Chapitre 2:

Langages, outils et pratiques incontournables

4DATA

Benjamin Jurczak

SUPINFO

Sommaire

1. Python, un langage de choix
2. Bases de données et SQL
3. Utilisation d’APIs
4. Bonnes pratiques générales

1. Python, un langage de choix

1. Python, un langage de choix

Pourquoi Python ?

LE **langage le plus populaire** pour la gestion des pipelines de données:

- **Simplicité et lisibilité** : Syntaxe claire, idéal pour automatiser des tâches.
- **Riche écosystème** : De nombreuses bibliothèques adaptées à l'ETL et ELT (`pandas`, `Numpy`, `SQLAlchemy`, `pySpark`,).
- **Intégration facile avec des bases de données et des APIs**.
- **Compatibilité avec le cloud** : AWS, Google Cloud, Azure.

1. Python, un langage de choix

Installation et environnement virtuel

- **Installer Python** :

  Téléchargez Python depuis python.org.

  Utilisez pyenv pour gérer plusieurs versions de Python sur la même machine.

- **Gestion des dépendances** :

  Utilisez pip pour installer les bibliothèques nécessaires.

  Créez un environnement virtuel pour isoler les dépendances de votre projet :

```
python -m venv mon_env
source mon_env/bin/activate  # Sur Linux/Mac
mon_env\Scripts\activate     # Sur Windows
```

Librairie *pandas*

bibliothèque **incontournable** pour :

- Lire et écrire différents formats de fichiers (CSV, JSON, Excel, Parquet).
- Manipuler des tableaux de données avec **DataFrames**.
- Transformer, nettoyer et analyser des datasets.
- S'intégrer facilement avec des bases de données et APIs.

```
pip install pandas
import pandas as pd
```

Librairie *pandas*

- Pour tester, rdv sur Github: https://github.com/jurczakB/4DATA

Librairie *Numpy*

bibliothèque **incontournable** pour :

- Lire et écrire différents formats de fichiers (CSV, JSON, Excel, Parquet).
- Manipuler des tableaux de données avec **DataFrames**.
- Transformer, nettoyer et analyser des datasets.
- S'intégrer facilement avec des bases de données et APIs.

Librairie *SQLAlchemy*

SQLAlchemy - Interaction avec les bases de données

Permet d'interagir avec des bases de données relationnelles (SQLite, PostgreSQL, MySQL, etc.).

Facilite l'extraction et le chargement de données dans les pipelines ETL/ELT.

Cas d'utilisation :

Lire des données depuis une base de données dans un DataFrame.

Écrire des données transformées dans une nouvelle table.

```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///ma_base.db')
df = pd.read_sql('SELECT * FROM ma_table', engine)
df.to_sql('nouvelle_table', engine, if_exists='replace')
```

Librairie *pySpark*

Qu'est-ce que PySpark ?

Interface Python pour Apache Spark, un moteur open-source utilisé pour le **traitement distribué de grandes quantités de données**.

Traitement rapide: Big Data (exécutions parallèles sur plusieurs machines).
Manipulation des données: mode batch ou en streaming.
Intégration avec SQL, Machine Learning et Graph Processing.

