Rapport : Web Scraping de données Twitter avec Playwright

Ayoub Abraich

1. Introduction

Ce projet avait pour objectif de développer une solution permettant d'extraire des données à partir de comptes Twitter spécifiques, sans utiliser l'API officielle de Twitter. Le web scraping représente une approche efficace pour collecter des données à partir de sites web dynamiques tels que Twitter.

L'approche adoptée consistait à créer un scraper Python capable d'extraire diverses données comme le contenu des tweets, les métriques des comptes et des tweets, ainsi que de suivre leur évolution sur une période de 7 jours.

2. Technologie et outils utilisés

Playwright

Playwright est une bibliothèque Python permettant l'automatisation de navigateurs web et le scraping de sites dynamiques. Elle offre une approche puissante pour interagir avec des pages web riches en JavaScript, en permettant l'exécution de scripts et la simulation d'interactions utilisateur avancées.

Playwright a été choisi pour ce projet en raison de ses fonctionnalités avancées et de sa facilité d'utilisation pour automatiser les interactions avec les pages Twitter.

Autres outils

- **Python** : Langage de programmation principal utilisé pour le développement du scraper.
- **BeautifulSoup** : Bibliothèque Python pour le parsing et l'extraction de données à partir du DOM.
- **Pydantic** : Bibliothèque de validation de données utilisée pour définir les modèles de données.
- **FastAPI** : Framework web Python utilisé pour créer une API RESTful autour du scraper.
- **Schedule** : Pour automatiser l'exécution du scraper à intervalles réguliers, nous avons utilisé la bibliothèque schedule de Python. Cette bibliothèque permet de planifier l'exécution de fonctions ou de tâches à des moments précis ou selon une fréquence définie.

3. Méthodologie

La méthodologie mise en œuvre pour le scraping de données Twitter impliquait plusieurs étapes clés :

5. **Stockage des données** : Les données extraites ont été structurées et stockées au format JSON, facilitant leur utilisation ultérieure.

4. Implémentation

Structure du projet

Le projet est structuré comme suit :

```
.
├── twitter_scraper.py
├── requirements.txt
└── twitter_scraper.log
```

- `twitter_scraper.py` : Fichier Python principal contenant le code du scraper et de l'API FastAPI.
- `requirements.txt` : Fichier listant les dépendances Python nécessaires.
- `twitter_scraper.log` : Fichier de journalisation des événements du scraper.

Classes principales

`User`

Modèle de données Pydantic représentant les informations d'un utilisateur Twitter, telles que le nom d'utilisateur et le nombre d'abonnés.

`Tweet`

Modèle de données Pydantic représentant les informations de base d'un tweet, comme l'identifiant, la date et le lien.

`TweetDetails`

Modèle de données Pydantic représentant les détails complets d'un tweet, y compris le contenu textuel, la date, les métriques (retweets, mentions, likes), et le lien.

`TwitterScraper`

Classe principale implémentant la logique de scraping. Voici quelques méthodes clés :

- `__init__` : Initialise le scraper avec une liste de comptes Twitter à scraper.
- `setup_logger` : Configure le logger pour enregistrer les événements de scraping.
- `wait_scroll_to_bottom` : Attend le chargement complet de la page en scrollant vers le bas.

```
.
├── twitter_scraper.py
├── api.py
├── scedule.py
├── twitter_scraper.log
└── requirements.txt
```

