# 4. Implementation

## 4.1. Website Gathering

The Website Gathering tool is used to collect multiple websites’ URLs and create a list with them, to be used for other purposes later. The core part of this tool can be found in the get\_site.py file (Appendix ..). The list returned by get\_url\_list function in the programme is used in the Policy Extraction component of the project, but the programme has been specifically designed so that the output list can be utilised for any other purposes in future work.

One of the original objectives of this project stated that privacy policies of UK-based retailers would be examined. However, it was difficult to find a list of UK retailers from a trusted and legitimate source that could be used for academic purposes. A well-known list of websites, which is commonly used for research purposes, is Alexa Top Sites, an Amazon Web Services (AWS) service which provides access to lists of websites ordered by Alexa Traffic Rank [44]. The Traffic Rank is based on the traffic data provided by Alexa’s global sample over a rolling 3 month period, and is updated daily [45].

In addition to being recognised by the academic community as a legitimate dataset for website ranking, Alexa Top Sites also presents the advantage of being a global resource, providing users with worldwide website rankings, and rankings for most countries in the world. This means that while this project focuses on the top sites in the United Kingdom, the Website Gathering tool is not limited to a particular country and can be used to collect lists of websites from any country. This feature is particularly interesting to conduct comparisons between sites from different countries, for example between European Union member states, as using the same source to collect data makes the study more reliable.

The first prototype of the policy gathering tool collected a list of urls with the web-scraping tool Selenium. The starting point was Alexa’s top 50 UK sites list [39]. The programme would search for the xpath element “//p/a”, which is where the 50 URLs were to be found based on previous manual inspection of the webpage. The list of URLs would then be stored in a csv file for future use.

While the web scraping method to get the list of sites’ URLs is convenient and straightforward, Alexa only allows access to the first 50 sites of its rankings in this way. To be able to access information on more sites, one must access Alexa’s API and make requests. The service enables users to page through the list of top sites 100 websites at a time, meaning that it is necessary to make multiple API requests to retrieve lists of a bigger size, for example ten requests for a list of 1000 websites [44]. By making these requests, lists of any size can be retrieved.

The main prototype of the Website Gathering tool was built by focusing on getting a list of 500 websites for Alexa’s API, which meant doing five separate API requests. This number was later increased to obtain more data. AWS provides users with a set of instructions to be followed to get code samples from GitHub and an API key to complete the request [46], as well as a readme file for each programming language that the API supports, in this case Python [47]. The function that is used to make the API requests, get\_api\_data, is stored in get\_sites.py (Appendix …), and calls a programme provided by AWS in their code samples, topsites.py (Appendix …). To avoid making multiple requests separately, and having to repeat the code, a for-loop was used to execute the function as many times as defined by the range.

The instructions provided by AWS indicate how to run the sample code and get an output in the command line, but they do not provide guidance on how to incorporate the API request in another programme. The Python subprocess module [48] was used with the function Popen to achieve this aim. This module enables the execution of external programs and the reading of their output directly in another Python code, which is exactly what was wanted here. The command for the API request, which is passed into Popen is:

"python3 websites\_data/topsites.py --key=KM19IsQpxiaSfQcBBcW431zop23rUEHq3hCbLpiA --action=TopSites --options {options} --country={country\_list}"

Variables have been created for the values allocated to “--options” and “--country” to avoid hard-coding and make the programme more robust.

* For the Service Options, which determine the number of results per output requests (Count=100), the output type (Output=json), the starting point and the response group:

options = f"\"&Count=100&Output=json&Start={start\_point} &ResponseGroup=Country\""

* To determine which country to get the topsites from, the variable country\_list has been created. The country should be given by a 2-letter country code (i.e. US, GB, RU, FR). The variable is currently set to country\_list = 'GB' but the programme was tested with ‘US’ and ‘FR’ and they both work and return the desired output.

Each iteration creates a file in JSON format, which contains information about 100 websites. There are as many JSON files stored in the websites\_data directory as there are API requests. These files contain a lot of information regarding Alexa’ selected top sites, while only the websites’ URLs are relevant for this project. However, the structure of the JSON makes the extraction of the URLs slightly complex. For example, here is the path to access information from google.com:

"Ats": {

"Results": {

"Result": {

"Alexa": {

"TopSites": {

"Country": {

"CountryName": "United Kingdom",

"CountryCode": "GB",

"TotalSites": "1603",

"Sites": {

"Site": [

{

"DataUrl": "google.com",

"Country": {

"Rank": "1",

"Reach": {

"PerMillion": "646000"

},

"PageViews": {

"PerMillion": "212000",

"PerUser": "4.74"

}

},

"Global": {

"Rank": "1"

}

},

The function get\_url\_list in get\_sites.py is used to extract all the ‘DataURL’ elements from each JSON file and store them in one single CSV file (url\_list.csv).

