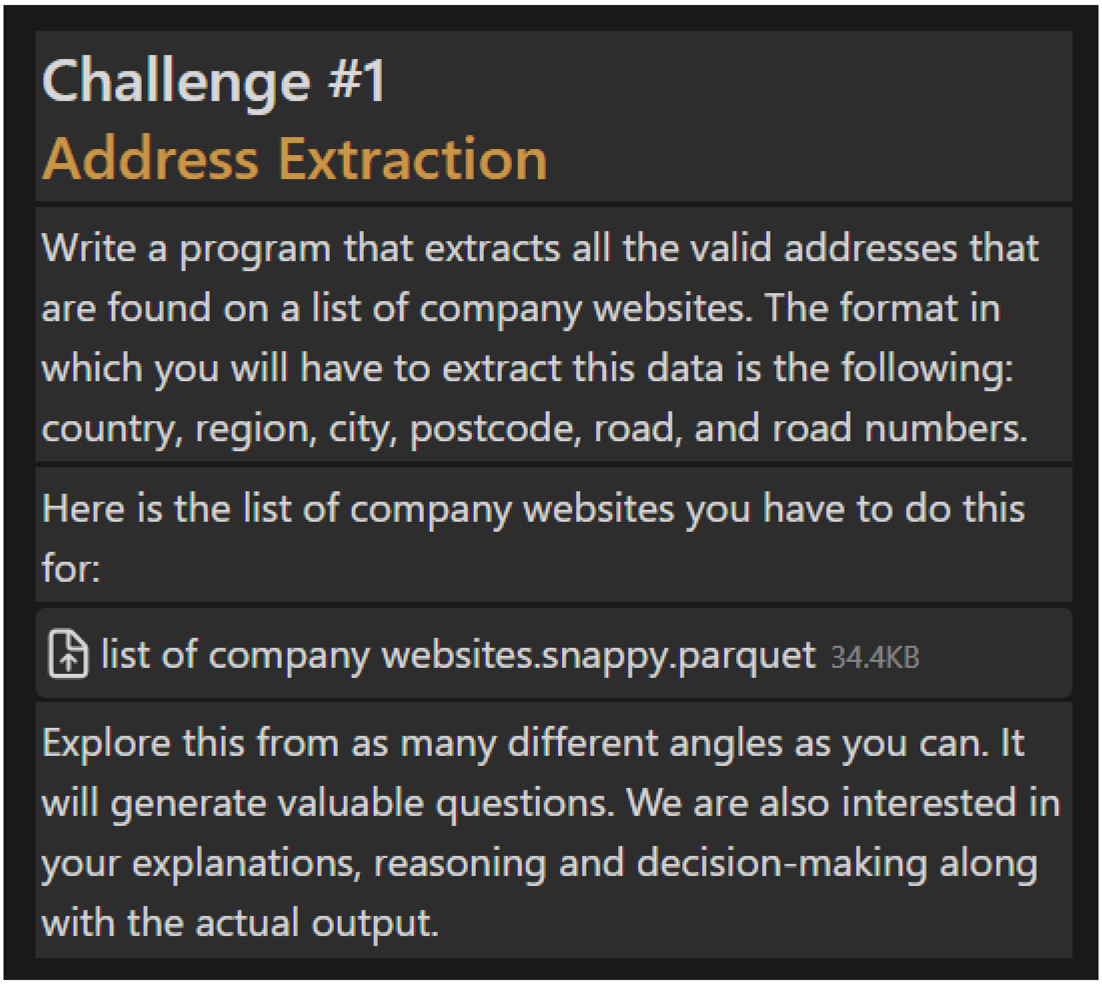
VERIDION PROJECT
CHALLENGE #1

- ***Why?***

This project was created for the internship position of **"Deeptech Engineer."** I've selected the first task from the three that were offered since I thought it was the most intriguing and challenging.



Challenge #1
Address Extraction

Write a program that extracts all the valid addresses that are found on a list of company websites. The format in which you will have to extract this data is the following: country, region, city, postcode, road, and road numbers.

Here is the list of company websites you have to do this for:

list of company websites.snappy.parquet 34.4KB

Explore this from as many different angles as you can. It will generate valuable questions. We are also interested in your explanations, reasoning and decision-making along with the actual output.

This is my first time attempting web scraping, and I find it fascinating. What I found most intriguing wasn't just the act of web scraping itself, but rather the way URLs function and the diversity of each web page, which made identifying a pattern for addresses challenging.

- ***First step - Analyzing the task***

I posed my questions in the following order.

How do I extract text from a webpage?

