

Requêter des données & construire un mini-ETL

Amina MARIE

Solutions d'échange inter-applicatif



Ce module présente différentes méthodes d'échange de données entre systèmes (interne ou publics). Il se focalise particulièrement sur l'utilisation d'APIs en présentant leur fonctionnement et les méthodes courantes pour communiquer avec elles (récupérer ou envoyer des données).

- Partie 1 – Exposer ses données via une API REST sécurisée
- Partie 2 – Requêter des données et construire un mini ETL
- Partie 3 – Utiliser des API Push
- Partie 4 – Publier des données avec un POST

Projet fil rouge – API de gestion de personnages de manga

Objectif global : Construire une API complète de gestion de personnages fictifs (type Olive et Tom) avec des fonctionnalités :

- Exposition sécurisée (GET /personnages)
- Scripts de requêtage
- Simulation d'événements (push)
- Envoi de données à l'API (POST /scores)
- Chaque partie du cours alimente et améliore ce projet en y ajoutant une brique fonctionnelle.

Plan de la 2ème partie



Objectifs pédagogiques

- Comprendre les flux de données inter-applicatifs
- Expliquer les concepts d'ETL et ELT
- Identifier les outils ETL du marché
- Requêter une API en Python et JavaScript
- Gérer les statuts, erreurs, pagination, formats de réponse
- Nettoyer, transformer et sauvegarder les données extraites
- Mettre en place de bonnes pratiques de sécurité
- Introduire des tests simples dans des scripts



Flux de données inter- applicatifs

Définition

Un **flux de données inter-applicatif** permet à des systèmes différents d'échanger automatiquement des informations sans intervention humaine, souvent via des **APIs REST** ou des connecteurs.

Cas métier	Données échangées	Exemple
Recrutement	CV, infos candidats	API Welcome to the Jungle → outil RH
E-commerce	Stock, prix	API fournisseur → site marchand
Finances	Transactions	API Stripe → tableau de bord interne



Concepts d'ETL / ELT

La différence

ETL : Extract – Transform – Load

- **Extract** : extraire les données (API, base de données, fichier...)
- **Transform** : filtrer, renommer, nettoyer, enrichir
- **Load** : charger dans un autre système (base, data lake, fichier)

ELT : Extract – Load – Transform

- Variante plus récente : la transformation se fait **après** le chargement, dans le système cible (BigQuery, Snowflake...)

La différence

	ETL	ELT
Transformation	Avant le chargement	Après le chargement
Stockage cible	Base SQL classique, fichiers	Data warehouse moderne (ex: BigQuery, Snowflake)
Volumétrie	Moyenne	Élevée
Outils	Python, Talend, Make	BigQuery, dbt, SQL
Contrôle métier	Très fort	Plus délégué à la plateforme

Exemple 1 – Envoi quotidien de rapports RH

Contexte :

- Une API RH expose les candidatures reçues
- On filtre les profils intéressants
- On génère un fichier Excel envoyé au manager

Pourquoi ETL ?

- On a besoin de *filtrer et structurer* avant l'envoi
- Les données sont petites mais doivent être *préparées proprement*

Techno : Python + requests + pandas → fichier .xlsx

Exemple 2 – Nettoyage d'un fichier client avant intégration

Contexte :

- Un CSV d'un partenaire contient des champs mal formatés (emails, codes postaux)
- Il faut les corriger avant import dans la base

Pourquoi ETL ?

- Le fichier est local et doit être *corrigé avant* import
- Le système cible n'acceptera pas les erreurs

Techno : Python ou Make → transformation → base SQL

Exemple 3 – Analyse de logs d'une banque

- Contexte :
- Des millions de lignes de logs bruts sont reçues chaque jour
- Elles sont chargées en l'état dans BigQuery, puis analysées en SQL

Pourquoi ELT ?

