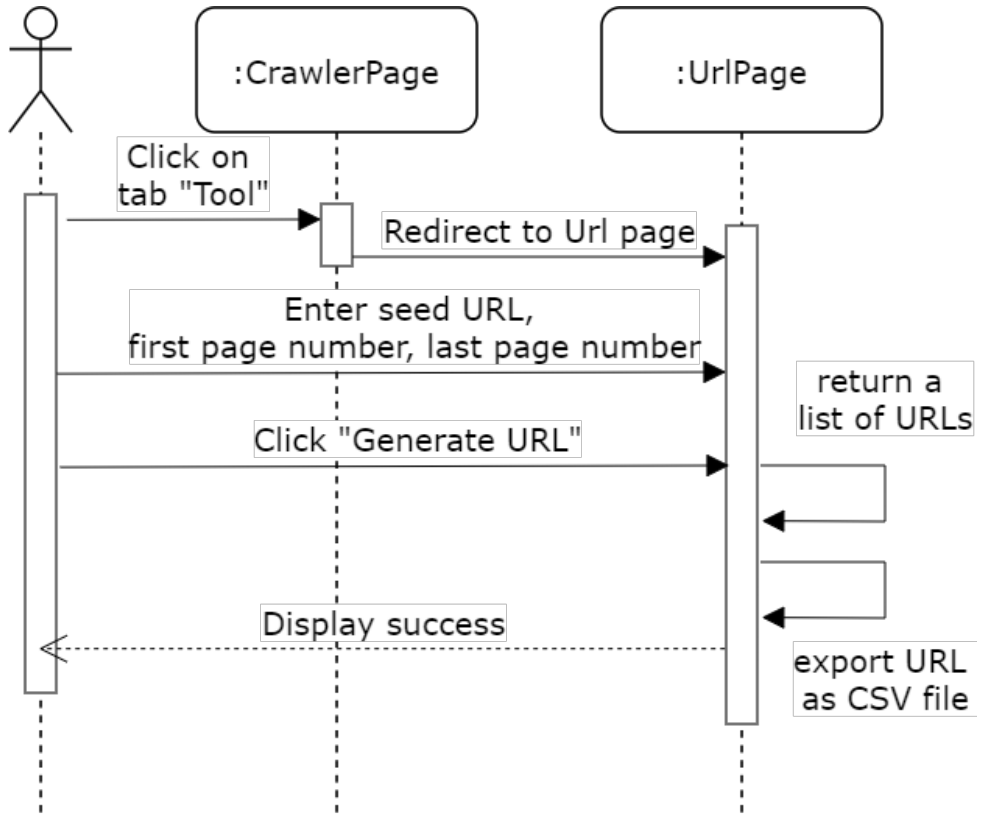

Figure 3.5: Generate Paginated URL(s) Sequence Diagram

3.3.1.3 Create Crawlers

Usecase #	UC3
System	Crawler System
Brief Description	The system allows users to create crawlers.
Trigger	Users want to create crawler for data extraction
Basic Flow	- Enter required fields for Crawler: Code, Name, Description, etc. - Check if Crawler Code is exist in this system. - Display error if exist. - Save Crawler configuration.
Precondition	Users must already added URL(s) to the crawler configuration.
Postcondition	None.

Table 3.5: Create Crawlers use case table

Sequence Diagram:

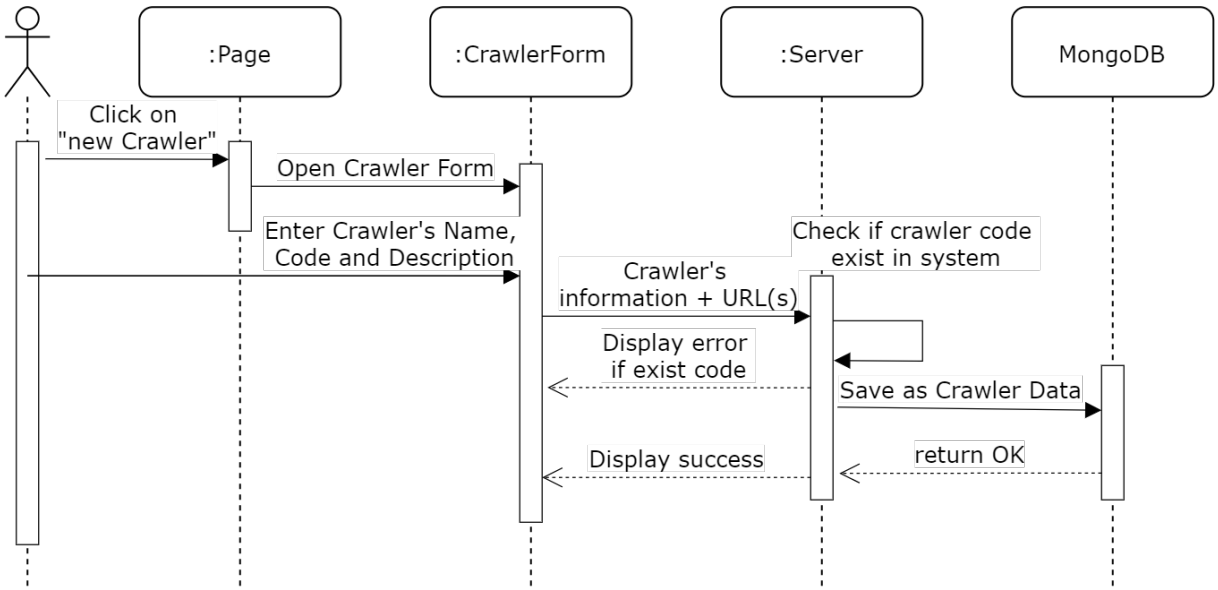

Figure 3.6: Create crawler Sequence Diagram

3.3.1.4 Manage Crawlers

Usecase #	UC4
System	Crawler System
Brief Description	The system allows users to update crawlers.
Trigger	Users want to update crawler.
Basic Flow	- Get crawler by entering its code. - Display error if Crawler does not exist in this system. - Users can update the crawler information such as import a new set of URLs. - Save Crawler configuration.
Precondition	Users must already have a crawler in the system.
Postcondition	None.