Then I realized that in the .parquet file, there are only domains and not URLs.
Then came the question:

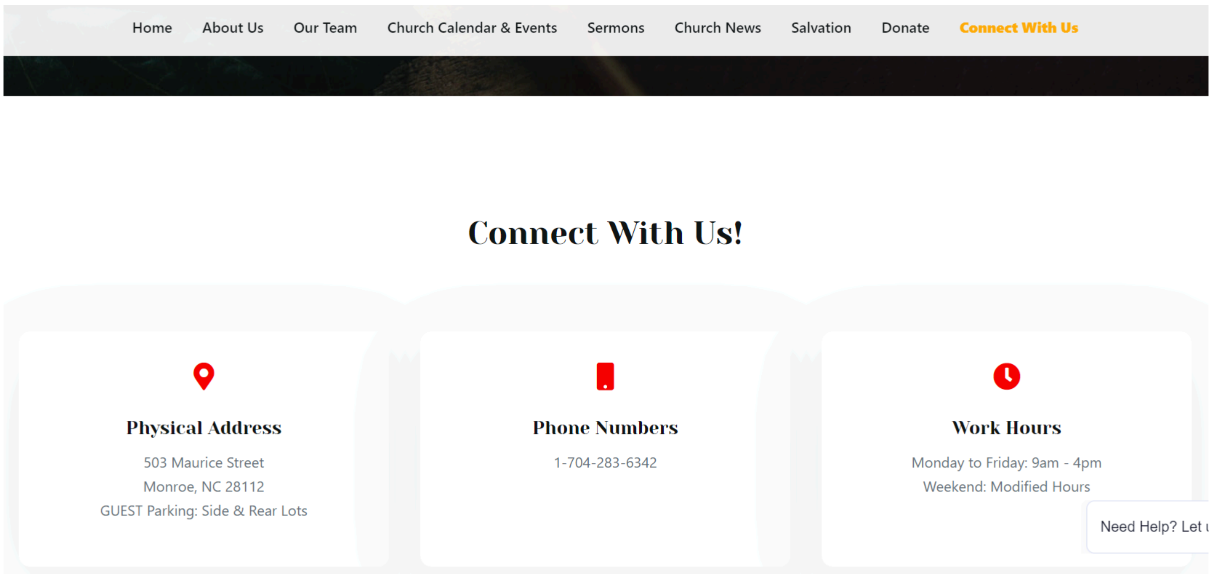
How do I construct a URL, how do I find out if it's HTTP or HTTPS?

Then, while searching online, I realized that this isn't as difficult to accomplish. With BeautifulSoup and Requests, these issues become resolved.
However, another very important question arises.
How do I generalize web scraping across all URLs to identify addresses?

- ***Step 2: Analyzing the domains***

I thought the best way to figure this out is to actually access some websites to see the formatting of addresses, what countries are involved, in search of a pattern.
The first time I accessed the first 20 websites, I noticed that they were all from the US. By doing this, I already observed quite a few things:

1) Addresses in the US have a formatting like this:
road number road name
city, state abbreviation postal code



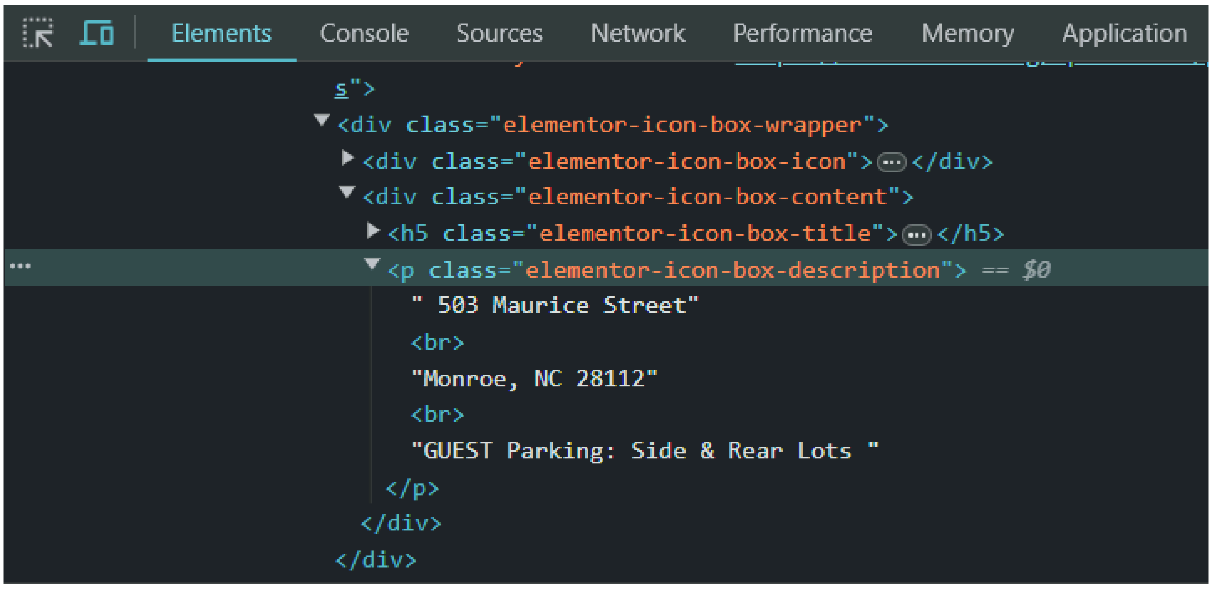
Connect With Us!

