



Grab-Food Scraper Assignment | Anakin

By

M.SAKSHAM

Problem Statement

Develop a Python-based program to automate the scraping of restaurant listings from Grab Food Delivery, aiming to extract comprehensive details about restaurants for a selected location. The focus is on capturing dynamic content loaded through XHR requests, including names, cuisines, ratings, delivery times, and promotional offers.

Website

<https://food.grab.com/sg/en/restaurants>

Location

Location used: PT Singapore - Choa Chu Kang North 6, Singapore, 689577

Set-Up

Install Libraries: Ensure selenium and webdriver_manager are installed for browser automation.

Initialize ChromeDriver: Use ChromeDriverManager to automatically handle the ChromeDriver setup, allowing programmatic control of the Chrome browser.

```
# -*- coding: utf-8 -*-  
"""  
Created on Sun Mar 10 14:50:35 2024  
  
@author: M.SAKSHAM  
GitHub: Saksham093
```

```

"""

import gzip
import json
from time import sleep
from selenium import webdriver
from seleniumwire import webdriver
from selenium.webdriver.common.by import By
from seleniumwire.utils import decode as sw_decode
from selenium.webdriver.chrome.service import Service
from selenium.common.exceptions import TimeoutException
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.support.wait import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

```

I used Spyder IDE.

My Approach



The complete code is divided into two parts,

- Driver Class
- XHR requests and responses Class


Driver Class

```

# Driver class
class Driver:
    def __init__(self) -> None:
        self.browser = None
        self.setup()

    # Setup the browser
    def setup(self):
        chrome_opts = webdriver.ChromeOptions()

```



```

        chrome_opts.headless = True
        user_agent = ('Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36
(KHTML, like Gecko) ' +
                    'Chrome/60.0.3112.50 Safari/537.36') # Set the user
agent
        chrome_opts.add_argument(f'user-agent={user_agent}') # Add the
user agent
        chrome_opts.add_argument('--no-sandbox') # Disable sandbox
        chrome_opts.add_argument("--disable-extensions") # Disable
extensions
        chrome_opts.add_argument('--disable-dev-shm-usage') # Disable
shared memory

        # Set up ChromeDriver
        service_path = Service(ChromeDriverManager().install())
        self.browser = webdriver.Chrome(service=service_path, options =
chrome_opts)

        # Close the browser
        def tear_down(self):
            self.browser.quit()

```

The **Driver class** manages automated web browser sessions. Here's how it works:

Initialization (__init__):

Sets up a browser session when a `Driver` instance is created by calling **setup**.

Setup(setup) :


Configures a headless Chrome browser with specific settings (like a custom user agent and disabling unnecessary features) for automation tasks. It uses

ChromeOptions for configurations and **ChromeDriverManager** to automatically handle driver requirements.

```
Clean-up (tear_down):
```

Closes the browser session to free resources.

XHR requests and responses Class



```
# Class for the XHR requests and responses
class Scraper:
    # initialize the scraper
    def __init__(self, driver: Driver, base_url) -> None:
        self.driver = driver # initialize the driver
        self.base_url = base_url # initialize the base url
        self.grab_internal_post_api =
"https://portal.grab.com/foodweb/v2/search" # initialize the grab-foods
internal post api
        self._init_request() # initialize the request

    def _init_request(self):
        self.driver.browser.get(self.base_url)
        sleep(60)

    def load_more(self):
        del self.driver.browser.requests # to clear previously captured
requests

        try:
            last_height = self.driver.browser.execute_script("return
document.body.scrollHeight")

            while True:
```

```

        # Scroll down to the bottom of the page
        self.driver.browser.execute_script("window.scrollTo(0,
document.body.scrollHeight);")

        # Wait for new data to load
        WebDriverWait(self.driver.browser, 60,
poll_frequency=1).until(
            lambda driver: driver.execute_script("return
document.body.scrollHeight") > last_height)

        new_height = self.driver.browser.execute_script("return
document.body.scrollHeight")
        if new_height == last_height:
            break
        last_height = new_height

        sleep(60)

    except TimeoutException:
        print("No more content to load or page took too long to
respond.\n")

    def capture_post_response(self): # capture the post response
        post_data = []

        for r in self.driver.browser.iter_requests():
            if r.method == 'POST' and r.url ==
self.grab_internal_post_api: # capture the post response

                data_1 = sw_decode(r.response.body,
r.response.headers.get('Content-Encoding', 'identity')) # decode the
response

                data_1 = data_1.decode("utf8")

                data = json.loads(data_1) # convert the response to json
                post_data.append(data)

        # print(post_data)