Table 3.6: Manage Crawlers use case table

Sequence Diagram:

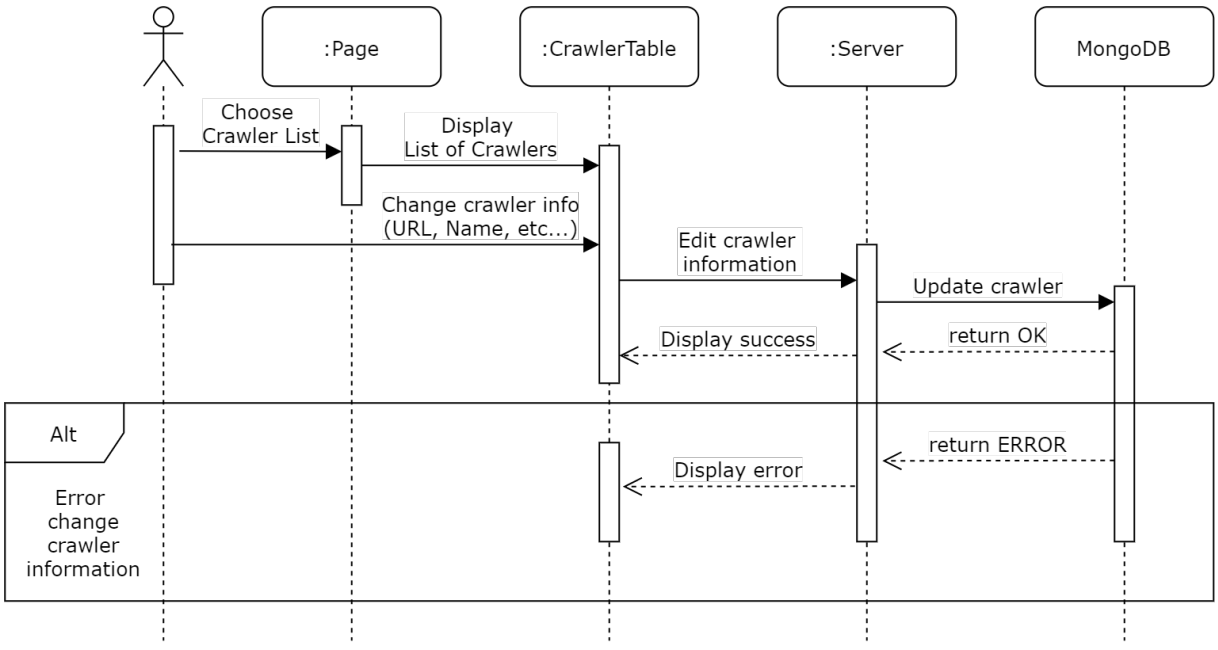

Figure 3.7: Manage crawler Sequence Diagram

3.3.1.5 Extract Selectors

Usecase #	UC5
System	Crawler System
Brief Description	The system allows crawlers to extract selectors from URL.
Trigger	Users want to choose which data to be extracted from the website.
Basic Flow	- Recorder are run and go to the first default URL in Chromium Browser. - Users choose which data's selectors in the page get extracted. - Close the browser after users are done. - Enter each selectors field name according to its meaning. - Save Crawler Configuration.
Precondition	Users must already added URL(s) to the crawler and created a crawler in configuration.
Postcondition	None.

Table 3.7: Extract selectors use case table

Sequence Diagram:

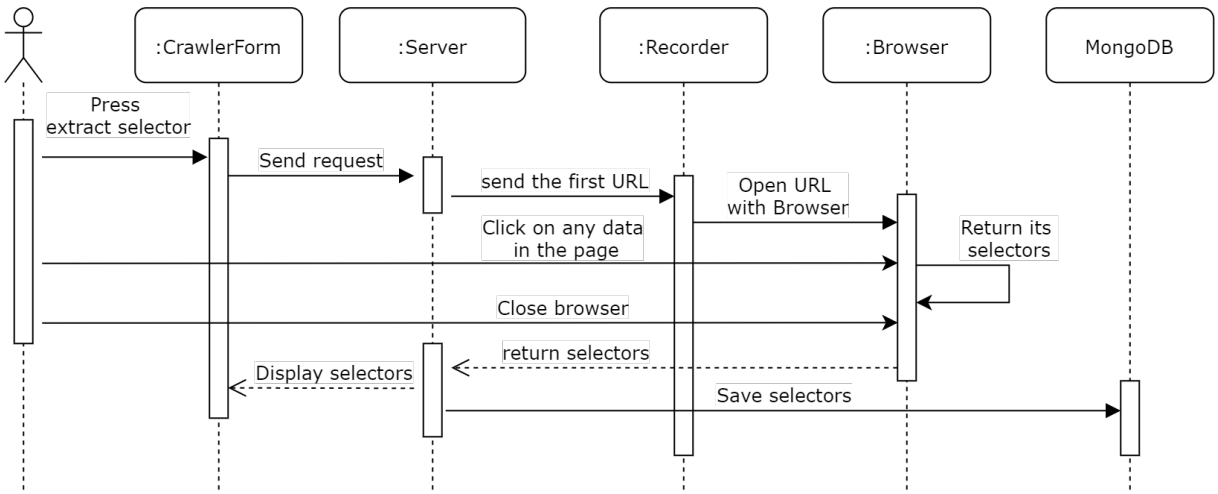

Figure 3.8: Extract Selectors Sequence Diagram

3.3.1.6 Crawl Data

Usecase #	UC6
System	Crawler System
Brief Description	The system allows crawlers to extract data from targeted URL(s) and selectors.
Trigger	Users want to extract data from the website.
Basic Flow	- Crawler are run and go to targeted URL(s) in Chromium Browser. - Crawler extracts data according to the chosen selectors. - Continue to the next URL and repeat. - The crawler output are display in UI (User Interface).
Precondition	Users must already added URL(s) to the crawler, created a crawler and had selectors in the configuration.
Postcondition	None.