Physical Address
503 Maurice Street
Monroe, NC 28112
GUEST Parking: Side & Rear Lots

Phone Numbers
1-704-283-6342

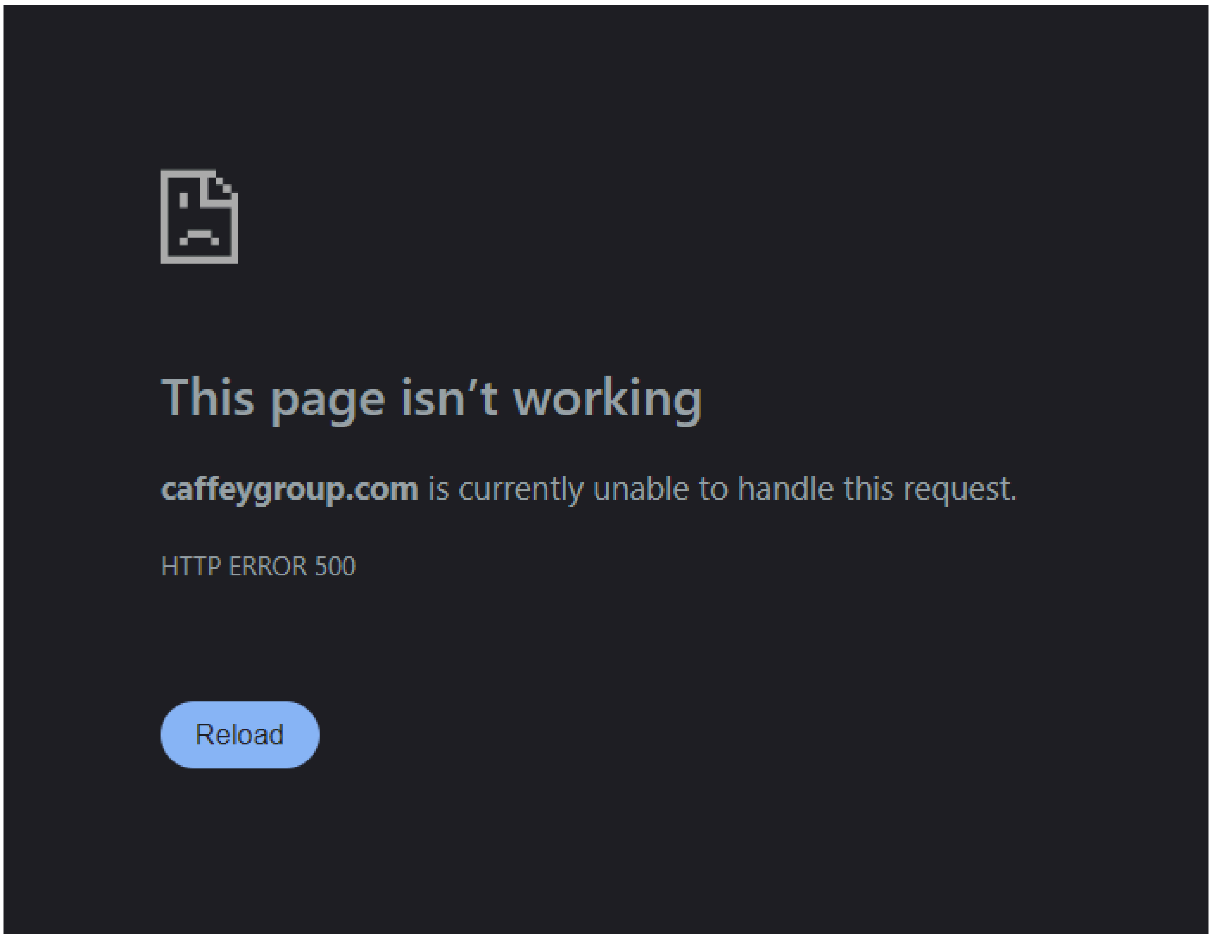
Work Hours
Monday to Friday: 9am - 4pm
Weekend: Modified Hours

2) Addresses are not necessarily found on the main page. Many of them are on pages like Contact, Contact Us, Connect With Us, etc.
3) By inspecting Addresses in the HTML code, I noticed that most of them are found within paragraphs (<p>).



```
s">
<div class="elementor-icon-box-wrapper">
  <div class="elementor-icon-box-icon">…</div>
  <div class="elementor-icon-box-content">
    <h5 class="elementor-icon-box-title">…</h5>
    <p class="elementor-icon-box-description"> == $0
      " 503 Maurice Street"
      <br>
      "Monroe, NC 28112"
      <br>
      "GUEST Parking: Side & Rear Lots "
    </p>
  </div>
</div>
```

4) Some of the websites are not functional



This page isn’t working

caffeygroup.com is currently unable to handle this request.