The function get\_url\_list also adds the prefix “https://www.” to each elements of the list of URLs before adding them to the CSV file so that the websites URLs are all in the correct format to communicate between the browser and server when they will be accessed.

Once the implementation was over and the programme was functioning correctly to collect 500 websites, the number of sites requested was extended to the total number of sites available on Alexa Top Sites in the UK’s API which is, as of August 2021, 1603.

The final output of the Website Gathering component is a list of 1603 websites’ URLs stored in a CSV file, ready to be used for any sort of analysis.

## 4.2. Web Scraper

*ADD TITLE---- Overview and general implementation choices*

The Policy Extraction component of the project, which aims at extracting the content privacy policies from multiple given websites, was built using a Web Scraper. Python was chosen to build the scraper, as it appeared to be the most appropriate language for this task due to the vast community and library resources available [37][38]. Because of that, the rest of the project was also written in Python to ensure consistency.

Two open-source frameworks were considered when deciding how to build the Web Scraper, Scrapy and Selenium. Scrapy is a free and open-source web crawling framework written in Python [27]. It is an integrated system which includes an engine for controlling data flow between all components, a scheduler for receiving requests, a downloader for fetching web pages, and custom classes called “spiders” to parse response and extract items [50]. Scrapy is a popular choice for web scraping because of its easy of use, and the large community available online. However, it does not allow for a lot of freedom to programme the scraper and get it to behave in a very customised way. If the aim is to mimic a human browsing online, Scrapy is not necessarily the best option available, and Selenium appeared as a more appropriate choice to conduct this project.

Selenium is a powerful web scraping tool, which was originally created for developers to test their own website, but is now widely used in projects where it is useful or required to see websites as they actually appear in a browser. Selenium Python, which is used in this project, automates browsers to load the website and retrieve data by providing a simple API to write functional tests using Selenium Webdriver [43].

Selenium does not contain a web browser, and requires integration with third-party browsers such as Google Chrome or Firefox in order to run. Another option is to use a “headless” browser, like PhantomJS [52], which will run quietly in the background without showing the user any graphics of the website [51]. Headless browsers can be very powerful, but for this project there is a clear benefit in having visibility on the browser’s graphics, to gain a better understanding of how a human would be interacting with the websites. Based on Selenium’s documentation [43], the most popular supported drivers include Chrome [53] and Firefox [54]. No research or information found online identified drivers as better than the other for Selenium, but Firefox Driver has bundled with the Selenium standard library. This means that there is no need to install any external file, which makes it easy to initialise and run, and therefore Firefox is the browser used for the web scraper.

*ADD TITLE*---- *Manual review of websites*

The aim of the implementation of the Policy Extraction component is to build a web scraper that, given a URL, would:

* Autonomously access load a website,
* Find a link to the privacy policy on the website’s homepage,
* Load that the privacy policy page,
* Retrieve the data from that page, and
* Store the data locally in a file.

The starting point of the scraper’s creation was to manually inspect the top ten URLs from Alexa Top Sites in the UK [39] to identify patterns and similarities when looking for a link to the privacy policy on each website’s homepage. A table detailing these findings is available in Appendix ... . This step was useful to understand a couple of key elements to set up the scraper (which apply at least to these websites):

* The keyword “privacy” can be found in an anchor tag <a>. An anchor tag is used to identify sections within a document.
* The anchor tag contains an HTML href attribute which itself contains the term “privacy” in most cases. An href attribute (Hypertext REFerence) is the HTML code used to create a link to another page.
* Capitalisation needs to be taken into account, because “privacy” may be written with a lower-case or upper-case “P” in the anchor tag or href.
* The term “privacy” may appear several times on the page, either in another anchor tag, or in another tag such as a paragraph of a text (<p> tag). This needs to be considered when designing the web scraper as it might cause issues when searching for the privacy link.

*ADD TITLE*---- *Finding elements with an XPath Selector*

Web scrapers use selectors to find HTML elements in web pages. Two popular types of selectors supported by Selenium are XPath, which uses path expression to select nodes in an XML document, and CSS Selector, which tells the browser which HTML elements should be selected to have CSS property values inside the rule applied to them.

A Web Scraper online course on Udemy [56] recommended using XPath selectors rather than CSS selectors. The main advantage they cited was that XPath gives the ability to look for elements forwards and backwards in the DOM hierarchy, while CSS only supports forward search. Even though neither this feature or Scrapy ended up being used in this project, no resources were found stating the existence of significant differences in performance between XPath and CSS. Therefore, XPath Selectors were used to look for elements in the web pages.