Table 3.8: Crawl data use case table

Sequence Diagram:

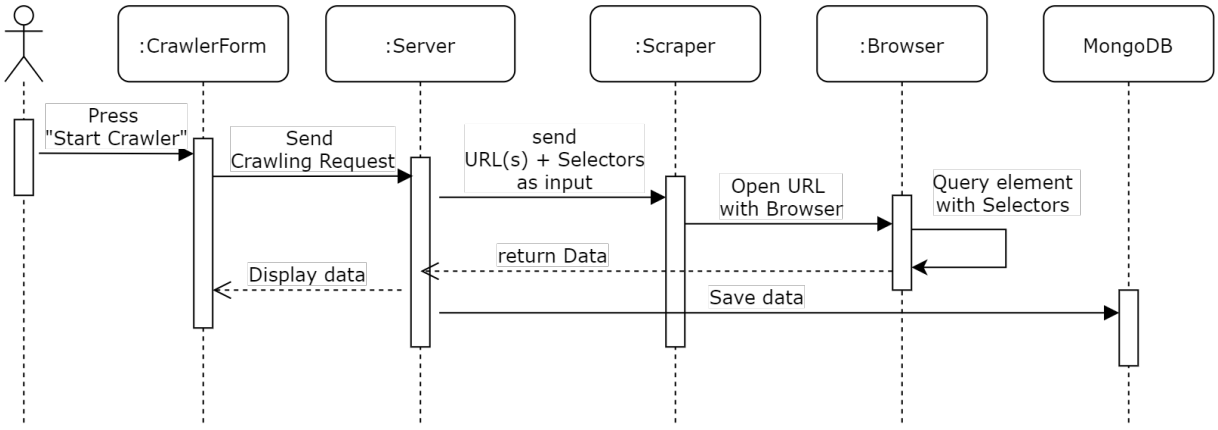

Figure 3.9: Crawl Data Sequence Diagram

3.3.1.7 Export Data

Usecase #	UC7
System	Crawler System
Brief Description	The system allows data extracted by crawler to be exported in CSV or JSON format
Trigger	Users want to save the data.
Basic Flow	- Users check if the output data are accurate. - Export data in CSV or JSON format.
Precondition	Users have already run crawler and the output are shown on the web.
Postcondition	None.

Table 3.9: Export data use case table

Sequence Diagram:

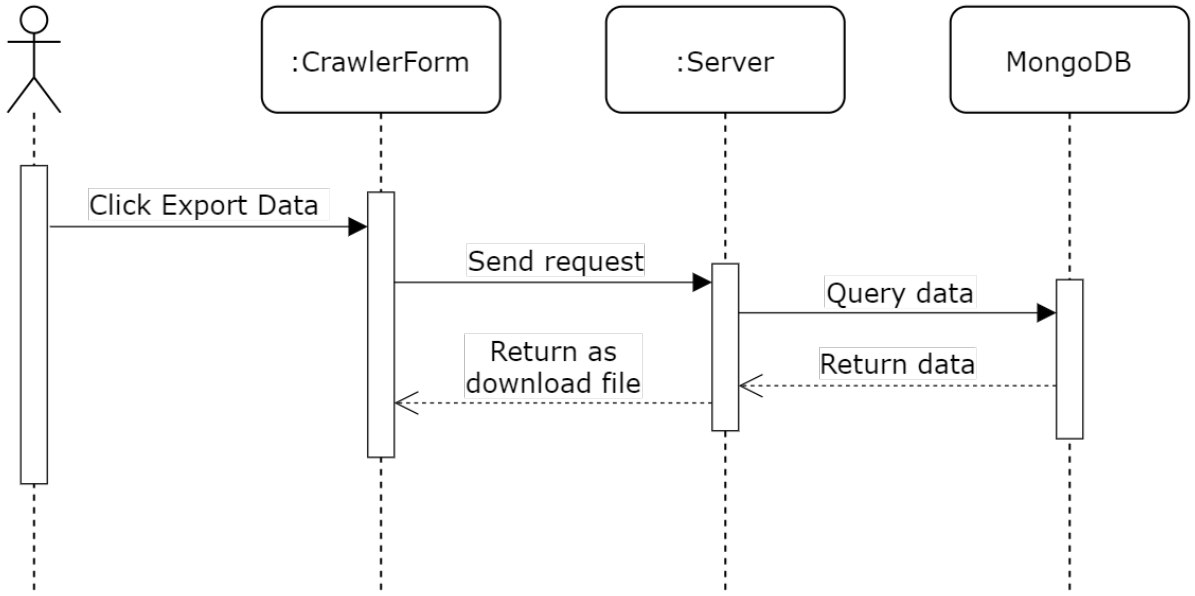

Figure 3.10: Export data Sequence Diagram

3.3.2 Database Design

3.3.2.1 Database Structure

The database consists of three tables. The main tables are the Crawlers, Selectors and Data. Since we need to identify the selectors and data belongs to which crawler and MongoDB is a NoSQL database, table Selectors and Data use field *crawlerCode* as a mock foreign key to table Crawlers.

Database Table	**Description**
Crawlers	Contains all of crawler's information.
Selectors	Contains all of selectors's data.
Data	Contains all of crawlers's result output.

Table 3.10: Crawler System Database Structure

3.3.2.2 Functionality

The database and web application allow developers to complete the following functions:

- Add a new crawler into database.
- Modifying an existing crawler information.
- Add new selectors into database.
- Modifying existing selectors information.
- Add new data extracted by crawler into database.

3.3.2.3 Entity Table

The Entity Tables describe all the data and data types of different entities in the crawler system.

Entity	Column	Type	Required	Description	Example
Crawlers	id	String	true	Crawler's id	62bffbb11045278faa338887
	crawlerCode	String	true	Code for crawlers	CRWL0001
	crawlerName	String	true	Crawler's name	Books Crawler
	description	String	false	More information about the crawler	"This is an example crawler."
	selectorType	String	true	selector type used to scrape from website	XPath / CSS
	urls	Array	true	URL(s)	[example.com /page-1, example.com /page-2, example.com /page-3]
	performance	String	false	Total crawling time in second	"11.32"
	modifiedDate	Date	false	The date which the crawler was updated.	20-06-2022

Table 3.11: Crawler Entity Table

Entity	Column	Type	Required	Description	Example
Selectors	id	String	true	selector's id	62bffbb11045278faa338887
	index	Int	false	selector's position in list	1
	crawlerCode	String	true	mock foreign key to table crawler	CRWL0001
	metadata	String	true	field name for this selector	Book Title
	tag	String	false	type of DOM element	DIV
	selectorXPath	String	true	selector type	/HTML[1] /BODY[1] /DIV[1] /DIV[1]
	selectorCSS	Array	true	selector type	article .product_pod >h3 >div
	eventType	Date	false	how data was recorded	"click"

Table 3.12: Selector Entity Table

Entity	Column	Type	Required	Description	Example
Data	id	String	true	data's id	62bffbb110 45278f aa338887
	crawlerCode	String	true	mock foreign key to table crawler	CRWL0001

	data	Array	true	the output of crawler	[
					{
					title: "In Her Wake",
					price: 52.37
					},
					{
					title:"Wimpy Kid"
					price: 24.99
					}
					]

Table 3.13: Data Entity Table

3.4 Crawling Algorithm

This section describe the algorithm to extracting the selectors and data from any website in some details. The algorithm main purpose is to solve the problem of the webs changing structure. Firstly, we will demonstrate our selector extraction algorithm. Afterwards, we will talk about our data crawling algorithm in detail.