- *Gros volume* de données
- Le data warehouse peut stocker et transformer très efficacement

Techno : BigQuery, SQL, dbt

Exemple 4 – Données e-commerce pour reporting

Contexte :

- Les commandes, clics, paniers sont extraits via Fivetran
- Les données sont stockées dans Snowflake
- La transformation (calculs, regroupement) est faite par des analystes via SQL

Pourquoi ELT ?

- Les transformations sont *complexes et évolutives*
- Les analystes veulent *contrôler le SQL* eux-mêmes

Résumé

Situation	Choix conseillé
Données sensibles ou instables	ETL (on contrôle la qualité avant)
Fort besoin métier sur la logique	ETL (transformation = logique)
Gros volume, besoin de performance	ELT (charge brut, transforme après)
Analyses flexibles par les métiers	ELT (transformation = SQL métier)
Petits flux simples	ETL rapide (Python, Make, Zapier)








Panorama des outils ETL du marché

Les outils du marché

Outil	Type	Avantages	Inconvénients
Talend	Desktop, open source	Très complet	Complexe
Fivetran	SaaS	Plug & play, maintenance automatique	Payant
Airbyte	Open source	Moderne, connecteurs variés	Installation à maîtriser
Segment	SaaS	Excellent pour analytics	Moins flexible
Make / Zapier	No-code	Très simple	Peu adapté aux gros volumes
Ascend.io	Cloud ETL	Programmation automatique des flux	Moins répandu

Cas d'utilisation

Critère	Outils recommandés
 Débutant	Make, Zapier, Python simple
 Niveau développeur	Python + Pandas, n8n, Airbyte
 Entreprise / gros volume	Fivetran, Airbyte + dbt, Talend
 Budget très limité	Python, Airbyte (self-hosted), Make (free)
 Données analytiques	dbt + BigQuery / Snowflake

- Tu veux automatiser sans coder → **Make / Zapier / n8n**
- Tu veux connecter Stripe, Shopify, GA, Hubspot... → **Fivetran / Airbyte**
- Tu veux écrire des règles complexes, transformer et filtrer → **Python / Talend**
- Tu veux faire des analyses dans **BigQuery / Snowflake** → **ELT avec dbt**



Requêtage d'API – HTTP et Pagination

Librairies utilisées

Langage	Librairie
Python	requests
JavaScript	fetch() intégré, ou axios

```
import requests # On importe la librairie permettant de faire des requêtes HTTP

# On envoie une requête GET à l'API de Coindesk pour obtenir le prix du Bitcoin
response = requests.get("https://api.coindesk.com/v1/bpi/currentprice.json")

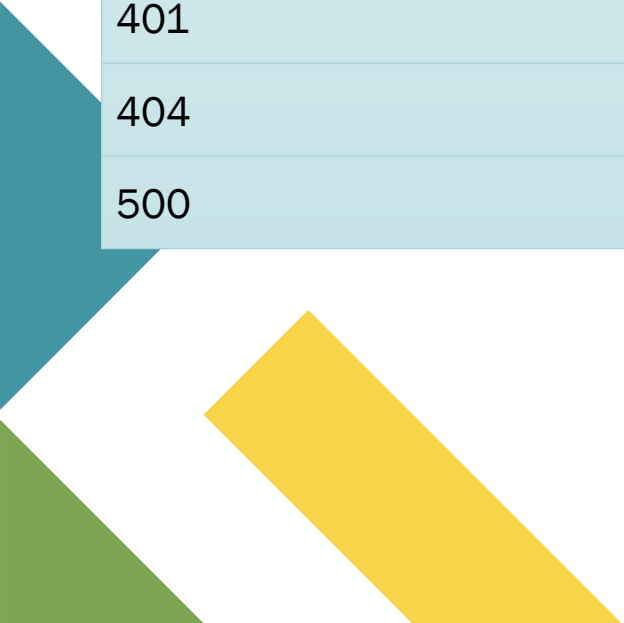
# Si tout s'est bien passé (statut 200)
if response.status_code == 200:
    data = response.json() # On transforme la réponse JSON en dictionnaire Python
    print(data["bpi"]["EUR"]["rate"]) # On affiche le prix en euros
else:
    print("Erreur :", response.status_code) # Sinon, on affiche le code d'erreur
```

```
// On envoie une requête GET à l'API agify.io pour estimer l'âge d'un prénom
fetch("https://api.agify.io?name=amine")
  .then(res => res.ok ? res.json() : Promise.reject("Erreur API")) // Si tout va bien, on transfère
  .then(data => console.log(data.age)) // On affiche l'âge estimé
  .catch(err => console.error(err)); // On affiche les erreurs si la requête échoue
```