- `api.py` :

```python
import uvicorn
from fastapi import FastAPI
from twitter_scraper import TwitterScraper
from pydantic import HttpUrl
from typing import List
from twitter_scraper import User, Tweet, TweetDetails

# ------------------
#  Partie API FastAPI
# ------------------

# Création d'une instance de FastAPI
app = FastAPI()

# Liste des comptes Twitter à scraper
accounts = [
    "https://twitter.com/RevolutApp",
    "https://twitter.com/mabanque_bnpp?lang=fr",
    "https://twitter.com/Leocare_Assure"
]

# Création d'une instance de TwitterScraper avec les comptes Twitter à scraper
scraper = TwitterScraper(accounts)

# Route racine de l'API
@app.get("/")
def read_root():
    return {"message": "Welcome to the Twitter Scraper API!"}

# Route pour obtenir les informations d'un utilisateur Twitter
@app.get("/user/", response_model=User)
def get_user(account_url: HttpUrl):
    return scraper.get_user(account_url)

# Route pour obtenir les informations de plusieurs utilisateurs Twitter
@app.get("/users/", response_model=List[User])
def get_users():
    return scraper.get_users()
```

```python
# Importation des librairies nécessaires

import logging
import time
from playwright.sync_api import sync_playwright
from pydantic import BaseModel
from typing import List


# ------------------
#  Partie Scraper
# ------------------
# Liste des comptes Twitter à scraper
accounts = [
    "https://twitter.com/RevolutApp",
    "https://twitter.com/mabanque_bnpp?lang=fr",
    "https://twitter.com/Leocare_Assure"
]


# Définition de la classe User
class User(BaseModel):
    username: str
    follower_count: str

# Définition de la classe Tweet
class Tweet(BaseModel):
    id: int
    date: str
    link: str

# Définition de la classe TweetDetails
class TweetDetails(BaseModel):
    id: str
    tweet_body: str
    tweet_date: str
    retweet: str
    mention: str
    like: str
    url: str

# Définition de la classe TwitterScraper
class TwitterScraper:
    def __init__(self, accounts):
        self.accounts = accounts
        self.logger = self.setup_logger()
        self.page = None
```

```python
    # Méthode pour obtenir les données d'un utilisateur
    def get_user(self, account_url: str) -> User:
        username = account_url.split("/")[-1]

        with sync_playwright() as p:
            self.logger.info(f"Scraping data for {account_url}")
            browser = p.chromium.launch()
            self.page = browser.new_page()
            self.page.goto(account_url)

            follower_count_element = self.page.wait_for_selector("//*[@id='react-root']/div/div/div[2]/main/div/div/div/d
            follower_count = follower_count_element.inner_text().strip()

            browser.close()
            user = User(username=username, follower_count=follower_count)

            self.logger.info(f"Data scraped for {account_url}: {user}")

            return user

    # Méthode pour obtenir les données de plusieurs utilisateurs
    def get_users(self) -> List[User]:
        users = []
        for account in self.accounts:
            user = self.get_user(account)
            users.append(user)
        return users

    def get_followers(self, account_url: str) -> List[User]:
        pass

    def get_following(self, account_url: str) -> List[User]:
        pass

    def get_tweets(self, account_url: str) -> List[str]:
        with sync_playwright() as p:
            self.logger.info(f"Scraping tweets for {account_url}")
            browser = p.chromium.launch()
            self.page = browser.new_page()
            self.page.goto(account_url)
            self.wait_scroll_to_bottom()

            tweet_elements = self.page.query_selector_all("a[href*='/status']")
            tweet_elements = [tweet for tweet in tweet_elements if "analytics" not in tweet.get_attribute("href")]
            tweets = []
            for tweet_element in tweet_elements:
                try:
                    tweet_link = 'https://twitter.com' + tweet_element.get_attribute("href")
```

```
            tweet_body = self.page.query_selector("div[lang='en'] > span").inner_text()
            tweet_date = self.page.query_selector("time").inner_text()

            inf = self.page.query_selector_all("span[data-testid='app-text-transition-container']")
            try:
                retweet = inf[0].inner_text()
                mention = inf[1].inner_text()
                like = inf[2].inner_text()
            except IndexError:
                retweet, mention, like = 0, 0, 0

            tweet_details = TweetDetails(id=tw_id, tweet_body=tweet_body, tweet_date=tweet_date, retweet=retweet, mention

            self.logger.info(f"Data scraped for {url}: {tweet_details}")

            return tweet_details

    def get_hashtags(self, account_url: str) -> List[str]:
        pass

    def get_media(self, account_url: str) -> List[str]:
        pass

    def get_likes(self, account_url: str) -> List[str]:
        pass

    def run(self):
        self.get_users()
        self.get_tweets(self.accounts[0])
        self.get_tweet_details(self.accounts[0], "1763611561493491828")


# Exécution du scraper

scraper = TwitterScraper(accounts)

scraper.run()
```

Suivi des données sur 7 jours avec Cron Job Au lieu d'utiliser un cron job système, nous avons opté pour une approche basée sur un script Python utilisant les bibliothèques `schedule` et `time`. Cette approche offre une plus grande flexibilité et facilite l'intégration de la planification de tâches directement dans le code du scraper.