3.4.1 Extracting Selectors

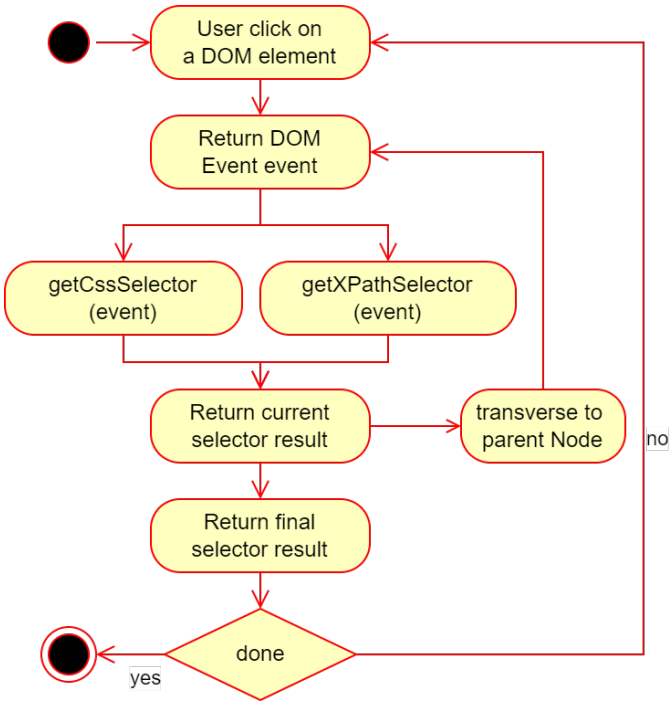

Figure 3.11: Extract selector algorithm

This **Figure 3.11** is an overview of how to extract selector from any DOM elements. By making use of the DOM event in the browser, developers can create an click event listener and then return the target node of the data that the user clicks through the property "event.target" of the DOM event. From the target node, we can get the node property such as its class name, unique id, element tag and create a selector. We also need to get the selector of the parent node since the target node selector may not be enough for the DOM to recognize. To transverse to the parent node, we use the property of the DOM element "parentNode". We will extract both the CSS selector and XPath selector from the target node.

CSS selector is highly accurate when locating the element in the DOM. Each element object represent as an HTML element, can use the properties of "className", "id", "tagName". With these 3 properties, a CSS selector can be created. In the example website in section 2.1.4, to get the data contain author's name "Albert Einstein", we create a selector in the current node "small.author". Then visit the parent node and get selector "span", repeat and get selector "div.quote". The final selector result is merge into "div.quote span small.author". CSS selector is simple to locate in the DOM element so we will not need to visit every parent node.

In a website, it is possible to have dynamic CSS selectors in any HTML element, meanings a randomly generated selector such as "div.pXksd2". These day, it is not uncommon for websites to have dynamic selectors. That's why we also need XPath selector since it can access almost any element without the class name or any id. XPath contains the path and position from the current node to the HTML body. To create XPath selector, we use the current node's tag and its position. Using the same example mentioned above, we have "small[1]" since the current node is a <small> tag and placed at position 1 compare. We repeat these steps at the parent nodes until we reach the html body. Finally, we have an XPath selector "/html/body/div[1]/div[2]/div[1]/div[1]/span[2]/small[1]".

As a result, we get both the CSS selector and XPath selector to extract data from any websites. The result selector data is like in the example of **table 3.12**

3.4.2 Extracting Data

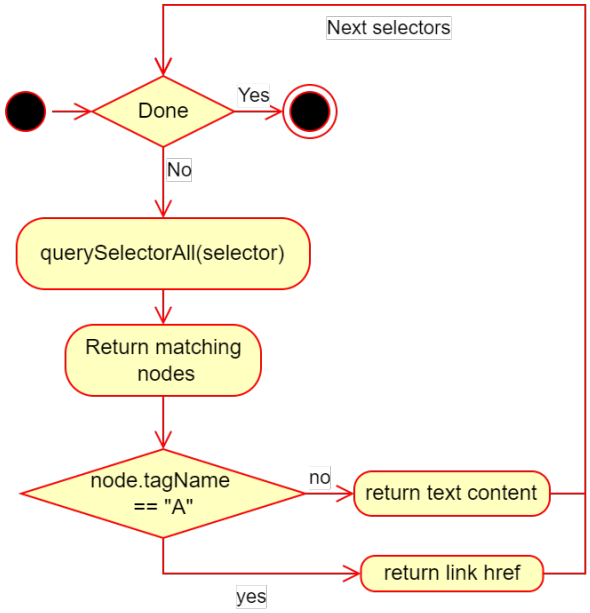

Figure 3.12: Extract data algorithm

Even though data extraction use case has been mentioned in **table 3.8**, we have not mentioned in detail about its crawling algorithm. In each URL, we perform the same data extraction method. First, with each selector, we use the DOM method to query all results that match the selector. If the resulting nodes contain the <a> tag, meaning that is a link, the DOM will get the link instead of the text content. The result data will be pushed into a list and continue to the next selector. This action will repeat until there are no more selectors. Afterwards, the browser will continue to the next URL in line. As a

result, the data will save into the database and displayed in the crawler form UI.

One of the problem worth mentioning in this crawler algorithm is performance issues. In order to extract data from dynamic pages, our system use headless browser via the Puppeteer framework. The problem is mentioned in this paper [9] by P. Meschenmoser, that the use of headless browser can reduce crawler performance. Although using a Headless browser can solve the problem of dynamic content, it takes longer to complete extracting a page compared to a typical crawler.