HTTP ERROR 500

I said in my mind that I will think how to solve the 4) observation after I solve the pattern for the address.
Then I realized that I would probably need regex to identify addresses, now that I've found a pattern. **But can't I find a shortcut without using regex?**

- ***Step 3: Doing research***

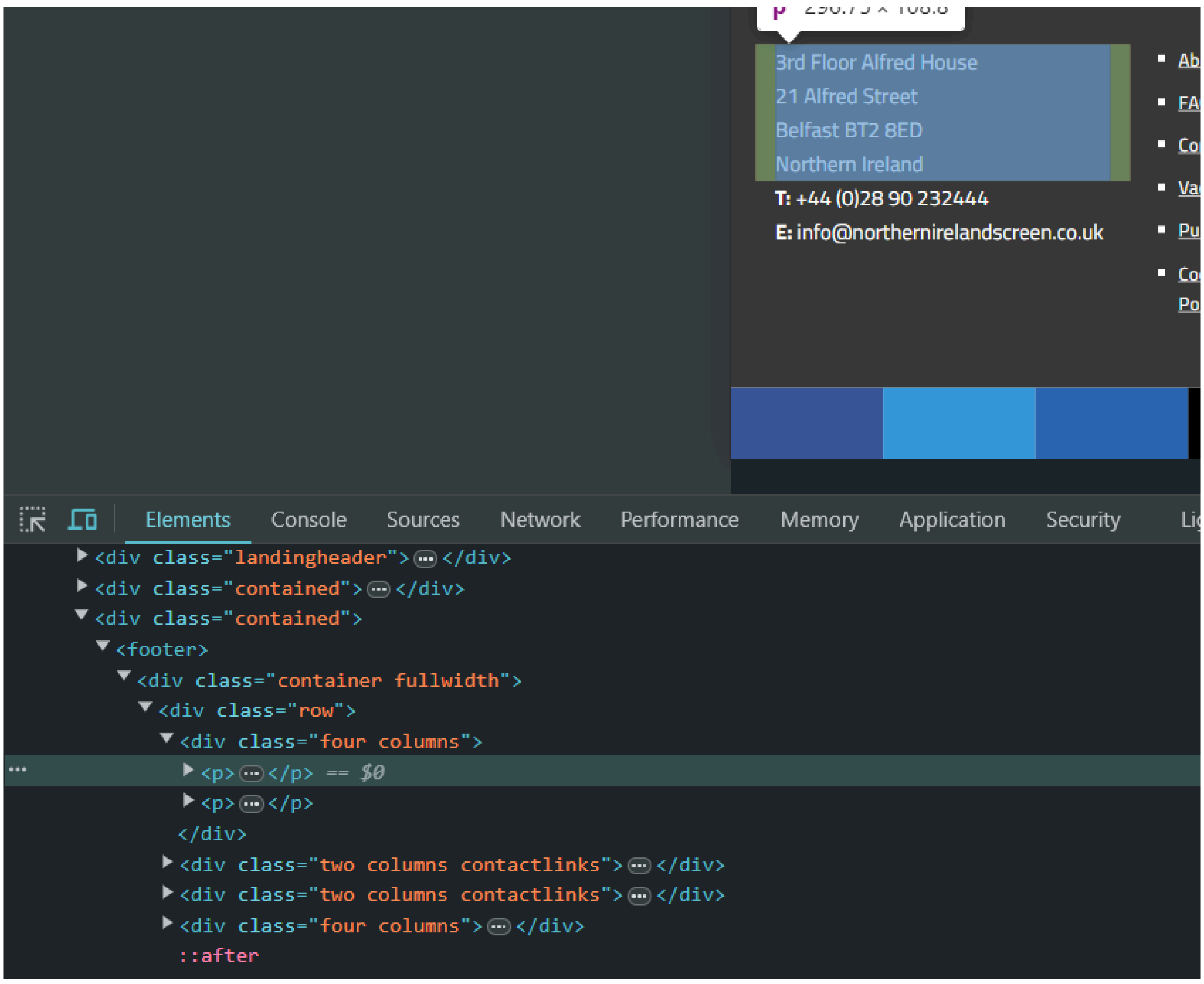
I searched online and found that someone has created a library that automatically identifies addresses from the content of a page if they are from the US or the UK (you have to specify what you're looking for). It's called pyap. So, I've solved this problem as well.

Now comes the question: **Besides the US, are there any other countries among the domains?**

- ***Step 4: Discovering the UK websites.***

Therefore, I took the last 10 websites. And I found that these were from the UK. I analyzed them to draw some conclusions (similarities and differences with those from the US).