Voici un exemple d'implémentation :

```
import schedule
import time
```

```
    schedule.run_pending()
    time.sleep(1)
```

5. Résultats et performances

Le scraper a été capable d'extraire avec succès les données souhaitées à partir des comptes Twitter ciblés, y compris le contenu des tweets, les métriques des comptes et des tweets, ainsi que leur évolution sur une période de 7 jours.

Exemple de sortie JSON pour un tweet :

```
{
  "id": "1234567890",
  "tweet_body": "Ceci est un exemple de tweet extrait par le scraper.",
  "tweet_date": "23 mai 2023",
  "retweet": "42",
  "mention": "12",
  "like": "237",
  "url": "https://twitter.com/CompteExemple/status/1234567890"
}
```

`twitter_scraper.log`

```
2024-03-04 11:40:28,779 - INFO - Data scraped for https://twitter.com/RevolutApp: username='RevolutApp' follower_count='3
2024-03-04 11:40:29,083 - INFO - Scraping data for https://twitter.com/mabanque_bnpp?lang=fr
2024-03-04 11:40:32,519 - INFO - Data scraped for https://twitter.com/mabanque_bnpp?lang=fr: username='mabanque_bnpp?lang
2024-03-04 11:40:32,806 - INFO - Scraping data for https://twitter.com/Leocare_Assure
2024-03-04 11:40:37,138 - INFO - Data scraped for https://twitter.com/Leocare_Assure: username='Leocare_Assure' follower_
2024-03-04 11:40:37,436 - INFO - Scraping tweets for https://twitter.com/RevolutApp
2024-03-04 11:41:10,913 - INFO - Scraped 7 tweets for https://twitter.com/RevolutApp
2024-03-04 11:41:11,234 - INFO - Scraping data for https://twitter.com/RevolutApp/status/1763611561493491828
2024-03-04 11:41:15,248 - INFO - Data scraped for https://twitter.com/RevolutApp/status/1763611561493491828: id='17636115
```

Le scraper s'est montré fiable et robuste, gérant correctement les pages dynamiques de Twitter et les chargements progressifs de contenu. Cependant, des améliorations sont encore possibles pour augmenter les performances et éviter les bannissements potentiels.

6. Défis rencontrés et solutions apportées

Au cours du développement du scraper, plusieurs défis techniques ont été rencontrés :

Gestion des pages dynamiques

Pour suivre l'évolution des métriques sur une période de 7 jours, il a fallu exécuter le scraper quotidiennement et stocker les données de manière structurée.

Solution : Une boucle principale a été implémentée pour exécuter le scraper sur chaque compte Twitter, stockant les données extraites au format JSON structuré à chaque exécution.

7. Améliorations futures

Bien que le scraper actuel soit fonctionnel, plusieurs améliorations sont envisageables pour augmenter ses performances et sa fiabilité :

Rotation des IP et utilisation de proxies

Pour éviter les bannissements potentiels de Twitter, une rotation d'adresses IP et l'utilisation de proxies pourraient être mises en place. Cela permettrait de masquer l'origine des requêtes et de contourner les limitations imposées par Twitter.

Simulation de navigateur réaliste

Afin de rendre le scraping plus discret et moins détectable, une simulation de comportement de navigation réaliste pourrait être implémentée. Cela impliquerait des délais aléatoires entre les actions, des mouvements de souris simulés, et d'autres techniques visant à imiter le comportement d'un utilisateur humain.