Performance issues also occur when the crawler is run in complex web pages with a lot of resources such as images, stylesheets. The more resources used in the site, the longer it takes for the crawler to scrape the data.

Chapter 4

Implementation

4.1 System Environment

4.1.1 Platform Enviroment

The system in this thesis is implemented using Javascript programming language, NodeJs version 16.14.2, MongoDB version 5.0.9 and Puppeteer framework. The web application is builed based on platform:

- **Frontend:** VueJs.
- **Backend:** NodeJs, ExpressJs, Puppeteer.
- **Database:** MongoDB.

4.1.2 Hardware Enviroment

The crawler system is executed in the Window 10 operating system. The hardware is listed as below:

- Processor: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz 2.11 GHz
- RAM: 12.0 GB
- System type: 64-bit operating system, x64-based processor

4.1.3 Data Source

Website	Data per page	Page amount	Data amount
https://quotes.toscrape.com/	10	10	100
https://books.toscrape.com/	20	50	1000
https://tiki.vn/dien-thoai -may-tinh-bang/c1789	48	15	720
https://laptop88.vn/may -tinh-xach-tay.html	40	10	400

Table 4.1: Different Websites Source for crawling

The websites used for our crawler system is consist of 4 websites:

- **quotes.toscrape.com** [1] is a popular website used by developers who wants to practice data extraction problems. This website contains hundreds of inspirational quotes, their authors with different subjects taken from *goodreads.com/quotes*.
- **books.toscrape.com** [2] is another popular example website used by data crawling practitioner. This website contains a larger amount of books data, which helps with identifying our crawler performance and accuracy.
- **tiki.vn** [3] is a huge Vietnamese e-commerce website, contains many types of product for consumers to buy from. In this website, we will be crawling tablets and electronic phones products.
- **laptop88.vn** [4] is a Vietnamese e-commerce website that focus on laptop products.

[1] `http://quotes.toscrape.com/`
[2] `https://books.toscrape.com/index.html`
[3] `https://tiki.vn/`
[4] `https://laptop88.vn/`

4.2 System Interface

4.2.1 User Interface (UI)

4.2.1.1 Crawler Dashboard

This page displays all the basic information of all the crawlers in the system. The dashboard display the statistic in the system: total URL(s) needs to be crawled, total amount of data extracted and average performance time it takes for a crawler to finish. In this dashboard, users can create a new crawler by clicking on "Create new". Users can also watch, update a crawler by clicking on "Details".

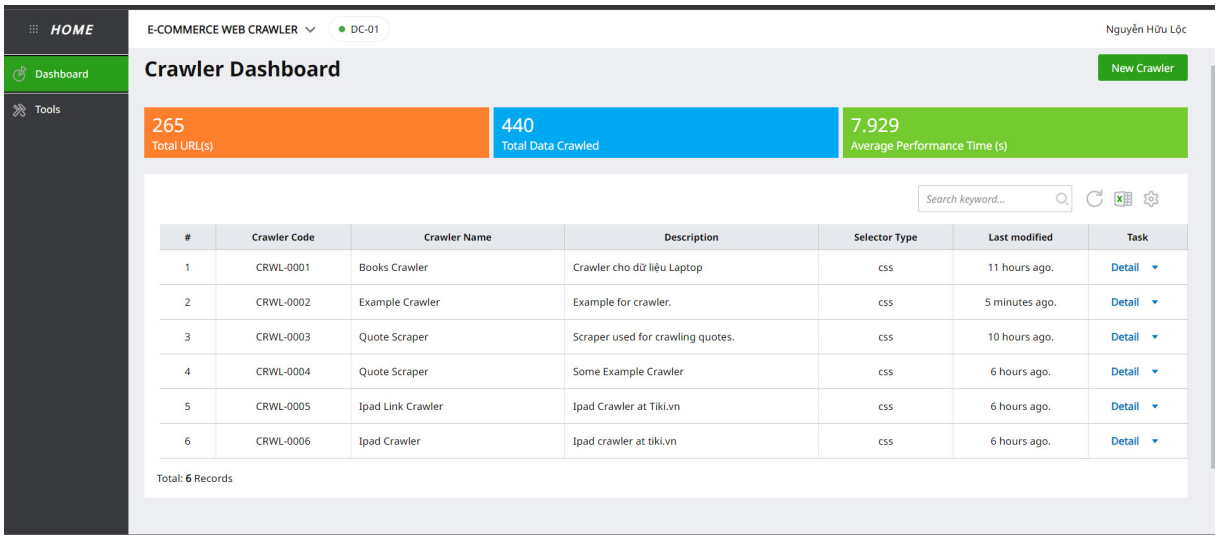

Figure 4.1: User Interface Crawler Dashboard

4.2.1.2 Crawler Form

This form contains the details information of a crawler. In this form, users can import the CSV files with URLs, extract selectors from the URL, crawl data from the imported URLs. Users can also manage the crawler information: creating a new crawler, updating the existing crawler, updating the extracted selectors. Users click "Save Crawler" to create or edit a crawler.

