latex@warning@no@lin...

23 octobre 2024

Table des matières

Introduction

Bienvenue dans ce tutoriel, où nous allons vous faire decouvrir comment créer un projet de *scraping* de données à partir d'un site web, en les stockant dans une base de données MySQL, puis en les affichant via un site web interactif développé avec *Streamlit*.

Si vous vous êtes déjà demandé comment collecter des données de manière automatisée à partir d'un site web, les organiser proprement dans une base de données, et créer un site web dynamique pour les rendre accessibles et exploitables, vous êtes au bon endroit ! Ce projet vous guidera pas à pas dans le *scraping* du site Auchan , le stockage de ces données dans une base de donnee AWS et la création d'une interface web qui fera office de Dashbord.

Nous nous appuierons sur des outils comme Scrapy pour le *scraping*, MySQL pour la gestion des données, et Streamlit pour la création d'un site web interactif.

Ce tutoriel est le fruit de deux semaines de documentation et d'expérimentations intensives. Durant cette période, nous avons exploré divers aspects du scraping et de la programmation web, surmontant de nombreux défis techniques. Chaque obstacle a été une opportunité d'apprentissage, nous permettant de développer des compétences solides et de partager nos découvertes. Nous espérons que ce guide, né de nos efforts collectifs, vous sera utile dans vos propres projets.

0.1. Scraping des données depuis le site Auchan

Avant de plonger dans la pratique, il est important de comprendre ce qu'est le *scraping*

? Mais au fait, qu'est-ce que le scraping ?

Le *scraping* de données consiste à extraire des informations directement à partir des pages web. En d'autres termes, nous allons parcourir les pages d'un site, comme le ferait un utilisateur lambda , sauf que nous utiliserons du code pour capturer et enregistrer les données.

? Pourquoi utiliser le *scraping* ?

Les raisons peuvent être nombreuses : obtenir des informations qui ne sont pas facilement accessibles via des API, automatiser la collecte de données, ou encore extraire en masse des produits, des prix, des commentaires, etc.

Attention toutefois ! Le *scraping* doit être effectué dans le respect des règles d'utilisation des sites web. Chaque site possède généralement un fichier *robots.txt* qui définit ce qui est autorisé ou non. Nous devons donc nous assurer que notre activité de scraping est légale et respectueuse des termes du site.Nous auront l'occasion d'en reparler

Dans notre cas, nous souhaitons récupérer un maximum d'informations sur le site d'Auchan. Avec plus de 3 000 articles actuellement en vente, vous conviendrez qu'il est plus efficace de confier l'extraction des prix à un programme informatique. Parlons justement de ce programme ! Nous avons naturellement choisi Python pour plusieurs raisons : son accessibilité, sa facilité d'utilisation et ses nombreuses bibliothèques qui ont déjà fait leurs preuves dans le domaine du web scraping. À propos des bibliothèques : lesquelles choisir ? Lorsqu'on effectue une recherche avec les mots-clés 'scraping python' sur Google, on est généralement dirigé vers des tutoriels qui ne jurent que par Requests et BeautifulSoup. Cependant, nous avons décidé de ne pas les utiliser dans ce projet.

Pourquoi se priver d'outils aussi populaires et disposant d'un grand nombre de ressources sur le sujet ?"

La raison principale qui nous a poussé à renoncer à Requests et BeautifulSoup vient d'abord de leur nature synchrone. En effet, avec ces outils, chaque requête est traitée l'une après l'autre, ce qui peut considérablement ralentir notre processus de collecte de données. Pour illustrer cela, imaginons que nous ayons 3 000 articles à scraper, et que chaque requête prenne en moyenne 1 seconde : nous aurions besoin de près d'une heure pour récupérer toutes les données !Et je vous parle meme pas des astuces saugrenu qu'il faut echafauder pour limiter votre programme seulement quelques liens, ajouter a cela que les injonctions du fichier roboot.txt vous allez les implementer «from scratch». De plus, ces bibliothèques montrent rapidement leurs limites face aux sites web modernes :

- Incapacité à exécuter le JavaScript , ce qui est problématique car de nombreux sites, comme celui d'Auchan, chargent leur contenu dynamiquement ;
- Difficulté à gérer les popups de cookies et autres éléments interactifs
- Pas de support natif pour les éléments qui se chargent progressivement lors du défilement de la page

Nous ne sommes pas entrain de dire que Requests et BeautifulSoup sont des reliques du passe,no nous remarquons simplement que pour notre projet,ce ne sont pas forcement les meilleurs choix.

Bien sûr, il existe des solutions pour paralléliser les requêtes, comme le threading ou asyncio. Cependant, leur mise en œuvre peut rapidement devenir complexe. Le threading en Python a ses limites à cause du GIL (Global Interpreter Lock), et asyncio, bien que puissant, nécessite une bonne compréhension des concepts de programmation asynchrone (coroutines, event loops, etc.). Si ces notions ne vous sont pas familières, il vaut mieux éviter de s'aventurer dans cette direction pour l'instant. C'est pourquoi nous allons privilégier des solutions plus modernes qui intègrent nativement des mécanismes de gestion des requêtes parallèles, tout en restant simples

d'utilisation.Comme mentionne dans l'introduction,nous avons choisi de travailler avec scrapy

Scrapy est un framwork est spécialement conçu pour le web scraping et gère nativement les requêtes asynchrones. Plus besoin de se prendre la tête avec la programmation asynchrone,Scrapy s'occupe de tout ! Il suffit de définir nos "spiders" (nos robots collecteurs) et le framework optimise automatiquement les performances. Mais ce n'est pas tout ! Scrapy propose tout un tas d'outils pour le scraping dont nous ne parleront pas entierement ici