1) The addresses are formatted quite similarly; the only thing missing for the UK is the abbreviation of a state, but what's more distinctive is the postal code, which contains white spaces and includes letters, unlike in the US where there are only digits.
2) Addresses here are also not necessarily found on the main page; the same words: Contact, Connect... lead to finding addresses. So, I thought that these would be good keywords for identifying pages with addresses in the future.
3) Addresses here are also identified within paragraphs.



3rd Floor Alfred House
21 Alfred Street
Belfast BT2 8ED
Northern Ireland
T: +44 (0)28 90 232444
E: info@northernirelandscreen.co.uk

```
▶<div class="landingheader">⋯</div>
▶<div class="contained">⋯</div>
▼<div class="contained">
  ▼<footer>
    ▼<div class="container fullwidth">
      ▼<div class="row">
        ▼<div class="four columns">
          ▶<p>⋯</p> == $0
          ▶<p>⋯</p>
         </div>
        ▶<div class="two columns contactlinks">⋯</div>
        ▶<div class="two columns contactlinks">⋯</div>
        ▶<div class="four columns">⋯</div>
         ::after
```

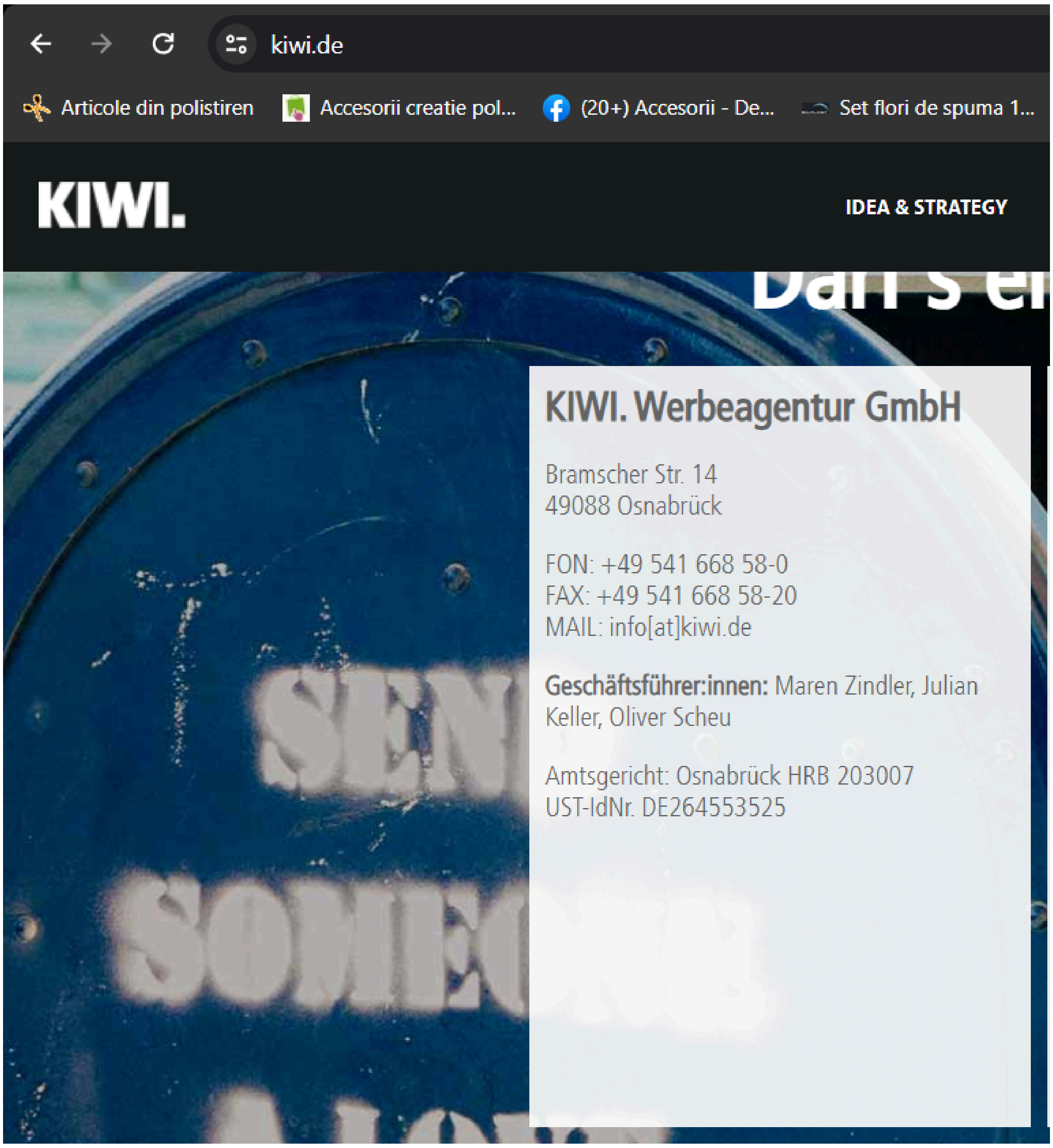
So, I realized that the same regex couldn't be used for them, but they still looked quite similar. Anyway, the extraction of UK addresses is also resolved by the pyap library, so I easily got rid of this problem.