Statuts HTTP



Code	Signification
200	OK
401	Non autorisé (token manquant)
404	Ressource introuvable
500	Erreur serveur



Pagination

- Certaines APIs renvoient les données par page

GET /clients?page=1

GET /clients?page=2

Exemple de boucle avec Python:

```
page = 1
tous = []

while True:
    r = requests.get(f"https://api.exemple.com/users?page={page}")
    data = r.json()
    if not data["users"]:
        break
    tous += data["users"]
    page += 1
```



Transformation des données

Nettoyage

```
nom = "Léo 🚧"  
nom_propre = nom.replace("🚧", "")
```

Corriger ou supprimer les **erreurs, doublons ou incohérences** qui empêchent une analyse fiable.

Exemples de problèmes :

- Noms mal encodés : "Renée" devient "RenÃ©e"
- Données manquantes : email vide, date absente
- Données absurdes : un âge de 400 ans, un code postal à 3 chiffres
- Doublons dans une liste de clients

Pourquoi c'est indispensable :

- Les **outils** de BI ou d'analyse **plantent ou donnent de faux résultats** si les données sont sales
- Ça évite de **tirer de mauvaises conclusions**
- Ça garantit une **base saine** pour les étapes suivantes

Filtrage

```
data = [{"nom": "Tom", "score": 90}, {"nom": "Ben", "score": 60}]  
filtrés = [d for d in data if d["score"] >= 80]
```

Conserver uniquement les **données utiles** selon des critères métier.

Exemples :

- Ne garder que les ventes de l'année en cours
- Exclure les utilisateurs inactifs depuis plus de 12 mois
- Filtrer les articles avec un stock > 0

Pourquoi c'est utile :

- On **réduit** la taille des jeux de données
- On **concentre** l'analyse sur ce qui est pertinent
- On améliore la **performance** des traitements suivants

Enrichissement

```
for perso in filtrés:  
    perso["niveau"] = "expert" if perso["score"] > 85 else "moyen"
```

Ajouter des **informations calculées ou croisées** pour rendre les données plus complètes ou parlantes.

Exemples :

- Calculer un **niveau de fidélité** client à partir du nombre d'achats
- Ajouter une **catégorie de prix** à un produit selon son montant
- Croiser avec une autre base pour ajouter des données géographiques

Pourquoi c'est utile :

- On **ajoute de la valeur métier** à des données brutes
- On peut créer des **segments**, des **KPIs**, des **indicateurs**
- On prépare les données pour des modèles d'IA ou du reporting

Agrégation

```
scores = [d["score"] for d in filtrés]
moyenne = sum(scores) / len(scores)
```

Résumer les données par regroupements (par mois, par client, par pays...) pour mieux les analyser.

Exemples :

- Moyenne de paniers par client
- Nombre de commandes par jour
- CA total par catégorie de produit

Pourquoi c'est utile :

- On **pass**e d'un niveau détaillé à un niveau stratégique
- On **facilite** la visualisation dans des tableaux de bord
- On identifie **des tendances**, des pics, des comportements

Résumé

Étape	Pourquoi on la fait	À quel moment
Nettoyage	Corriger / fiabiliser	Tout début
Filtrage	Se concentrer sur l'essentiel	Après le nettoyage
Enrichissement	Ajouter de la valeur métier	Avant la charge ou le modèle
Agrégation	Préparer pour la visualisation ou l'analyse	En sortie (Load ou BI)



Sécurité API – Bonnes pratique

Mauvais pratique

```
headers = {"token": "cle_dans_le_code"}
```

1. Exposition directe de secrets dans le code

- En mettant la clé en dur dans le code, n'importe qui ayant accès au script voit la clé API.
- Si le fichier est partagé (sur GitHub, un cloud, une clé USB), la sécurité est compromise.
- C'est l'équivalent de coller ton mot de passe écrit sur ton bureau.