One important drawback with using the XPath Selector in this case is that the selector is case-sensitive, meaning that it would not find “Privacy” with an upper-case “P” unless expressly specified in the search. Different options were considered to deal with this issue, including switching back to using a CSS Selector, which seems to deal better with capitalisation. However, there are only two options to consider, which are “privacy” and “Privacy”, therefore searching for both terms separately within the find\_elements\_by\_xpath function was more appropriate than learning a whole new technique.

To avoid hard-coding and to make the scraper more robust and extendable to other languages, two variables (privacy\_lower\_case and privacy\_upper\_case) were created to support lower-case and upper-case versions of “privacy”. They are assigned to “privacy” and “Privacy” if the language is set to ‘English’ A potential improvement to consider here would be to use regular expressions to deal with the capitalisation issue, and have only one variable for “privacy” in English.

While most websites that were manually had the term “privacy” contained in the href link to the privacy page, Live.com (<https://www.live.com>) did not, which shows that setting a XPath rule which only looks for “[P-p]rivacy” in the href would not be enough to get the privacy policy links from all website, and errors would occur. This potential issue was fixed by adding another condition in the XPath Selector, to look for “[P-p]rivacy” directly in the anchor tag as well as in the href.

*ADD TITLE*---- *Dealing with several privacy tags*

Some web pages visited by the scraper may have several occurrences of the term “[P-p]rivacy”. The XPath selector is only searching for the term in anchor tags and hrefs, which means that it will not be a problem if the term is used in another type of tag, such as a paragraph of text. If there are several href attributes linking to the privacy policy page, the scraper will automatically pick one without any issue. However, a problem may arise if multiple anchor tags or href attributes meet the rules of the XPath selector, but that some of them do not actually link to the privacy policy page of the website.

An example of this problem is observed for [www.theguardian.com](http://www.theguardian.com). The HTML has two anchor attributes which contain the keyword: “Privacy settings” and “Privacy policy”. Here’s what the HTML looks like for each element:

* “Privacy settings”:   
  <a data-link-name="privacy-settings" class="">Privacy settings</a>
* “Privacy policy”:   
  <a data-link-name="privacy" href="<https://www.theguardian.com/info/privacy>" class=""> Privacy policy </a>

The “Privacy settings” tag does not actually contain an href. It’s an IFrame (Inline Frame), an HTML document embedded inside the HTML document of the homepage, which offers the user options to change their privacy settings, without moving to another link (Figure 2). However, the web scraper chooses that tag by default because it meets the XPath Selector rules and it comes up first in the HTML.

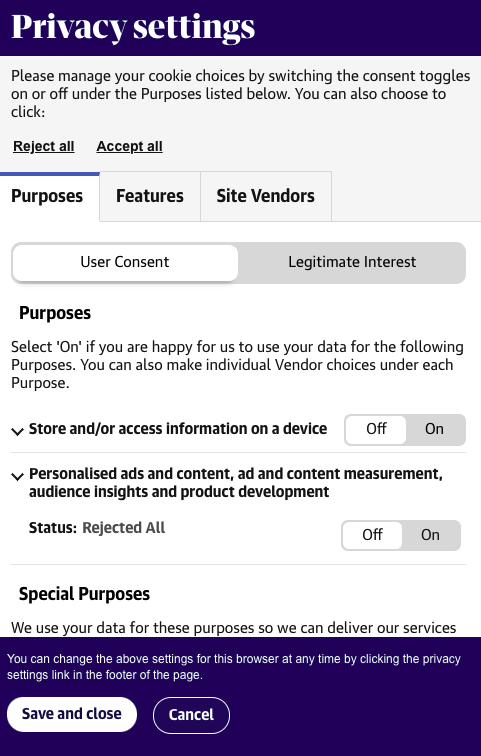


Figure 2. theguardian.com Privacy Setting pop-up

The score\_privacy\_tag() function deals with this problem by assigning a score to each tag found by the scraper using the XPath Selector. Each tag’ score will lose or gain points depending on whether certain conditions are met. The tag with the highest score will be selected as most likely to link to the privacy policy page. Currently, the function supports two conditions, and could be improved in the future by adding more conditions.