Utilisé dans les pipelines de données pour transformer et analyser de gros volumes de données.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("MonAppSpark").getOrCreate()
df = spark.read.csv('donnees.csv', header=True, inferSchema=True)
df_filtre = df.filter(df['age'] > 30)
```

Bonnes pratiques

Liste non exhaustive

Modularité : Diviser le code en fonctions et classes réutilisables.

Gestion des erreurs :

Utiliser des blocs try-except pour capturer les exceptions.

Logger les erreurs pour faciliter le débogage.

Tests unitaires : Tester ses fonctions avec unittest ou pytest.

Documentation : Utiliser des docstrings pour expliquer le fonctionnement du code.

-

1. Python, un langage de choix

2. Bases de données et SQL

2. Bases de données

Les bases de données, cœur des pipelines de données

- **Pourquoi les bases de données sont centrales ?**
  - Stockage structuré et persistant des données.
  - Point de passage obligatoire pour l'ETL et l'ELT.
  - Permettent la scalabilité, la sécurité et l'accès concurrentiel aux données.

- **Plusieurs rôles :**
  - Source de données (extraction).
  - Destination finale (entrepôt de données).
  - Environnement de transformation (ELT).

2. Bases de données

Le rôle central des bases dans les pipelines

- **Persistance des données :**

  Les bases stockent les données brutes et transformées.

- **Transformation ELT :**

  Exécuter des transformations directement dans l'entrepôt (ex : dbt + Snowflake).

- **Serving des données :**

  Fournir des données aux applications, dashboards, ou modèles ML.

- **Scalabilité :**

  Gérer des TB de données grâce au partitionnement et à l'indexation.

2. Bases de données

Les types de bases de données – Choisir le bon outil

- **Bases relationnelle**:
  - *PostgreSQL, MySQL* : Transactions rapides, intégrité des données (ACID).
  - Cas d'usage : Applications transactionnelles (e-commerce, CRM).