The question becomes: **Are the US and UK the only countries?**

- ***Step 5: Discovering a German website***

I started selecting random sites from the list until I identified a German site. I analyzed its address, and here came the most intriguing part:

1) it was formatted differently:
road name road number
postcode city



KIWI. Werbeagentur GmbH

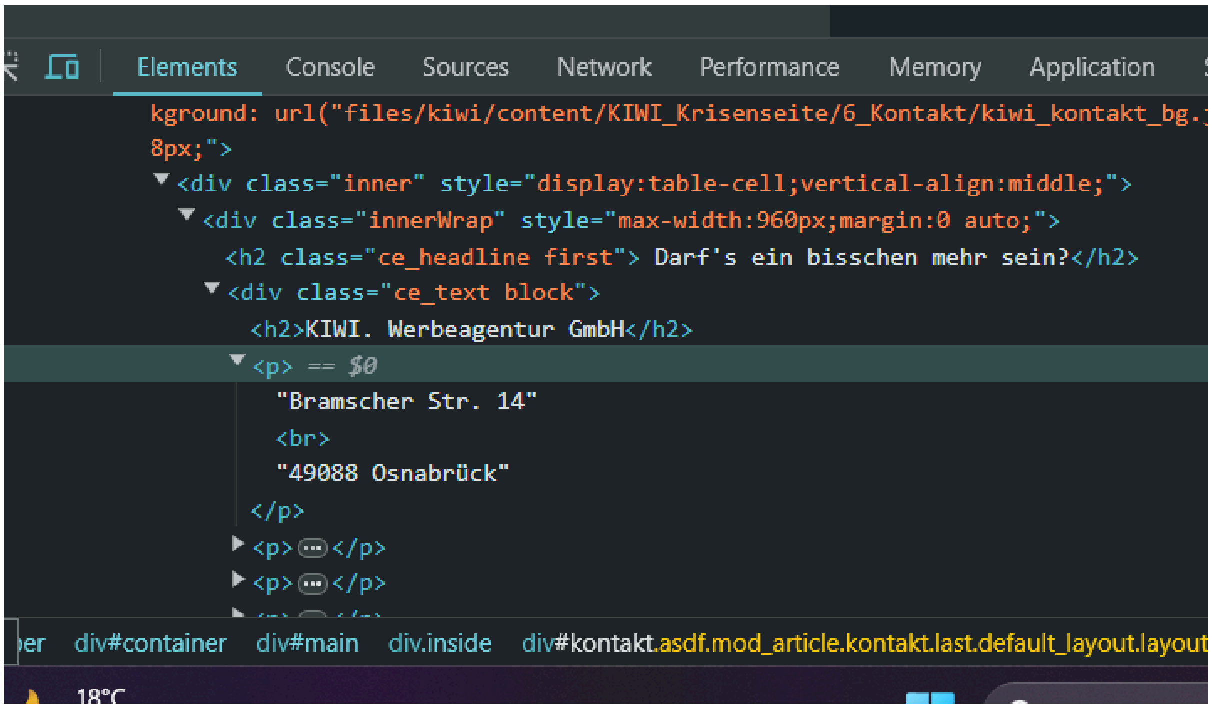
Bramscher Str. 14
49088 Osnabrück

FON: +49 541 668 58-0
FAX: +49 541 668 58-20
MAIL: info[at]kiwi.de

Geschäftsführer:innen: Maren Zindler, Julian Keller, Oliver Scheu

Amtsgericht: Osnabrück HRB 203007
UST-IdNr. DE264553525

2) the address had letters not found in the English alphabet
3) One good thing is that addresses are still found within paragraphs.


Elements
Console
Sources
Network
Performance
Memory
Application

```
kground: url("files/kiwi/content/KIWI_Krisenseite/6_Kontakt/kiwi_kontakt_bg.j
8px;">
  ▼<div class="inner" style="display:table-cell;vertical-align:middle;">
    ▼<div class="innerWrap" style="max-width:960px;margin:0 auto;">
       <h2 class="ce_headline first"> Darf's ein bisschen mehr sein?</h2>
      ▼<div class="ce_text block">
         <h2>KIWI. Werbeagentur GmbH</h2>
        ▼<p> == $0
           "Bramscher Str. 14"
           <br>
           "49088 Osnabrück"
         </p>
        ▶<p>…</p>
        ▶<p>…</p>
```

er div#container div#main div.inside div#kontakt.asdf.mod_article.kontakt.last.default_layout.layout
18°C

4) In German, "contact" is called "kontakt," so it will be added to the list of keywords that I will explain later how I will use.

