JOBMARKET

Introduction aux API et à la pagination

Qu'est-ce qu'une API?

Une **API** (Application Programming Interface ou Interface de Programmation d'Application) est un ensemble de règles et de protocoles qui permettent à différentes applications de communiquer entre elles. Dans le contexte des offres d'emploi, les API comme celles d'Adzuna et France Travail permettent aux développeurs d'accéder aux bases de données d'offres d'emploi de manière structurée.

Pourquoi la pagination est-elle nécessaire?

Lorsqu'une API doit renvoyer un grand nombre de résultats (comme des milliers d'offres d'emploi), il n'est pas pratique de tout renvoyer en une seule fois pour plusieurs raisons :

- Charge réseau excessive
- Temps de réponse trop long
- Utilisation mémoire importante
- Risque de timeout

C'est pourquoi les API implémentent différentes stratégies de pagination pour découper les résultats en "pages" plus petites.

Méthodes courantes de pagination des API

1. **Pagination par numéro de page** (utilisée par Adzuna)
    - Paramètres courants : `page` (numéro de page) et `per_page` (nombre d'éléments par page)
    - Exemple : `/search?page=2&per_page=50`
    - Avantages : Simple à comprendre et à implémenter
    - Inconvénients : Moins efficace pour les datasets très grands
2. **Pagination par offset et limite** (variante utilisée par France Travail)
    - Paramètres courants : `offset` (décalage) et `limit` (nombre d'éléments à retourner)
    - France Travail utilise un format `range=p-d` où `p` est l'offset et `d` est l'offset+limit-1
    - Exemple : `/search?range=50-99` (équivaut à offset=50, limit=50)
    - Avantages : Flexible, permet de sauter directement à un point précis
    - Inconvénients : Peut devenir inefficace pour de grands offset
3. **Pagination par curseur**
    - Utilise un "pointeur" ou un ID vers le dernier élément récupéré
    - Exemple : `/search?cursor=abc123xyz`
    - Avantages : Très efficace pour les grands datasets, résistant aux modifications entre les appels
    - Inconvénients : Plus complexe à implémenter, souvent unidirectionnelle
4. **Pagination par timestamp**

    - Filtre les résultats par date/heure, en demandant les éléments après un certain timestamp
    - Exemple : `/search?created_after=2023-01-01T12:00:00Z`
    - Avantages : Excellent pour les flux chronologiques comme les offres d'emploi récentes
    - Inconvénients : Nécessite un champ de date bien indexé

5. **Pagination avec liens hypermedia** (HATEOAS)

    - L'API fournit des liens vers la page suivante/précédente dans sa réponse
    - Exemple : Liens dans les en-têtes HTTP ou dans le corps JSON
    - Avantages : Autodescriptif, facilite la navigation
    - Inconvénients : Requiert plus de traitement côté client

Notre script implémente les méthodes appropriées pour chaque API : pagination par numéro de page pour Adzuna et pagination par plage (offset-limite) pour France Travail.

Gestion de la pagination dans le script

Le script intègre un mécanisme qui :

1. Récupère d'abord une page de résultats (50 pour Adzuna, jusqu'à 150 pour France Travail)
2. Continue à récupérer les pages suivantes si nécessaire jusqu'à atteindre le nombre maximum de résultats demandé
3. Gère les erreurs spécifiques liées à la pagination (dépassement de limite, etc.)
4. Combine tous les résultats dans un ensemble unifié## Sécurité des données

Le script applique les mesures de sécurité suivantes pour garantir l'intégrité des données :

1. **Nettoyage des données textuelles** : Suppression des caractères problématiques (retours à la ligne, caractères de contrôle) et échappement des guillemets.

2. **Validation des valeurs numériques** : Conversion et validation des valeurs numériques (salaires, coordonnées géographiques).

3. **Protection contre l'injection CSV** : Les caractères spéciaux sont échappés pour éviter les injections dans les fichiers CSV.

4. **Gestion des valeurs manquantes** : Remplacement des valeurs nulles par des valeurs appropriées au contexte.

5. **Sanitization des noms de fichiers** : Nettoyage des noms de fichiers pour éviter les problèmes de sécurité.

6. **Chemins de fichiers sécurisés** : Conversion des chemins relatifs en chemins absolus et vérification de l'existence des dossiers.

7. **Repli sur des solutions alternatives** : En cas d'échec lors de la sauvegarde, utilisation de méthodes alternatives pour préserver les données.

8. **Protection des identifiants API** : Aucun identifiant d'API n'est stocké dans le code (utilisation de variables d'environnement).# Normalisateur de données d'emploi

Ce script permet de récupérer, normaliser et analyser des offres d'emploi provenant des API Adzuna et France Travail (anciennement Pôle Emploi).

L'utilisation de Pydantic pour la validation des données

Pourquoi Pydantic?

Pydantic est une bibliothèque Python qui permet la validation des données et la gestion des paramètres à l'aide d'annotations de type Python. Dans ce projet, nous utilisons Pydantic V2 pour plusieurs raisons essentielles:

1. **Validation robuste des données** - Pydantic vérifie automatiquement que les données reçues des API correspondent aux types attendus, ce qui nous permet de détecter immédiatement les incohérences.
2. **Conversion automatique des types** - Pydantic convertit intelligemment les chaînes de caractères en entiers, dates, URLs, etc., selon les annotations de type.
3. **Documentation intégrée** - Les modèles Pydantic sont autodocumentés grâce aux annotations de type et aux descriptions de champs.
4. **Sérialisation et désérialisation** - Conversion facile entre objets Python et formats comme JSON, nécessaire pour communiquer avec les API.
5. **Performance** - Pydantic V2 utilise Rust en arrière-plan, ce qui lui confère d'excellentes performances, essentielles pour traiter de grands volumes de données.

Comment nous utilisons Pydantic

Dans notre script, nous utilisons Pydantic à trois niveaux:

1. **Modèles de requête** - Pour structurer et valider les paramètres envoyés aux API:

```
class SearchParams(FranceTravailModel):
    motsCles: Optional[str] = Field(default=None, description="Mots
clés de recherche")
    range: Optional[str] = Field(default="0-49",
description="Pagination des données")
```

2. **Modèles de réponse** - Pour valider et structurer les données reçues des API:

```
class Offre(FranceTravailModel):
    id: str = Field(description="Identifiant de l'offre d'emploi")
    intitule: str = Field(description="Intitulé de l'offre")
```

3. **Modèle normalisé** - Pour uniformiser les données des différentes sources:

```
class NormalizedJobOffer(BaseModel):
    id: str
    source: str  # 'adzuna' ou 'france_travail'
    title: str
```

Cette approche nous permet de:

- Détecter rapidement les changements dans la structure des API
- Assurer l'intégrité des données tout au long du processus
- Gérer proprement les valeurs manquantes ou mal formatées
- Fournir une base de code maintenable et évolutive
- Documenter clairement la structure des données

Pour les développeurs qui souhaitent étendre ce script, la compréhension des modèles Pydantic est essentielle car ils définissent la structure et les contraintes de tous les échanges de données.