- **Entrepôts de données**:
  - *BigQuery, Snowflake, Redshift* : Optimisés pour l'analyse de gros volumes.
  - Cas d'usage : Reporting, Business Intelligence.
- **NoSQL** :
  - *MongoDB (document), Cassandra (colonnes)* : Flexibilité, scalabilité horizontale.
  - Cas d'usage : Données non structurées, IoT, temps réel.

2. Bases de données

PostgreSQL – La base relationnelle polyvalente

- **Pourquoi PostgreSQL ?**
  - Open source, extensible (supporte JSON, géospatial, etc.).
  - Conforme ACID, idéale pour les transactions complexes.
- **Cas d'usage dans les pipelines :**
  - Source de données pour l'ETL.
  - Base de staging pour les transformations ELT.
  - Exemple : CREATE TABLE ..., COPY pour l'import de données.

2. Bases de données

Entrepôts de données – BigQuery, Snowflake, Redshift

- **Caractéristiques clés :**
  - Stockage colonnaire pour les requêtes analytiques rapides.
  - Scalabilité cloud native (coût basé sur l'usage).
  - Intégration native avec les outils BI (Tableau, Power BI).
- **Cas d'usage :**
  - Centraliser des données provenant de multiples sources.
  - Exécuter des transformations SQL à grande échelle (ELT).

2. Bases de données

Bases NoSQL – Flexibilité pour les données non structurées

- **Quand utiliser NoSQL ?**
  - Données JSON/XML, logs, capteurs IoT.
  - Besoin de scalabilité horizontale (ex : Cassandra).
- **Exemple avec MongoDB :**
  - Stockage de documents JSON flexibles.
  - Requêtes simples pour l’ingestion de données semi-structurées.

2. Bases de données

Stockage cloud vs On-Premise – Avantages et inconvénients

- **Cloud (BigQuery, AWS RDS, Azure SQL) :**
  - Avantages : Scalabilité, maintenance gérée, coût à l'usage.
  - Inconvénients : Dépendance au fournisseur, coût à long terme.
- **On-Premise (PostgreSQL, MySQL) :**
  - Avantages : Contrôle total, sécurité physique.
  - Inconvénients : Coûts initiaux élevés, maintenance complexe.