2. Aucune possibilité de mise à jour facile

- Si la clé change (ce qui est recommandé régulièrement), il faut modifier le code à la main, puis redéployer. C'est peu maintenable.

Mauvais pratique

```
headers = {"token": "cle_dans_le_code"}
```

3. Impossible de gérer plusieurs environnements

- En production, tu ne dois **pas utiliser la même clé** qu'en test ou développement. Si la clé est dans le code, tu es bloqué avec une **clé unique**.

4. Danger si le code est versionné (Git)

- De nombreux développeurs ont **accidentellement publié leurs clés sur GitHub public**.
- Il existe des **robots** qui surveillent GitHub à la recherche de clés valides pour les exploiter instantanément.

Bonnes pratiques

1. Créer un fichier .env

API_TOKEN=xyz123

2. Utiliser un dotenv

```
from dotenv import load_dotenv # Librairie pour charger Les variables d'environnement
import os # Pour interagir avec Les variables d'environnement

load_dotenv() # Charge Les variables du fichier .env
token = os.getenv("API_TOKEN") # Récupère La variable appelée API_TOKEN
```

3. Ne pas versionner .env (ajouter à .gitignore)

Bonnes pratiques

- ✓ La clé n'est **jamais visible** dans le code
- ✓ On peut avoir une clé différente **par environnement**
- ✓ On ne versionne **pas .env** (il est dans .gitignore)

Gestion des erreurs avancée

Timeout

```
requests.get(url, timeout=3)
```

Pourquoi c'est dangereux de ne pas fixer de timeout

- Ton application peut **se figer** pendant plusieurs minutes.
- Si tu appelles cette API dans une boucle ou dans un script automatisé, tu risques de **bloquer toute la chaîne**.
- Tu **ne peux pas gérer proprement les erreurs** si tu attends indéfiniment.

Gère le avec un try/except pour lever un Timeout

Gestion des erreurs avancée

Timeout

Exemple concret de cas où c'est utile

Tu as un script qui récupère les ventes toutes les heures depuis une API partenaire.

Si l'API est lente ou en panne, tu ne veux pas que ton script reste bloqué pendant 10 minutes.

Le timeout permet de **reprendre la main rapidement** et d'adapter ton traitement (par exemple, réessayer plus tard, envoyer une alerte...).

Gestion des erreurs avancée

Retry Pattern

Le **retry pattern** (ou « **stratégie de réessai** ») consiste à **répéter automatiquement une requête** qui a échoué, en espérant qu'elle réussisse au bout de quelques tentatives.

Pourquoi c'est utile ?

Certaines erreurs sont **temporaires** :

- Le serveur met trop de temps à répondre (timeout),
- Le réseau a brièvement coupé,
- Le serveur était en surcharge.
- Plutôt que d'abandonner au **premier échec**, on peut réessayer **1 à 3 fois** avant de déclarer l'échec définitif.

Gestion des erreurs avancée

Retry Pattern

```
import requests
import time

url = "https://api.exemple.com"
max_retries = 3 # Nombre de tentatives
delay = 2       # Délai entre chaque tentative (secondes)

for i in range(max_retries):
    try:
        response = requests.get(url, timeout=5)
        if response.status_code == 200:
            print("Succès :", response.json())
            break # On sort de la boucle si ça fonctionne
        else:
            print(f"Erreur HTTP {response.status_code}")
    except requests.exceptions.Timeout:
        print(f"Tentative {i+1} échouée (timeout), nouvelle tentative dans {delay}s...")
        time.sleep(delay)
else:
    print("Toutes les tentatives ont échoué.")
```

Gestion des erreurs avancée

Retry Pattern

Pourquoi on met un délai entre les tentatives ?