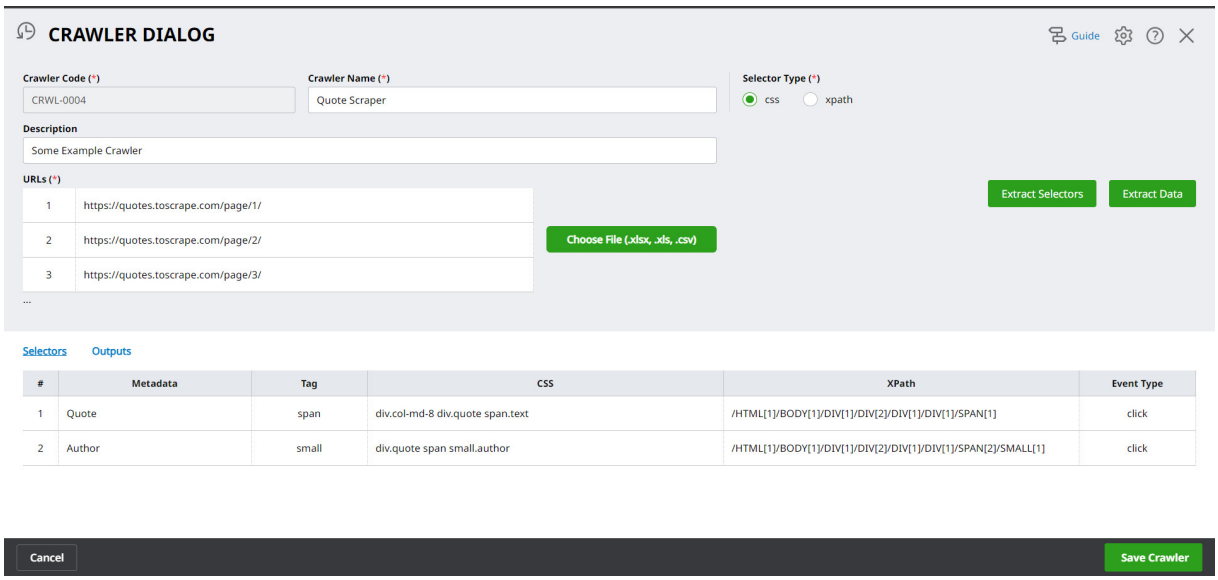

Figure 4.2: User Interface Crawler Form

Users can click "Extract selectors" to open a headless Chromium browser. In this browser, users can click on any data to extract its selector. When hovering on the data, users see the previewed tag of the data such as: a, p or h3. In order to finish choosing the selectors, click on the close (X) button of the browser.

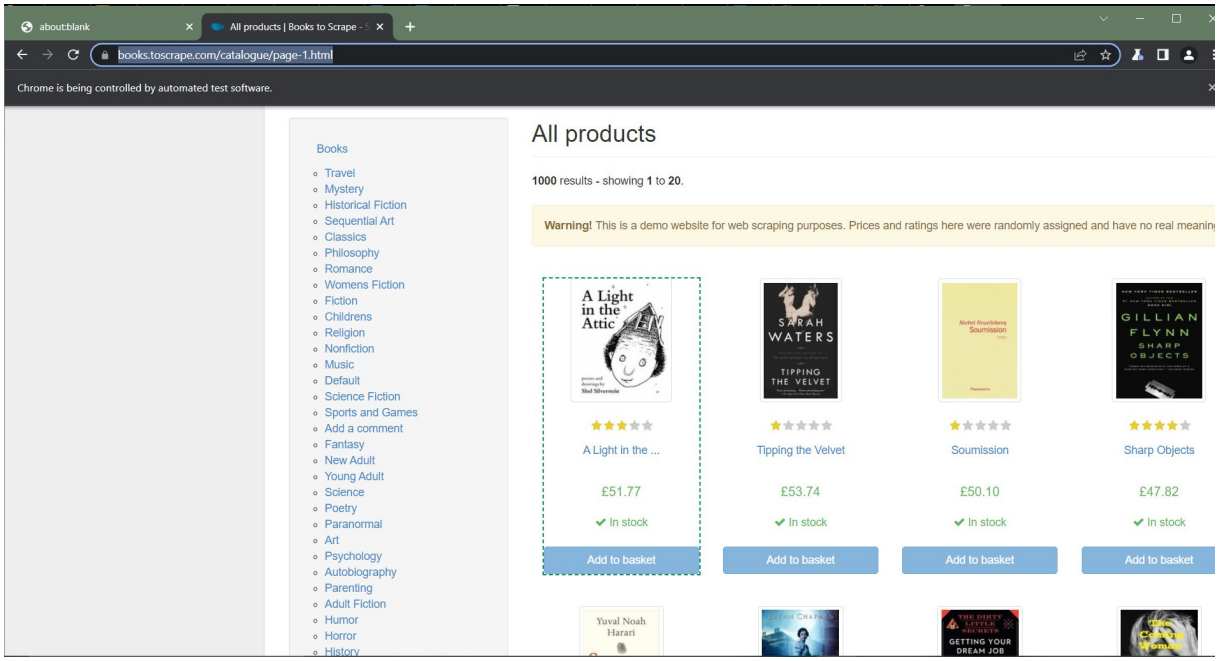

Figure 4.3: Headless Chromium Browser

4.2.1.3 Tool Page

This page is used for generating paginated URLs from a seed URL and exporting it into a CSV file which can be used again to import into a crawler.

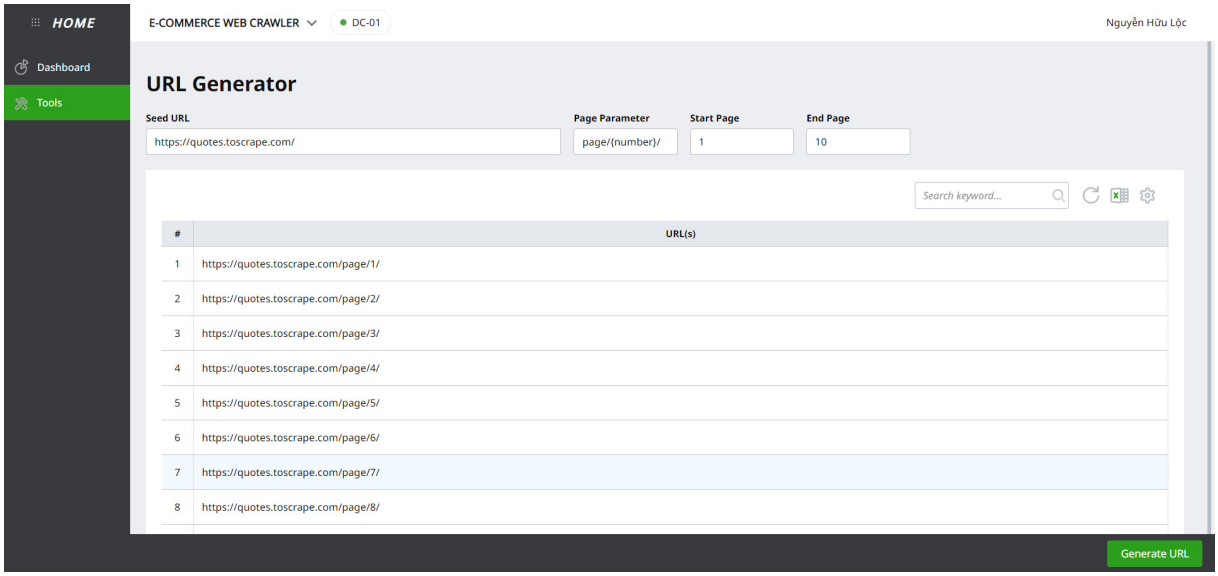

Figure 4.4: User Interface Tool Page

4.2.2 Application Programming Interface (API)

Our system use the RESTful API architecture style with CRUD operations. When the users query the existing crawler, server will also return its according selectors and crawler's output data. If the users want to update or delete a crawler, it also affects the selectors and crawler's output.