2. Bases de données

Bonnes pratiques – Choisir la bonne base

- **Critères de choix :**
  - Volume et vélocité des données.
  - Structure (structurées vs semi-structurées).
  - Budget (open source vs solutions cloud).
- **Exemples :**
  - Données transactionnelles ➔ PostgreSQL.
  - Analytics à grande échelle ➔ BigQuery.
  - Données temps réel non structurées ➔ MongoDB.

2. Bases de données

Optimisation des performances

- **Indexation :**
  - Créer des index sur les colonnes fréquemment interrogées (ex : CREATE INDEX ...).
- **Partitionnement :**
  - Découper les tables par date ou région pour accélérer les requêtes.
- **Caching :**
  - Utiliser des vues matérialisées pour les résultats fréquents (ex : PostgreSQL, BigQuery).

2. Bases de données

SQL, requêtes sur les données

Qu'est-ce que SQL ?

SQL (**Structured Query Language**) est un langage utilisé pour :

- **Interroger** et **manipuler** des bases de données relationnelles.
- **Créer** et **modifier** des structures de bases de données.
- **Gérer** les permissions et transactions.

2. Bases de données

SQL, requêtes sur les données

cheatsheet de qualité:

https://media.datacamp.com/legacy/image/upload/v1714149594/Marketing/Blog/SQL_for_Data_Science.pdf

2. Bases de données et SQL

3. Utilisation d’API

3. Utilisation d’API

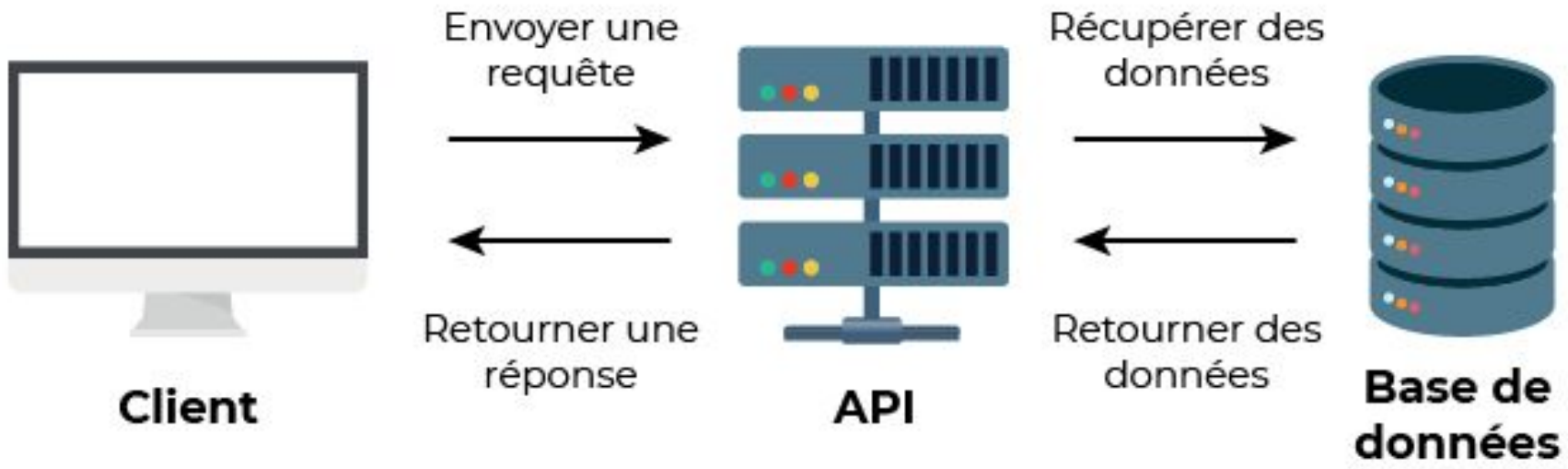
Les API - Interface de communication entre systèmes

Une API (Application Programming Interface) est un ensemble de règles qui permet à des logiciels de communiquer entre eux.

- **Utilité dans les pipelines de données :**
  - Ingestion de données en temps réel (météo, transactions, réseaux sociaux, etc.).
  - Intégration avec des services externes (Google Maps, Stripe, etc.).
  - Interconnexion de microservices dans une architecture moderne.
- **Types d'API :**
  - **REST** : Basé sur HTTP, simple et largement utilisé.