Sans délai, on ferait 3 appels très rapides à la suite, ce qui pourrait :

- Surcharger encore plus un serveur lent
- Ne pas laisser le temps à l'API de se "remettre"
- C'est comme frapper à une porte, attendre 2 secondes, puis réessayer si personne n'ouvre.

On peut aussi gérer les erreurs `requests.exceptions.ConnectionError`

- On peut **augmenter** le délai à chaque tentative → c'est le **backoff exponentiel**

Gestion des erreurs avancée

Retry Pattern

Situation	Faut-il réessayer ?
Timeout ponctuel	✓ Oui
Erreur réseau (DNS, WiFi...)	✓ Oui
Erreur 500 (serveur)	✓ Parfois
Erreur 404 (ressource introuvable)	✗ Non, ça ne changera rien
Erreur 401 (non autorisé)	✗ Non, c'est un problème d'authentification

Format de réponse API

Format	Exemple	Traitement
JSON	{"nom": "Tom"}	.json()
XML	<nom>Tom</nom>	xml.etree.ElementTree
CSV	nom,score\nTom,90	csv.DictReader

```
data = response.json()
```

```
import xml.etree.ElementTree as ET  
root = ET.fromstring(response.text)
```

```
import csv  
from io import StringIO  
  
f = StringIO(response.text)  
reader = csv.DictReader(f)
```



Tests simples pour valider un ETL

Pourquoi tester un ETL



Un ETL (ou un script de collecte/transformation de données) peut échouer **sans lever d'erreur apparente**. Tester permet de s'assurer que :

- les données **ont bien été extraites**,
- les transformations **ont bien eu lieu**,
- les fichiers produits **contiennent les bonnes valeurs**,
- les erreurs **sont gérées proprement**.

Tester sans framework – version simple

```
def test_etl():  
    data = extraire_donnees() # ← appel à ta fonction d'extraction  
  
    assert isinstance(data, list), "Résultat non valide : ce n'est pas une liste"  
  
    assert len(data) > 0, "Liste vide : aucune donnée récupérée"  
  
    assert "nom" in data[0], "Clé 'nom' absente des données"
```

- assert vérifie une condition.
- S'il y a un problème, Python renvoie une AssertionError.
- Tu peux appeler cette fonction **manuellement** en développement.

Version automatisée avec pytest

```
from etl import extraire_donnees

def test_structure():
    data = extraire_donnees()
    assert isinstance(data, list)

def test_non_vide():
    data = extraire_donnees()
    assert len(data) > 0

def test_cle_nom():
    data = extraire_donnees()
    assert "nom" in data[0]
```

pip install pytest

projet/

|— etl.py ← ton script principal

|— test_etl.py ← fichier de tests

Lancer les tests: pytest



Exercices

Exercice 1 – Script JS qui requête l'API personnages

Objectif

Créer un **script simple en JavaScript** qui envoie une requête GET à ton API /personnages (créée dans la Partie 1), en passant un **token d'authentification dans les headers**, puis affiche la réponse.

Exercice 1 – Script JS qui requête l'API personnages

Étapes guidées :

1. **Créer un fichier HTML de test**
Ex. index.html, dans lequel tu intègres ton script JS.
2. **Utiliser fetch pour appeler l'API**
Appelle l'URL `http://localhost:8000/personnages` (ou celle que tu as utilisée dans FastAPI).
3. **Ajouter le header token dans la requête**
Tu dois écrire `headers: { token: "ta_clé" }` dans l'objet fetch.
4. **Vérifier le status de la réponse**
Ne traite la réponse que si `response.ok` ou `response.status === 200`.
5. **Afficher le résultat ou une erreur**
Affiche les données dans la console (ou dans la page HTML), ou un message d'erreur.