Tests unitaires avec PyTest

Pour assurer la qualité du code et faciliter la maintenance, des tests unitaires devraient être développés avec PyTest. Cela permettrait de détecter rapidement les régressions et de garantir le bon fonctionnement du scraper après chaque modification.

Conteneurisation avec Docker

Pour faciliter le déploiement et l'exécution du scraper dans différents environnements, une conteneurisation avec Docker pourrait être envisagée. Cela permettrait de packager l'application et toutes ses dépendances dans un conteneur léger et portable.

Intégration continue et déploiement continu (CI/CD)

Afin d'automatiser les processus de build, de test et de déploiement, une intégration continue et un déploiement continu (CI/CD) pourraient être mis en place. Des outils comme GitHub Actions ou Jenkins pourraient être utilisés pour gérer ces processus de manière automatisée.

8. Conclusion

Ce projet a permis de développer une solution efficace pour le scraping de données à partir de comptes Twitter, sans utiliser l'API

Section 2 : Web Scraping et traitement de données

Ayoub Abraich

1. Extraction de données

Pour extraire les données des pages web mentionnées, j'ai créé un script Python appelé `web_scraper.py`. Ce script utilise les bibliothèques `requests` pour récupérer le contenu HTML des pages, `BeautifulSoup` pour parser le HTML et `urllib.parse` pour manipuler les URLs.

La classe `WebScraper` a été implémentée avec les méthodes suivantes :

- `__init__` : initialise la classe avec une liste d'URLs à scraper et un dictionnaire vide pour stocker les données.
- `_get_base_url` : extrait l'URL de base à partir d'une URL donnée.
- `_extract_documents` : extrait les liens vers des documents PDF de la page HTML.
- `_extract_paragraphs` : extrait les paragraphes contenant le mot "taux" de la page HTML.
- `_extract_conditions` : filtre les liens de documents contenant le mot "condition".
- `extract_content` : méthode principale qui récupère le contenu HTML d'une URL, appelle les méthodes d'extraction et stocke les données dans le dictionnaire.
- `get_data_all_urls` : boucle sur la liste d'URLs pour extraire les données de toutes les pages.
- `get_data` : extrait les données d'une seule URL donnée.

Voici un exemple d'utilisation :

```
from web_scraper import WebScraper

list_of_urls = [
    "https://www.creditmutuel.fr/fr/particuliers/epargne/livret-de-developpement-durable.html",
    "https://www.monabanq.com/fr/produits-bancaires/livret-developpement-durable/en-resume.html",
    "https://www.banquepopulaire.fr/bpaura/epargner/livret-transition-energetique/"
]

scraper = WebScraper(list_of_urls)
output_dict = scraper.get_data_all_urls()
print(output_dict)
```

2. Création d'APIs/Web Services

J'ai créé une API RESTful en utilisant le framework FastAPI. Le fichier `main.py` contient le code de l'API

3. Base de données NoSQL

J'ai choisi d'utiliser RavenDB comme base de données NoSQL pour stocker les données extraites. RavenDB est une base de données orientée documents qui convient bien pour stocker des données semi-structurées comme celles issues du web scraping.

Voici un exemple de schéma pour stocker les données extraites dans RavenDB :

```
{
    "id": "documents/1",
    "bank_name": "Credit Mutuel",
    "url": "https://www.creditmutuel.fr/fr/particuliers/epargne/livret-de-developpement-durable.html",
    "documents": [
        "https://www.creditmutuel.fr/documents/pdf/livret-developpement-durable.pdf",
        "https://www.creditmutuel.fr/documents/pdf/conditions-generales.pdf"
    ],
    "taux": [
        "Le taux d'intérêt annuel brut est de 0,75% au 01/01/2023.",
        "Le taux d'intérêt est révisable chaque année par le Crédit Mutuel."
    ],
    "conditions": [
        "https://www.creditmutuel.fr/documents/pdf/conditions-generales.pdf"
    ]
}
```

Ce schéma représente un document RavenDB contenant les informations suivantes :

- `id` : identifiant unique du document
- `bank_name` : nom de la banque
- `url` : URL de la page web scrapée
- `documents` : liste des liens vers les documents PDF
- `taux` : liste des paragraphes contenant le mot "taux"
- `conditions` : liste des liens vers les documents contenant le mot "condition"

Pour importer les données dans RavenDB, j'ai créé une classe `RavenDB` dans `main.py` qui initialise une connexion avec la base de données et fournit des méthodes pour ouvrir une session et stocker des données.