  - **GraphQL** : Permet des requêtes flexibles et précises.
  - **SOAP** : Plus ancien, utilisé dans certains systèmes d'entreprise.

3. Utilisation d’API

Les API - Interface de communication entre systèmes



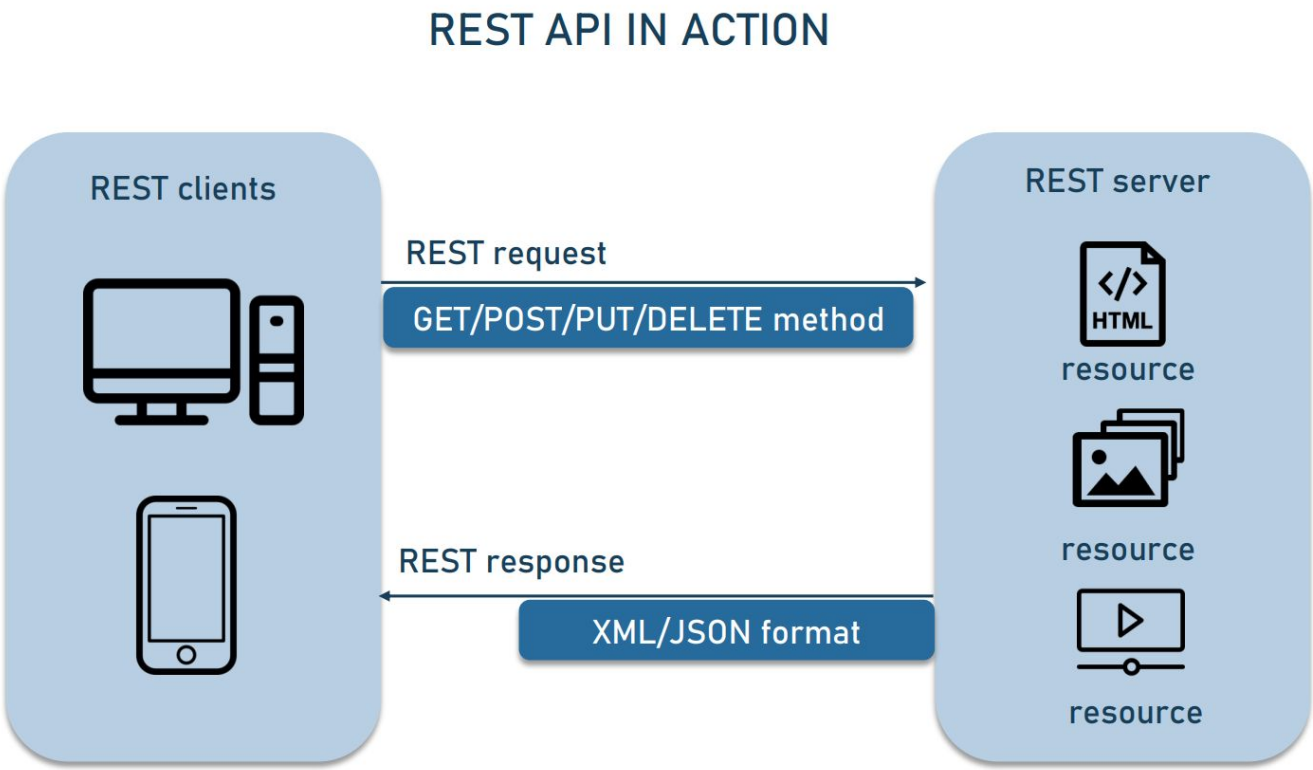
3. Utilisation d’API

Les requêtes HTTP - Fondamentaux des API REST

- **Méthodes HTTP :**
  - **GET** : Récupérer des données.
  - **POST** : Envoyer des données pour création.
  - **PUT/PATCH** : Mettre à jour des données.
  - **DELETE** : Supprimer des données.
- **Structure d'une URL d'API :**
  - Exemple : https://api.exemple.com/data?param1=value1&param2=value2
- **Réponses HTTP :**
  - Codes de statut (200 OK, 404 Not Found, 500 Server Error, etc.).
  - Format des réponses (JSON, XML, etc.).

3. Utilisation d'API

Les API - Interface de communication entre systèmes



altexsoft

3. Utilisation d’API

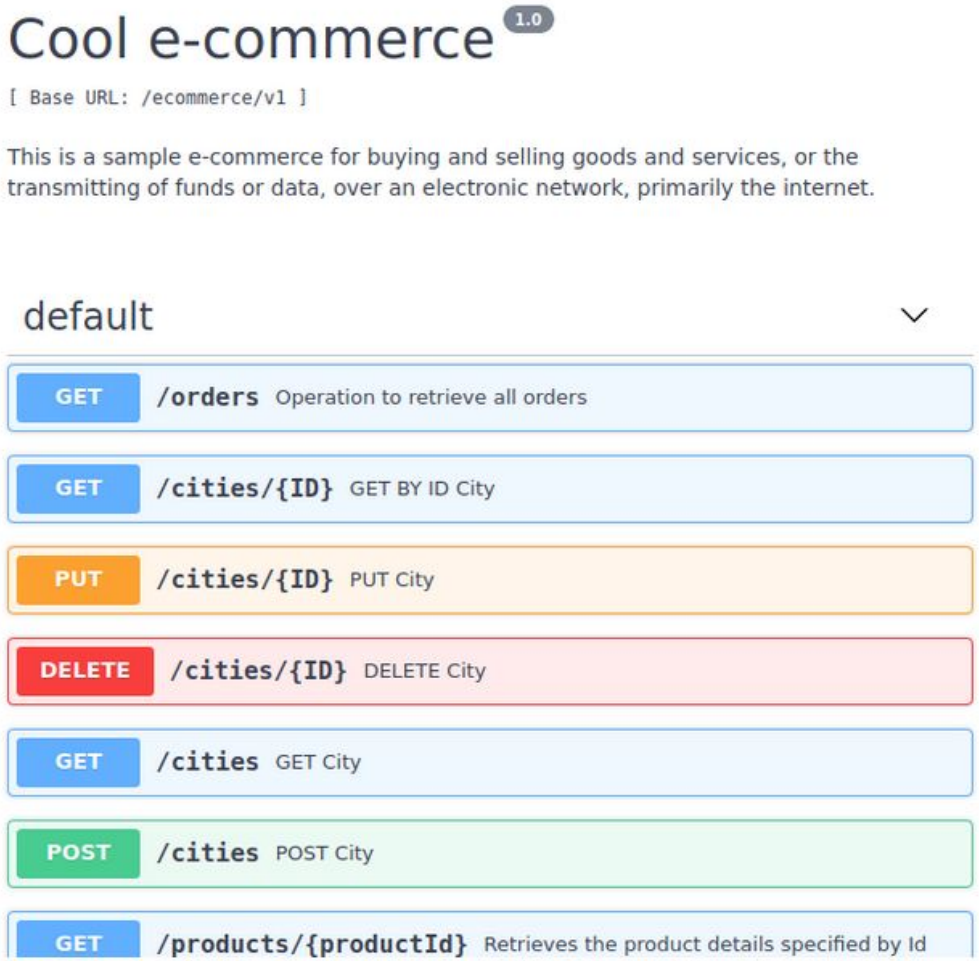
Swagger/OpenAPI - Documentation et test des API

- Un standard ouvert pour décrire les API REST.
- Permet de documenter de manière structurée les endpoints, les paramètres, les réponses, etc.
- Facilite la compréhension et l'utilisation des API par les développeurs.

2. **Utilité dans les pipelines de données :**
   - **Documentation claire** : Comprendre rapidement comment interagir avec une API.
   - **Test interactif** : Tester les endpoints directement depuis l'interface Swagger UI.
   - **Validation des données** : S'assurer que les requêtes et réponses respectent le schéma défini.

3. Utilisation d’API

Swagger/OpenAPI - Documentation et test des API



3. Utilisation d’API

Utilisation de l'API avec Python (librairie *requests*)

- **Bibliothèque Requests :**
  - Simplifie les interactions HTTP en Python.
- **Cas d'utilisation :**
  - Récupérer des données depuis une API.
  - Envoyer des données à une API.

3. Utilisation d’API

Utilisation de l'API avec Python (librairie *requests*)