Method	API Route	Description
GET	/crawlers/	Get all the crawlers in the database.
	/crawlers/{code}	Get the existing crawler data.
	/crawlers/newCrawlerCode	Generate a new crawler code.
POST	/crawlers/	Add a new crawler to the system.
	/crawlers/record/{code}	Extract selectors with the existing crawler.
	/crawlers/scrape/{code}	Extract data with the existing crawler.
PUT	/crawlers/{code}	Update the crawler in the database.
DELETE	/crawlers/{code}	Delete the existing crawler.

Table 4.2: Crawler System API

4.3 Result

4.3.1 Crawling Performance

Website	Data Amount	Crawling time	Total crawling time	Accuracy
https://quotes.toscrape.com/	100	1.139 s	11.39 s	100%
https://books.toscrape.com/	1000	1.014 s	16m 54s	100%
https://tiki.vn/dien-thoai -may-tinh-bang/c1789	720	2.36 s	27m 56s	98.75%
https://laptop88.vn/may -tinh-xach-tay.html	400	1.76s	11m 46s	83%

Table 4.3: Crawling Performance Figures

Even though the figures at **Table 4.3** shows that the efficency of our crawler decrease with increase in URLs, it still shows great result. In this paper [2], R. Chaulagain et al. (2017) compare his scraping performance with other systems. Their system fetches an average of 100 URLs within 8 minutes, 200 URLs within 14 minutes. Compared to their cloud system, our method is much more efficient.

Retrieving data from the first two website *quotes.toscrape* and *books.toscrape* has great results due to the low complexity of the site. The results of data extraction in the two websites mentioned are 100% accurate with no missing data and average around 1.05 second per page.

While on the other hand, *tiki.vn* and *laptop88.vn* is a large e-commerce website with high complexity, which takes around 2.1 second per page to crawl. The crawler system face difficulties when scraping from tables containing tablets product details such as battery capacity, brand name, etc. Since each product's table detail can contain different information, it is difficult to extract the correct information from each product. It still can extract simple information such as the product's name and price.

4.3.2 Dealing with Website Changes

To show how our proposed system deal with the problem of website structure changes, we will demonstrate it in our web application.

Figure 4.5: Example Portfolio Website

During the time we crawl the data using the website in 4.1.2, there are no web structure changes. So in order to create an example of how we solve this problem with our crawler, we use a mock portfolio website which contain a list of project information. The information is presented in **figure 4.5**.

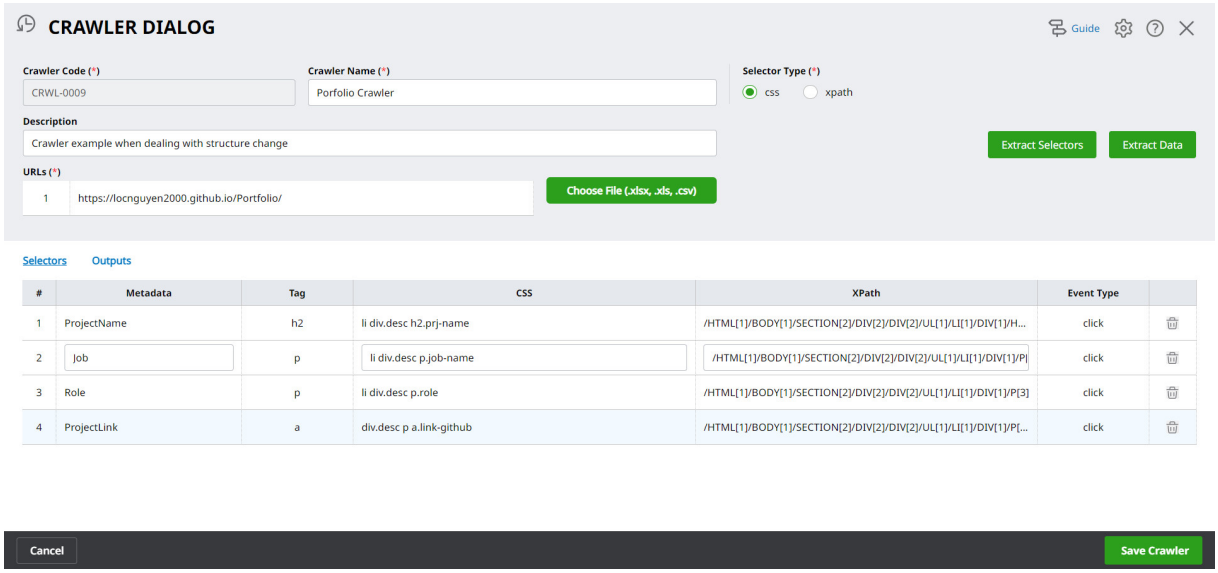

Figure 4.6: Portfolio selector before web structure change

Firstly, we create a new crawler and add the URL to its configuration. Afterwards, we can extract the web page's selectors matching with the information we need to crawl and then we will change the HTML class in the source code of the portfolio website. As a result, when we attempt to crawl the data again, it will return empty data since it cannot match the correct selector.