L'endpoint `/scrape/` de l'API utilise cette classe pour stocker les données extraites dans la base de données RavenDB.

4. Git et Collaboration

J'ai créé un dépôt GitHub pour partager le code de cette solution : https://github.com/zetaneh/projet_test/

Le dépôt contient les fichiers suivants :

```python
from pydantic import BaseModel
from ravendb import DocumentStore
import uvicorn
from web_scraper import *

app = FastAPI()

list_of_urls = [
    "https://www.creditmutuel.fr/fr/particuliers/epargne/livret-de-developpement-durable.html",
    "https://www.monabanq.com/fr/produits-bancaires/livret-developpement-durable/en-resume.html",
    "https://www.banquepopulaire.fr/bpaura/epargner/livret-transition-energetique/"
]
# Define the RavenDB class
class RavenDB:
    def __init__(self, urls, database):
        self.store = DocumentStore(urls=urls, database=database)
        self.store.initialize()

    def open_session(self):
        return self.store.open_session()

    def close(self):
        self.store.dispose()

# Define the model for incoming data
class BankData(BaseModel):
    bank_name: str
    data: str


# Initialize RavenDB
urls = ["http://localhost:8080"]
database = "BankData"
raven = RavenDB(urls, database)



# root endpoint
@app.get("/")
async def root():
    return {"message": "Welcome to the Bank Data API"}

# FastAPI endpoint to scrape data from URLs and store in RavenDB
@app.post("/scrape/")
async def scrape_and_store_data(data: BankData):
    with raven.open_session() as session:
        session.store(data.dict(), collection_name="BankData")
        session.save_changes()
        return {"message": "Data scraped and stored successfully"}
```

```
    scraper = WebScraper(list_of_urls)
    output_dict = scraper.get_data(url)
    return output_dict



# Run the FastAPI server
if __name__ == "__main__":
    scraper = WebScraper(list_of_urls)
    uvicorn.run(app, host="127.0.0.1", port=8000)
```

- `web_scraper.py` :

```
import requests
from bs4 import BeautifulSoup
from urllib.parse import urljoin, urlparse

class WebScraper:
    def __init__(self, list_of_urls):
        self.list_of_urls = list_of_urls
        self.data = {}

    def _get_base_url(self, url):
        parsed_url = urlparse(url)
        return parsed_url.scheme + "://" + parsed_url.netloc

    def _extract_documents(self, soup, base_url):
        links = soup.find_all("a", href=True)
        documents = [urljoin(base_url, l.get("href")) for l in links if 'pdf' in l.get("href")]
        return documents

    def _extract_paragraphs(self, soup):
        paragraphs = soup.find_all("p")
        taux_paragraphs = [p.text.strip() for p in paragraphs if 'taux' in p.text.strip().lower()]
        return taux_paragraphs

    def _extract_conditions(self, documents):
        return [doc for doc in documents if 'condition' in doc]

    def extract_content(self, url):
        try:
            response = requests.get(url)
            if response.status_code == 200:
                soup = BeautifulSoup(response.content, "html.parser")
                base_url = self._get_base_url(url)
                documents = self._extract_documents(soup, base_url)
```

```
        return self.data[url]

if __name__ == "__main__":
    list_of_urls = [
        "https://www.creditmutuel.fr/fr/particuliers/epargne/livret-de-developpement-durable.html",
        "https://www.monabanq.com/fr/produits-bancaires/livret-developpement-durable/en-resume.html",
        "https://www.banquepopulaire.fr/bpaura/epargner/livret-transition-energetique/"
    ]

    scraper = WebScraper(list_of_urls)
    output_dict = scraper.get_data_all_urls()
    print(output_dict)
```