- ***Diving more in the German websites***

It seemed to be altogether different. Then I decided to access more random sites to see if it's just an exception or if there's a considerable number of German addresses. By selecting random sites, I arrived at several conclusions.

1) German websites are clearly not an exception; I found a minimum of 10.
2) I couldn't find any library that extracts German addresses, so I'll have to find regex patterns on my own to identify them.
3) I noticed that I encountered German websites at indices close to each other. Then I thought that perhaps the domains are arranged by country. By this, I mean that the first n are from the US, the next m are from Germany, and the remaining ones are from the UK. There might be some small exceptions, but mostly it seemed to be the case.

- ***Step 6: Dividing the domains***

Therefore, I kept trying URLs until I managed to divide the list of domains into 3 parts:
Part 1: Indices 0 -> 849 domains from the US
Part 2: Indices 850 -> 1621 domains from Germany
Part 3: Indices 1622 -> end of the list domains from the UK

So, I thought it would be best to extract addresses in 3 stages, according to the country, and then organize what I extracted into a document in order.

- ***Step 7: Analyzing more German addresses.***

Then I thought, if Germany seems problematic to me, it means I need to analyze more patterns, to examine even more websites to feel like I can handle any exception.

I noticed that German websites often have a page called "Impressum" where the address is always found. Then I searched online to see what "Impressum" means to understand why. The definition from https://termly.io/resources/articles/impressum/ is: *"The Impressum may be a largely unfamiliar concept in North America, but it's a legal requirement for businesses that operate in German-speaking countries."*

So, I added "Impressum" to the list of keywords. I noticed that addresses are also found on "Datenschutz" pages, which would translate to *"data privacy" or "data protection."* I added this to the list of keywords as well. Additionally, I thought that "home" would be a suitable word to add to the list of keywords, as addresses can also be found there.

- ***Step 8: Extracting the German addresses***

Now that I've clarified some things and feel like I have a better handle on German websites, the question remains: **how do I identify German addresses from the text?**

I tried to create a regex that would encompass the entire address, but there were always cases that I missed. I didn't find much on the internet, but I did come across a GitHub page: https://gist.github.com/0OZ/447ecddcbbe8e9bef546701822fb8cde where two regex patterns were offered. One is for *road and street numbers*, and the other is for *city and postcode*. It was exactly what I needed, the problem being that they were separate.

So, I thought to perform two extractions for both regex patterns and save everything I found into two lists. Initially, I thought that what's in the first list at index it corresponds to what's in the list at index i. Then I realized that there might be situations on the page where it's not necessarily an address but rather a street name, or perhaps a city and country.

So, I decided to do something that isn't necessarily the smartest or most efficient thing, but I knew for sure that I would hit the correct address: to take the Cartesian product between the lists and later take care of verifying if the address is real. Anyway, an address in general from a website isn't guaranteed to be real or valid.

- ***Step 9: Finding how to validate the addresses***

Now, the new question on my mind is: **How do I find out if an address is real/valid?** After some searching, I discovered the Geopy library, which offers everything on a silver platter. If the address is real and not complete, it completes it for you. Moreover, if you want, it even formats it for you by country, city, etc. It's everything I wanted. Everything sounds perfect and simple up to this point. However, the problems only start to arise now when you move on to the coding part, where things never go as planned.

- ***Step 10: Explaining the keywords***

Before diving into the coding part, let me explain why I chose these keywords. Throughout the analysis of the task, I realized that it wouldn't be enough to search for addresses only on the given domains, but rather to discover other pages of the website where addresses might be found. To avoid analyzing thousands of pages in vain, I thought to search on pages that have keywords from the selected list because there I noticed it's most likely to identify an address (I'm referring to something like: *www.domain/contact*, or *www.domain/connect-with-us*, *www.domain/impressum*, etc.).

- ***Step 11: Generate other URLs starting from an URL***

How do I find out for each site if there are URLs of this type? Naturally, I thought of using Web Scraping with Beautiful Soup. Essentially, in a page's menu or buttons to change the page, there is a link in the HTML content, or more precisely, an <a href>. So, I thought to extract all <a href> tags from the page's content and take the links from them, keeping only those that contain keywords (as described in the previous paragraph).

- ***Step 12: Doing this more efficiently***

I initially tried to test my code on 50 US URLs to see how things went. When I saw that it was taking a long time even for 50 URLs, I realized I needed to find a way to make this more efficient. **How did I do that?** By abandoning the synchronous processing of URLs one by one and performing all tasks on multiple URLs in parallel (asynchronous).

So, I transformed everything that could be transformed in my code to be asynchronous, except for address validation with Geopy because in *Geopy's policy*, you are allowed to make only one request per second, and what I wanted to do would violate this policy. Therefore, the longest runtime of the application is due to this method. The rest executes within seconds completely asynchronously.