#	Metadata	Tag	CSS	XPath	Event Type	
1	Project	h2	ul#prj-list li div.desc h2.prj-name-updated	/HTML[1]/BODY[1]/SECTION[2]/DIV[2]/DIV[2]/UL[1]/LI[1]/DIV[1]/H...	click	
2	Job	p	ul#prj-list li div.desc p.job-name-updated	/HTML[1]/BODY[1]/SECTION[2]/DIV[2]/DIV[2]/UL[1]/LI[1]/DIV[1]/P[1]	click	
3	Description	p	ul#prj-list li div.desc p.job-desc-updated	/HTML[1]/BODY[1]/SECTION[2]/DIV[2]/DIV[2]/UL[1]/LI[1]/DIV[1]/P[2]	click	
4	Role	p	ul#prj-list li div.desc p.role-updated	/HTML[1]/BODY[1]/SECTION[2]/DIV[2]/DIV[2]/UL[1]/LI[1]/DIV[1]/P[3]	click	
5	Link	a	li div.desc p a.link-github-updated	/HTML[1]/BODY[1]/SECTION[2]/DIV[2]/DIV[2]/UL[1]/LI[1]/DIV[1]/P[	click	

Selectors Outputs

#	Project	Job	Description	Role	Link
1	PROJECT: WEB BÁN HÀNG QUẦN ÁO CỦA LỚP ...	Job: Front-end Developer	Description: Một thành viên trong nhóm làm ...	Role: Tạo front-end cho trang web bán hàng v...	https://github.com/LocNguyen2000/SwingsW...
2	PROJECT: WEB CHAT CỦA LỚP CÔNG NGHỆ PH...	Job: Web Developer	Description: Một trang web có đủ tính năng c...	Role: Trực tiếp làm một trang web chat cơ bản...	https://locnguyen2000.github.io/Chat-App/ind...

Figure 4.7: Portfolio selector after web structure change

To deal with this problem, our crawler system can extract the selector again and allow the users to click and choose which data they want to extract. New selectors will be updated in the table of the crawler form. Finally, we can crawl data again with the updated selectors.

Conclusions

This thesis has presented a web-based crawler system using the Puppeteer framework which can adapt to website changes by recording user's click-and-point event from a headless browser to extract data. While our system goal is present a new approach to deal with the website structure changes, this problem has been handled by many different state-of-the-art web crawling techniques [6] [14] [16] and each approach has their own advantages. Our system provide users with simple user interface to create crawlers with different purposes. When dealing with the website changes, users can make changes to the crawler quickly from a few simple click.

The limitations of the thesis is that when dealing with a large website with high complexity that contains many resources, it decrease the performance of the crawler significantly and causing the crawler to not extract the data accurately. Since our web application is running on a local computer, it will not achieve the efficiency compared to running on servers with the ability to run multiple threads to scrape data, thereby increasing the performance of the crawler.

Moreover, another limitation of this crawler system is the amount of manual works a user have to do. For example, we have a crawler that contains many selectors data. When the website changes, the user will have to locate all the selectors again.

Finally, this crawler approach still face difficulty dealing with tactics to prevent web crawlers such as authentication login, captcha, etc. Our crawler algorithm needs improvement in order to bypass these anti data extraction techniques.

In future works, we will work on building a crawler server with capability of running multiple threads to deal with performance issues. Additionally, to deal with crawler ethnicity, crawler should be able to recognize robots.txt to determine which pages are allowed to crawl from. It is imperative to create a crawler that can understand the robots policy of the site. Lastly, we can improve from our crawling algorithm to bypass some anti data extraction techniques.

With the growth of internet usage, websites are becoming more advance which always have the needs to be updated, causing many crawlers from extracting data. To efficiently extract meaningful data from the vast ocean of web contents, many techniques and approaches have been used to deal with this problem, such as machine learning algorithm and other custom advance algorithm. Web crawling/ scraping is a powerful and widely used technology which allow user to extract data from many website. In order to create a powerful generic web crawler, it will take a lot of works and lots of challenges to deal with.

References

[1] V. Carle, "Web Scraping using Machine Learning," *TRITA -EECS-EX-2020:515*, 2020.

[2] R. Chaulagain, S. Pandey, S. Basnet, and S. Shakya, "Cloud based web scraping for big data applications," 11 2017.

[3] ChecklyHQ, "Checkly - Headless Recorder," 08 2021, Accessed June 27, 2022.

[4] Google, "Puppeteer," https://devdocs.io/puppeteer/. Accessed June 26, 2022.

[5] Z. Guojun, J. Wenchao, S. Jihui, S. Fan, Z. Hao, and L. Jiang, "Design and application of intelligent dynamic crawler for web data mining," pp. 1098–1105, 05 2017.

[6] F. Khan, G. Tsaramirsis, N. Ullah, M. Nazmudeen, S. Jan, and A. Ahmad, "Smart algorithmic based web crawling and scraping with template autoupdate capabilities," *Concurrency and Computation Practice and Experience*, vol. 33, 10 2020.

[7] V. Krotov, L. Johnson, and L. Silva, "Legality and Ethics of Web Scraping," *Communications of the Association for Information Systems*, vol. 47, pp. 539–563, 01 2020.

[8] A. Lawal, Y. Ali, B. Ahmad, and M. Abdullahi, "An enhanced markup randomization to prevent xpath and css based web scrapers," vol. 11, pp. 104–117, 12 2021.

[9] P. Meschenmoser, N. Meuschke, M. Hotz, and B. Gipp, "Scraping Scientific Web Repositories: Challenges and Solutions for Automated content extraction," *D-Lib Magazine*, vol. 22, 09 2016.

[10] R. Mitchell, "Web Scraping with Python: Collecting more data from the modern web," 2018, accessed July 9, 2022.

[11] M. Mozilla.org., “CSS selectors - CSS: Cascading Style Sheets,” 05 2022, https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors. Accessed July 16, 2022.

[12] ——, “XPath,” 05 2022, https://developer.mozilla.org/en-US/docs/Web/XPath. Accessed July 16, 2022.

[13] C. N. Oskar Lloyd, “How to Build a Web Scraper for social media,” 06 2019, https://www.diva-portal.org/smash/get/diva2:1480471/FULLTEXT01.pdf. Accessed July 18, 2022.

[14] R. Penman, “Web Scraping Made Simple with SiteScraper,” 01 2009.

[15] B. Roberts, “Rethinking Web Scraping with Autoscrape - artificial informer,” 2019, https://artificialinformer.com/issue-one/rethinking-web-scraping-with-autoscrape.html. Accessed July 18, 2022.

[16] U. S, B. Gaind, A. Kundu, A. Holla, and M. Rungta, “Classification-based adaptive web scraper,” 12 2017.

[17] Scrapy.org, “Scrapy,” 05 2022, https://docs.scrapy.org/en/latest/. Accessed June 29, 2022.

[18] S. Stewart, “Selenium,” 03 2022, https://www.selenium.dev/documentation/. Accessed June 28, 2022.

[19] W3.org., “What is the Document Object Model?” 2022, https://www.w3.org/TR/WD-DOM/introduction.html. Accessed July 16, 2022.