Current conditions in the tags’ scoring function:

* If a tag does not contain an href attribute, then the tag’s score loses 100 points.

if button.get\_attribute("href") == None:

score[i] = -100

* If the tag’s href attribute contains the term “privacy” or “Privacy” is in the href attribute, then the tag’s score goes up by 50 points.

for st in [english\_lower\_case, english\_upper\_case]:

if st in button.get\_attribute("href"):

score[i] += 50

*ADD TITLE*--- *Dealing with zero privacy tags*

Some websites do not have a link to a privacy policy on their homepage, and therefore the web scraper will not find an HTML element which matches the rules of the XPath Selector. The reasons for this absence of link vary, and will be discussed in the Evaluation chapter of this section. However, it was necessary to add some code to handle these situations to avoid the interruption of the scraper. To deal with this exception, the programme counts the number of occurrences where the XPath Selector rules are met. If this number is equal to zero, then the function will print an error message and the scraper will move on to the next website of the input list, or will finish execution if there are no more websites left to visit.

elif len(privacy\_tags) == 0:

print(f"Error: No privacy tag found for {url}")

return ""

*ADD TITLE---- Dealing with URLs from input that cannot be found*

It is possible that the web scraper will not be able to find or to access some websites from the input list, either because the URL not longer exists or has been moved (client-side incident), or it might have been misspelled in the original source where the input list of URLs has been found. It is also possible that the website cannot be accessed for security reasons, for example if the site does not support the minimum security protocol TLS 1.1. To make sure that the scraper keeps running if it encounters one of these cases, *try* and *except* clauses were added around the executing statement. This way, if the URL cannot be found or accessed, the programme will print an error statement but will keep running, moving on to the next website.

try:

browser.get(url)

except:

print(f"Error: Something went wrong loading website {url}")

return ""

*ADD TITLE---- Accessing the privacy link and storing its the page source in a file*

Once the website’ homepage and the tag containing the privacy link has been found, the web scraper accesses the privacy page using the selenium function get. The page source - the HTML code of a web page - is then retrieved using the page\_source function.

The final task of the web scraper is to store the page source in a file, in order to be used later for other purposes.

The output of the Policy Extraction component is a corpus of privacy policies in html format, saved in files locally.

## 4.3. Sanitise HTML with Beautiful Soup

The aim of the Policy Sanitisation tool is to clean a given privacy policy page which is in HTML format, and return it as a more readable and manageable text file. Beautiful Soup is used to achieve that goal.

For the purpose of this project, the two architecture components - Policy Extraction and Policy Sanitisation - are closely intertwined. The function clean\_policy(), where most of the steps for the Policy Sanitisation part are completed, is directly called in get\_policy() which is the core function of Policy Extraction. However, this could easily be separated to use Policy Sanitisation independently on any given HTML file.

Beautiful Soup is a Python library that is commonly used to pull out data out of HTML and XML files. It creates a parse tree from the input, a page source code, which can then be used to extract the data in a more human-readable manner [57]. In this project, the focus is on HTML documents.

The HTML sanitisation starts with getting the content from a given URL or HTML file. At this stage of the project, the page URL has already been retrieved using the Policy Extraction tool, and the HTML stored in a variable of type string called page\_source. This is passed as an argument in the clean\_policy() function, and no further step is necessary.

The following stage involves parsing the content of the page source, currently in HTML format, into a Beautiful Soup object. The Beautiful Soup object represents the parsed document as a whole, and has the following syntax: *BeautifulSoup(document, parser)*. In this project’s implementation, the document is the page source previously retrieved by the Policy Extraction tool. The parser is a structured markup processing tool, here an HTML parser because of the format of the input document.

soup = BeautifulSoup(policy\_text, features="html.parser")

The next step of the HTML sanitisation is to iterate over the data to remove tags and other unwanted elements from the document. Beautiful Soup has a function called decompose() which removes a tag from the parse tree, and completely destroys it and its content [57]. To remove the structure of the HTML and only keep the text content of the page, all the JavaScript (“script”) and CSS StyleSheet code (“style”) is removed.

for tag in soup(["script", "style"]):

tag.decompose()

Once the unwanted tags have been removed, policy\_text can be reconverted from a Beautiful Soup object back to text. This is completed using the function get\_text(), which returns all the text in a document as a single Unicode string [57]. In other words, it returns the human-readable text inside the document.

policy\_text = soup.get\_text()

To produce a cleaner and more readable result, the text can be further modified before being outputted or stored in a file. The following text manipulation steps are implemented:

* Break policy\_text into line and remove the leading and trailing spaces on each of them.

lines = policy\_text.splitlines()

for line in lines:

line.strip()

* Remove all the blank lines from policy\_text.

merged\_text = '\n'.join(lines)

The final output is ready to be stored locally in a text file, for future use. For this project, the clean\_policy() fonction, which constitutes the whole Policy Sanitisation component, is used to clean the output from get\_policy(), the main function of the Policy Extraction tool.

The privacy policies are then clean, readable, and usable for any future analysis. They are all stored in text files in one directory, ready to be used either as a corpus of documents, or examined individually.

## 4.4. Analysis