# Projet scraper pour SEO

*Christopher Alfred, Fatima Berrabah, Dorian Cormerais, Christ Mougani*

**Diginamic**

January 16, 2024

# Introduction

Dans le domaine du référencement (SEO), l'accès rapide et efficace à des informations précieuses sur les pages web est crucial. C'est dans cette optique que notre équipe a développé un ensemble d'outils de scraping et de gestion de données, regroupés sous le nom de "Projet_scraper_SEO".
La structure du projet comprend plusieurs composants essentiels. Tout d'abord, le fichier **logs.py** offre une gestion robuste des journaux, enregistrant les événements et les erreurs liés au scraping. Ces informations sont cruciales pour comprendre et résoudre les problèmes potentiels rencontrés lors du processus.
Le cœur du scraping est représenté par le fichier **distribue_scraper.py**, qui orchestre le traitement distribué des URLs en attente. Ce script utilise des processus distincts pour accélérer le scraping et assure une gestion adéquate des erreurs, garantissant la fiabilité du processus.

La logique de scraping est implémentée dans le fichier **programme_scraper.py**. Ce dernier extrait des informations significatives à partir des pages web, telles que le titre, les balises d'en-tête, les liens, et bien plus encore. Les résultats sont ensuite stockés dans une base de données *MongoDB*, assurant une traçabilité et une analyse ultérieure.
La conception de la base de données est méticuleusement planifiée, avec trois collections distinctes : '**logs**' pour les journaux, '**urls**' pour la gestion des URLs, et '**pages_metadata**' pour stocker les informations extraites des pages.
Enfin, pour une interaction conviviale, l'interface en ligne de commande, **interface_command_line.py**, permet aux utilisateurs d'ajouter manuellement des URLs à scraper dans la base de données, offrant ainsi une flexibilité d'utilisation.

Au cours de cette présentation, nous explorerons en détail chaque composant de notre projet, mettant en lumière leurs fonctionnalités et interactions pour répondre aux besoins spécifiques du référencement.

# logs.py

Le fichier **logs.py** est responsable de la gestion des journaux **(logs)** pour le projet
Cette partie du projet est dédiée à la gestion des événements et des erreurs, en les stockant dans une base de données MongoDB. La fonction **log_event** enregistre un événement avec un message associé, tandis que la fonction **log_error** enregistre une erreur avec une URL et un message d'erreur associés.

# distribue_scraper.py

Le fichier **distribue_scraper.py** implémente un scraper distribué qui traite les URLs de manière asynchrone en utilisant des processus distincts
Ce fichier définit une fonction **distributed_scraper** qui utilise la multiprocessing pour créer un processus distinct qui traitera les URLs en parallèle. Il gère les erreurs liées aux requêtes, introduit des tentatives de réessai en cas d'échec et utilise des journaux pour enregistrer les événements et les erreurs.
Le bloc **if __name__ == "__main__"**: montre un exemple d'utilisation où le scraper distribué est lancé en tant que processus séparé avec l'URL de base et un identifiant de processus comme arguments.

# programme_scraper.py

Le fichier **programme_scraper.py** est responsable de la logique principale de scraping, notamment l'extraction d'informations à partir d'une page web et leur stockage dans une base de données MongoDB.
Ce fichier contient des fonctions pour l'insertion d'URL, la récupération d'URL en attente, la marque d'URL comme complétée, et une fonction principale **simple_scrape** qui effectue le scraping d'une page web. Cette dernière fonction extrait différentes informations telles que le titre, les balises d'en-tête, les balises en gras, les balises italiques, etc., à partir de la page HTML.

# Conception

La structure de notre base de données est conçue pour stocker les logs, les URLs en attente, et les métadonnées des pages:

- **Collection 'logs':**
  - **_id:** Identifiant unique généré par MongoDB pour chaque document.
  - **type:** Type de log, pouvant être "event" pour les événements ou "error" pour les erreurs.
  - **timestamp:** Date et heure de l'événement ou de l'erreur.
  - **message:** Message associé à l'événement ou à l'erreur.

  Cette collection est utilisée pour enregistrer des informations sur les événements et les erreurs qui se produisent pendant l'exécution du programme.

# Conception

- **Collection 'urls':**
  - **_id:** Identifiant unique généré par MongoDB pour chaque document.
  - **url:** L'URL de la page web.
  - **scope:** Le contexte ou la portée associée à l'URL.
  - **status:** Statut de l'URL, pouvant être "pending" (en attente), "processing" (en cours de traitement), ou "completed" (traitée).

  Cette collection est utilisée pour gérer les URLs en attente de traitement, suivre leur statut, et éviter le traitement répété des mêmes URLs.

# Conception

- **Collection 'pages_metadata':**
  - **_id:** Identifiant unique généré par MongoDB pour chaque document.
  - **url:** L'URL de la page web.
  - **html:** Le contenu HTML de la page.
  - **scrapping_date:** Date et heure du scraping de la page.
  - **title:** Le titre extrait de la page.
  - **header_tags:** Les balises d'en-tête extraites de la page.
  - **bold_tags:** Les balises en gras extraites de la page.
  - **italic_tags:** Les balises italiques extraites de la page.

  Cette collection stocke les métadonnées des pages web après le scraping, ce qui permet de conserver des informations utiles pour l'analyse ultérieure.

# exemple_utilisation.py

Le fichier **exemple_utilisation.py** est un exemple d'utilisation de notre système de scraping.

- Insertion d'une URL en attente:
  - La fonction **insert_url** est utilisée pour insérer une nouvelle URL ('https://quotes.toscrape.com/page/2/') dans la collection 'urls' avec le statut initial 'pending'.

# exemple_utilisation.py

- Traitement des URLs en attente :
  - Une boucle infinie est utilisée pour continuellement vérifier et traiter les URLs en attente.
  - La fonction **get_pending_url** récupère une URL en attente depuis la base de données.
  - Si une URL est récupérée, la fonction **simple_scrape** est utilisée pour effectuer le scraping de la page associée.
  - Ensuite, la fonction **set_url_completed** est utilisée pour marquer l'URL comme traitée dans la collection 'urls'.
  - Si aucune URL en attente n'est trouvée, la boucle se termine.

Cet exemple montre comment utiliser nos fonctions pour ajouter des URLs en attente, les traiter en continu, et les marquer comme complétées après le scraping

# interface_command_line.py

Le fichier **interface_command_line.py** est une interface en ligne de commande permettant d'insérer des URLs dans votre base de données MongoDB.

- Parsing des arguments de la ligne de commande:
  - La fonction **parse_args** utilise le module argparse pour analyser les arguments de la ligne de commande. Trois arguments sont pris en charge : **–url**, **–scope**, et **–status** (optionnel, avec une valeur par défaut "pending").
- Insertion de l'URL dans la base de données:
  - La fonction **insert_url** insère une nouvelle URL dans la collection '**pending_urls_correct**' de la base de données avec les informations fournies.

# interface_command_line.py

- Connexion à la base de données et exécution du programme:
  - La connexion à la base de données MongoDB est établie.
  - Si les arguments **–url** et **–scope** sont fournis, l'URL est insérée dans la base de données.
  - Sinon, un message est affiché demandant de fournir au moins les arguments requis.

Cela permet à l'utilisateur d'insérer des URLs dans la base de données en utilisant la ligne de commande avec des options spécifiques.