```





```

        return post_data

    def get_restaurant_data(self, post_data):
        d = {}

        for p in post_data: # get the restaurants data
            l = p['searchResult']['searchMerchants'] # list of
restaurants
            for rst in l: # for each restaurant
                try:
                    d[rst['id']] = {'restaurantName': rst.get('address',
{}).get('name', ""),
                                'restaurantCuisine':
rst.get('merchantBrief', {}).get('cuisine', ""),
                                'restaurantRating':
rst.get('merchantBrief', {}).get('rating', ""),
                                'estimateDeliveryTime':
rst.get('estimatedDeliveryTime', ""),
                                'distanceFromLocation':
rst.get('merchantBrief', {}).get('distanceInKm', ""),
                                'promotionalOffers':
rst.get('merchantBrief', {}).get('promo', {}).get('description', ""),
                                'noticeIfVisible':
rst.get('merchantBrief', {}).get('closingSoonText', ""),
                                'imageLink': rst.get('merchantBrief',
{}).get('photoHref', ""),
                                'isPromoAvailable':
rst.get('merchantBrief', {}).get('promo', {}).get('hasPromo', ""),
                                'restaurantID': rst.get('id', ""),
                                'latlng': rst.get('latlng', ""),
                                'estimateDeliveryFee':
rst.get('estimatedDeliveryFee', {}).get('priceDisplay', "")
                                }

                except Exception:
                    print(rst)

        return d

```

```

def scrape(self):
    self.load_more() # load more restaurants in the list
    post_data = self.capture_post_response() # capture the post
response
    restaurants_data_ = self.get_restaurant_data(post_data) # get the
restaurants data
    return restaurants_data_

def save(self, restaurants_latlng, file: str =
'grab_restaurants_latlng.json'):
    with open(file, 'w') as f:
        json.dump(restaurants_latlng, f, indent=4)

```

The **Scraper Class** is designed for handling web scraping tasks, specifically for capturing XHR requests and responses from a web application.



Initialization(__init__): Sets up the scraper with a web driver and a base URL. It also specifies an internal API URL for grabbing data from a food delivery platform.

Initializing Request(_init_request): Opens the base URL in the browser and waits for 60 seconds, allowing the page to load fully, including any dynamic content.

Loading More Content(load_more): Scrolls the web page to load more content dynamically. It clears previous requests, scrolls to the bottom, wait for new content to load, and repeats until no more new content is loaded.

Capturing POST Response(capture_post_response): Looks through all network requests made by the browser, filters for POST requests to the

specified API, decodes, and converts the response from JSON, collecting the data.

Extracting Restaurant Data(get_restaurant_data): Processes the captured data to extract detailed information about restaurants, such as name, cuisine, rating, delivery time, and promotional offers.

Scraping(scrape): the scraping process by calling methods to load more content, capture responses, and extract restaurant data, returning the compiled data.

Saving Data(save): Saves the collected restaurant data to a ndJSON file for later saving in gzip format.



Overall Approach and Methodology

- **Data Extraction:** A custom `Scraper` class to automate the browser, navigate to the target URL, and interact with the page to load dynamic content.
- **Data Processing:** Post-extraction, data was processed to filter out and structure restaurant details, including names, cuisines, ratings, and other relevant information.
- **Quality Control (QC):** Performed QC to validate the accuracy, completeness, and consistency of the data.

This involved:

- **Total Count Verification**
- **Null Checks**

- **Data Structuring:** Converted the final dataset into NDJSON format and compressed it using gzip for efficient storage and access.

Challenges Faced

- **Dynamic Content Loading:** The main challenge was managing dynamic content that loads as the user scrolls, requiring the implementation of a sophisticated scrolling mechanism in the scraper.
- **Rate Limiting and Blocks:** Encountered issues with request rate limiting and IP blocking.
- **Data Inconsistency:** Some inconsistencies in the structure of the data returned by the API made parsing and normalization a challenge, requiring adaptive data processing logic.

Improvements and Optimizations

- **Improved Error Handling:** Implement more robust error handling.
- **Efficiency Enhancements:** Optimize the scrolling and data loading mechanism.
- **Data Validation:** Enhance data validation steps to automatically flag discrepancies or outliers in the dataset, ensuring high data quality.



Thank You