Now that I've finished explaining everything I've thought about throughout this task, let's move on to the coding part, how I structured the code, and the difficulties encountered.

❖ CODING PART

- *SSL Errors*

The first thing I tried was to assign either https or http for each domain. Right from the first step, I encountered multiple errors, which I believe held me back for the most part of the time I spent on the project: SSL errors.

As soon as I got rid of one error, another one popped up. Interestingly, I didn't encounter these errors for the 50 domains selected for testing. I faced them when I gathered more from the list (100 URLs).

The first error was of this type:
`requests.exceptions.SSLError: [Errno 1] _ssl.c:1006: error:14090086:SSL routines:SSL3_GET_SERVER_CERTIFICATE:certificate verify failed`.

I resolved it by setting `verify = False` for the URL request, although it's not recommended. Initially, I tried to somehow obtain an SSL certificate, but I abandoned the idea as it took too much time.

Then, when I thought everything was resolved and I gathered the courage to test on all US domains, I encountered another SSL error:
`SSLError: [SSL: SSLV3_ALERT_HANDSHAKE_FAILURE] sslv3 alert handshake failure (_ssl.c:1006)`.

After some research, I resolved it by adding `return_exceptions=True`. Initially, I tried something with try-except blocks, but it didn't consider them at all.

After overcoming these errors, everything worked properly for each domain, determining whether it should be assigned http or https. Since I divided the domains into 3 parts, I ran this function 3 times, and each time it took approximately 11 seconds to run. So, we'll approximate the runtime for this to be 35 seconds.

- *Checking the URLs*

After constructing the URLs from domains, the next step is to test if they are functional. How do we determine functionality? Each URL returns a status code, and the functional ones will return a status of 200. A very surprising discovery was to see that out of 850 URLs, only approximately 200 remained functional.

The runtime for all three countries is approximately 35 seconds.

- Adding new URLs to the list that have keywords

keywords = ['connect', 'contact', 'kontakt', 'impressum', 'adresse', 'reach', 'datenschutz', 'home']

At this step, I made sure to do everything necessary to avoid SSL errors as well. Here, I encountered another error. For certain websites, the utf-8 text type wasn't suitable, so I had to transform it into latin-1 to make it work. After that, everything worked as expected, generating new links with the keywords from the list.

After that, I filtered the URLs again to keep only the functional ones because if the main page is functional, it doesn't guarantee that the rest of the pages are also guaranteed to be functional.

It's worth mentioning that there are URLs that don't get verified because they take too long to load, so I added a timeout after which, if they don't load, they're abandoned and considered invalid, then removed from the list.

The runtime for all three countries is approximately 35 seconds.

- Saving the data to have it permanently and not lose it in case we want to resume the code from a certain point

I decided that it would be best to save the URLs in a CSV file in case something doesn't go as planned in the subsequent steps, so that I can extract them at any time for use in the following steps. As I mentioned earlier, I divided the code execution by countries. Therefore, for each country, there is a separate CSV with its URLs.

- Extracting the address from the text of a URL page

I couldn't create a single function to extract the address from the text of a page because this process differs for each country. So, I have 3 functions. Two of them, for the US and UK, are almost identical, with the difference being that in pyap, the specified country changes for address identification.

On the other hand, the function for Germany is quite different, where paragraphs are extracted in a special way, and the two regex patterns are searched for twice in them, returning the Cartesian product. I noticed that these regex patterns also extract very long phrases that are not addresses, so to streamline the process a bit, I added a condition that the final Cartesian product should have a maximum of 7 words; otherwise, it's definitely not an address.

Each function individually takes approximately 11-12 seconds because the addresses are extracted asynchronously. We'll approximate it to 35 seconds.

- Checking if the address is valid + parsing the address

This function takes the longest because it runs synchronously, and after each request, there is a 2-second sleep to ensure that there are no overlapping requests to avoid violating Geopy's policy of one request per second.

So, for the US and UK, it takes approximately 15 minutes each, while for Germany, the process is much longer because many of the addresses in the list are not valid but were extracted due to the Cartesian product. For Germany, it took approximately 60 minutes, so the final runtime is one and a half hours.

I should mention that this time is not continuous because the addresses are extracted in 3 batches, depending on the country.

- Conclusions

The code execution process is very simple. You just need to click "run" in order on each code cell in the file. The advantage of this coding mode is that when you revisit it, you can run only a specific sequence for a particular country from it; you're not obligated to run the entire code. Additionally, another advantage is that the runtime can be fragmented; continuous running is not necessary. You can come back on another day to run the code for another country, for example. To be precise, the approximate total runtime of this application is one hour and 35 minutes, during which 702 addresses were extracted, with the mention that there is a possibility that some may be repeated, especially in the case of Germany.