Exercice 1 – Script JS qui requête l'API personnages

Points de vigilance :

- Le nom du header est "token" (identique à celui défini dans ton backend)
- Attention aux **CORS** : si tu appelles depuis une page web, ton API doit autoriser cette origine (voir Partie 1 : CORS)

Bonus :

- Ajouter un champ de formulaire pour taper un prénom → appeler /personnages?prenom=...
- Afficher les résultats sous forme de liste HTML (...)

Exercice 2 – Script Python pour API paginée

Objectif

Créer un **script Python** qui appelle une **API paginée publique**, récupère **toutes les pages**, **filtre les données**, puis les enregistre dans un **fichier JSON**.

API suggérée pour débiter :

<https://projects.propublica.org/nonprofits/api/v2/search.json?q=chat&page=0>

Cette API renvoie une liste d'associations américaines contenant le mot "chat" dans leur description. Elle est paginée par le paramètre page.

Exercice 2 – Script Python pour API paginée

Étapes guidées :

1. Choisir une API paginée : L'URL doit permettre de changer de page avec `page=0`, `page=1`, etc.
2. Écrire une boucle `while` ou `for` : Appelle chaque page jusqu'à ce qu'il n'y ait plus de résultats.
3. Stocker tous les résultats dans une liste Python : Ex : `toutes_donnees = []`
4. Filtrer les données : Par exemple, ne garder que celles avec un `city` défini, ou un `revenu > 0`.
5. Écrire la liste dans un fichier JSON : Utilise `json.dump(...)` et `open("fichier.json", "w")`

Exercice 2 – Script Python pour API paginée

Points de vigilance :

- Bien vérifier que tu **ne récupères pas indéfiniment** les mêmes pages (boucle infinie)
- Gérer les erreurs (par exemple, si `status_code != 200`)
- Penser à `time.sleep()` si l'API limite la fréquence

Bonus :

- Créer une fonction `extract()`, une `transform()`, une `load()` pour séparer les étapes ETL
- Ajouter un `try/except` + `timeout` + `retry` pattern
- Calculer la moyenne d'un champ numérique (ex : revenus)

Exercice 3 – Envoyer des données transformées à une API (POST)

Objectif :

À partir des données extraites et filtrées dans l'exercice 2, les **transformer** légèrement (ajout d'un champ calculé, renommage, etc.), puis les **envoyer** à ton API FastAPI via un **endpoint POST /feedback ou /scores**.

Pré-requis :

- Ton API FastAPI doit déjà inclure un endpoint POST /scores (à créer si besoin).
- Le script de l'exercice 2 doit avoir sauvegardé un fichier .json avec des données.

Exercice 3 – Envoyer des données transformées à une API (POST)

Étapes guidées :

1. Lire les données sauvegardées dans le fichier JSON

```
import json with open("fichier.json", "r") as f: data = json.load(f)
```

2. Transformer les données

1. Ajouter un champ avis (ex. : "positif" si un critère est respecté)
2. Supprimer des clés inutiles
3. Renommer certains champs

Exercice 3 – Envoyer des données transformées à une API (POST)

3. Envoyer les objets un par un à l'API avec `requests.post()`

1. Passer les données dans `json=...`

2. Ajouter le header avec le token de sécurité

4. Gérer les erreurs et afficher les retours

1. Vérifier le `status_code`

2. Afficher une réponse claire (success, échec, déjà existant...)

Exercice 3 – Envoyer des données transformées à une API (POST)

Points de vigilance :

- ⚠ Vérifier que le Content-Type est bien application/json
- ⚠ Gérer les erreurs du serveur (422, 400, 401, etc.)
- ⚠ Ne pas envoyer 100 objets trop vite → `time.sleep(0.5)` conseillé

Bonus (si rapide) :

- Ajouter une barre de progression (tqdm)
- Sauvegarder un log des réussites/échecs dans un fichier .txt
- Ajouter un test unitaire pour valider que le champ "avis" est bien ajouté

Merci

