

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

```

- `api.py` :

```

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_bnp?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()

```

```
# 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/ReolutApp",
    "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
```

```

# 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.page.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=mention, like=like)

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_bnpplang=fr
2024-03-04 11:40:32,519 - INFO - Data scraped for https://twitter.com/mabanque_bnpplang=fr: username='mabanque_bnpplang
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