GW2-SRS EXTRACT

powered by LaTeX

Daniel Lopez: **Extraction algorithm**

November 7, 2022

1.0 EXTRACT

1.1 Introduction

As the first part of the **ETL** it's important to have in mind our data sources. When I began with the project I only had one way of getting logs, and they were all coming from friends. This meant I had to manually parse them with EliteInsights[1].

This process was extremely slow, and when at first it was viable due to having few logs, it started being quite useless when the project meant to have lots of logs to analyze. The best option then was thinking on using web-scraping or an API connection; and thankfully we have a web-page available, thanks to *Johannes Pfau*, where people can upload their logs.

GW2-Wingman is probably the best approach I made to this project, not only in efficiency, but also in learn curve. Scraping was something I craved for since I started focusing on the Data Engineering route. It taught me a lot of things and problems I could solve, some easier, some harder, but all of them were at the end solved.

[1]EliteInsights it's an app designed to parse *.zevtc* files into JSON, HTML or CSV files.

1.2 About GW2-Wingman architecture

As I was learning to scrape the web-page I realized something wasn't right, and it made the scrape a bit harder when I understood the data was stored inside an **iframe tag**[2], so it was a matter of getting to the actual web-page.

It was quite simple in the end, the iframe had an **href** tag containing the actual path, which only added */logContent* before the log name.

This is a comparison of the actual URL showed, with the nested one:

```
1    https://gw2wingman.nevermindcreations.de/log
2
3    https://gw2wingman.nevermindcreations.de/logContent
4
```

1.3 JSON data gathering

Extract algorithms are rather simple once you understand what you need to find, and that was the hardest part, since it required looking into the actual HTML code. I tackled the extraction with BeautifulSoup[3] by passing an URL and the needed headers.

After that I just needed to get the correct script and extract the JSON. This required a bit of research; the JSON was nested on a JavaScript variable, which I had to strip off, and save only the JSON data.

```
1    response = requests.get(url=url,headers=HEADERS)
2    soup = BeautifulSoup(response.content,'html.parser')
3
4    data = soup.find_all('script')[8]
5    dataString = data.text.rstrip()
6
```

[2] An iframe is a nested web inside the main website

[3] As well as Selenium, BeautifulSoup let us extract information from any web-page.

Last thing I needed to do is set up an algorithm, or in this case, a class, that help us know the boss Name and create a tag with it. This will make the further processes so much agile:

```
    class boss:

        def __init__(self, url: str) -> str:
            self.url = url

        def getBossName(self) -> str:
            url = self.url
            urlLines = url.split('/')
            if len(urlLines) < 5:
                bossName = urlLines[3]
            elif len(urlLines) == 5:
                bossName = urlLines[4]
            return bossName

        def getBossTag(self, bossName: str) -> str:
            bossTag = bossName.split('_')
            nameTag = bossTag[1]
            return nameTag

```

1.4 URL copy code

Ending with the extraction part of the project, I will explain a quite relevant part here. Normally we could extract information URL by URL, but it is not really convenient when we have, like in this case, more than 100 logs. Therefore, best option is creating a script that copy and write down this URLs for us on a text file we can later read.

- I set the search method to scope the 'a' tags, so I only had to just take the href of each one and add the url_str and have the complete link for a later iteration.
- Nonetheless, the 'a' tag contains about 5-6 lines that are not log hrefs, instead, they show an apikey href and JavaScript void hrefs, so I just omitted them and set a new line for each one the script finds.

- Finally, I wrote it down on a .txt file; however I realized that, at first I was using 'w' mode, but it ended being quite inefficient since it constantly overwrites the file. To avoid this, I just simply changed the 'w' mode to 'a'[4] mode, this way we can just append the information and not worry about it overwriting over and over again.

This is how the code looks like:

```
fh = open(path,'a')

for link in soup.find_all('a'):
    url_str = 'https://gw2wingman.nevermindcreations.de'
    data = link.get('href')

    try:
        log_str = url_str+data
        if log_str.endswith('apikey'):
            log_str.replace('apikey','\n')
        elif log_str.endswith('void(0)'):
            log_str.replace('void(0)','\n')
        else:
            fh.write(log_str)
            fh.write('\n')
    except Exception as e:
        print('Error: ', str(e))

```

[4] An 'a' stands for append, and will add new data without overwriting the existing one