```python
import requests

# Requête GET
response = requests.get('https://api.exemple.com/data')
data = response.json()  # Convertir la réponse en JSON

# Requête POST
payload = {'key1': 'value1', 'key2': 'value2'}
response = requests.post('https://api.exemple.com/data', json=payload)
```

3. Utilisation d’API

Sécuriser les interactions avec les API

- **Méthodes d'authentification :**
  - **Clé API** : Une clé unique pour accéder à l'API.
  - **OAuth** : Protocole d'autorisation sécurisé.
  - **JWT (JSON Web Tokens)** : Tokens signés pour l'authentification.
- **Bonnes pratiques :**
  - Ne jamais exposer les clés API dans le code (utiliser des variables d'environnement → fichier .env).
  - Utiliser HTTPS pour chiffrer les communications.

3. Utilisation d’API

Sécuriser les interactions avec les API

```python
import os
import requests

api_key = os.getenv('API_KEY')  # Clé API stockée dans les variables d'environnement
headers = {'Authorization': f'Bearer {api_key}'}
response = requests.get('https://api.exemple.com/data', headers=headers)
```

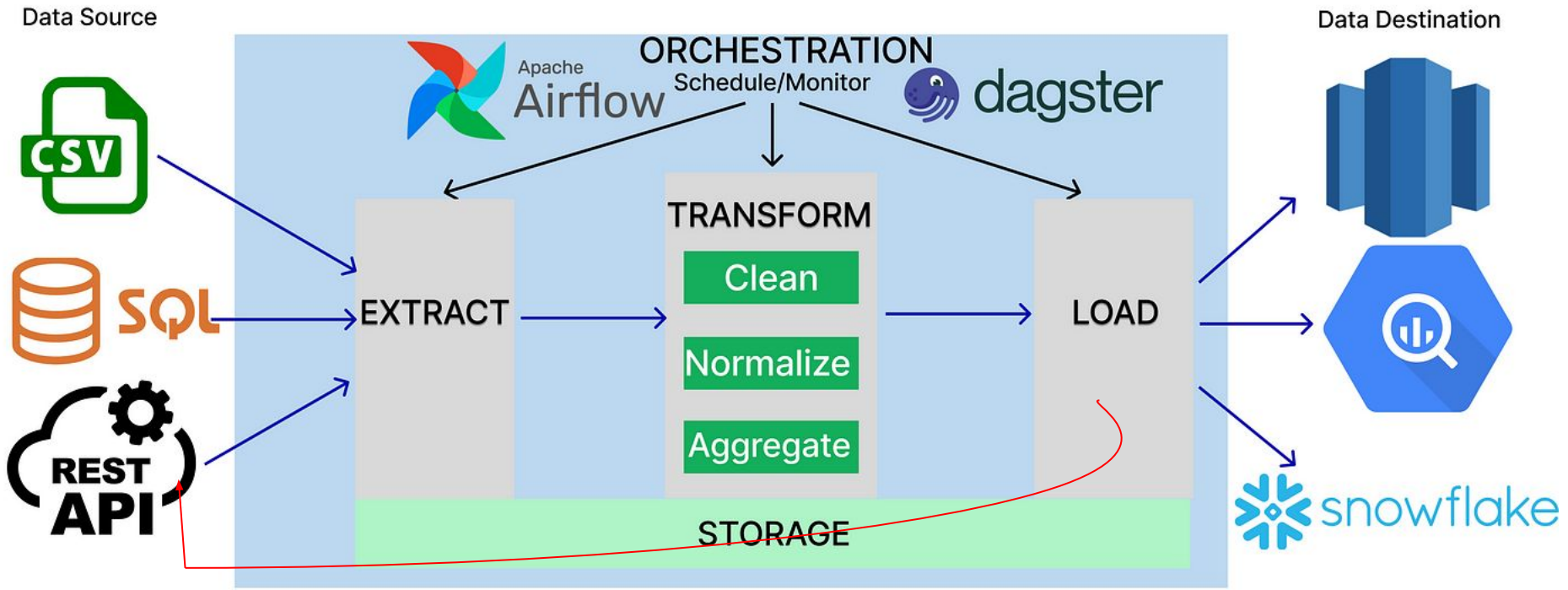
3. Utilisation d’API

Intégration des API dans un pipeline de données

- **Rôle des API :**
  - Source de données pour l'extraction (E).
  - Point de destination pour le chargement (L).
- **Cas d'utilisation :**
  - Ingérer des données en temps réel depuis une API.
  - Envoyer des données transformées vers une API.
- **Exemple de pipeline :**
  - Extraction : Récupérer des données depuis une API.
  - Transformation : Nettoyer et transformer les données avec Pandas.
  - Chargement : Envoyer les données transformées vers une autre API ou une base de données.

3. Utilisation d’API

Intégration des API dans un pipeline de données



3. Utilisation d’APIs

4. Bonnes pratiques générales

4. Bonnes pratiques générales

Pourquoi les bonnes pratiques sont essentielles?

- **Objectif :**
  - Éviter les pannes, réduire la dette technique, faciliter la collaboration.
- **Conséquences d'un code mal conçu :**
  - Données corrompues, temps de débogage explosif, coûts cloud incontrôlés.
- **Exemple:**
  - Un pipeline sans gestion d'erreurs et qui échoue silencieusement ➔ Données manquantes pendant des semaines.

4. Bonnes pratiques générales

Séparation claire entre exploration et production

- **Notebooks (Jupyter, Colab) :**
  - *À utiliser pour :* Exploration, prototypage rapide, visualisation.
  - *À éviter pour :* Le code de production (difficile à versionner, peu scalable).
  - *Best practice :* Convertir le notebook en scripts Python modulaires après validation.

4. Bonnes pratiques générales

Environnements locaux vs production

- **Pourquoi différencier les environnements ?**
  - Éviter les "ça marche sur ma machine".
  - Adapter les outils aux besoins (ex : SQLite en local vs Postgres/Redshift en prod).
- **Outils recommandés :**
  - *Local :* SQLite, Docker pour isoler les dépendances.
  - *Production :* PostgreSQL, BigQuery, solutions cloud.

4. Bonnes pratiques générales

Avantages de SQLite en phase de test

Simplicité et légèreté :

- SQLite est un moteur de base de données léger qui ne nécessite pas de serveur indépendant. Tout est contenu dans un fichier unique, ce qui facilite sa configuration et son utilisation.
- Il n'y a pas besoin de gérer un processus serveur ou de configurer une connexion réseau.

Rapidité pour les tests locaux :

- SQLite est extrêmement rapide pour les petites bases de données ou les environnements à faible concurrence. Cela en fait un choix idéal pour les tests unitaires ou de développement local.

4. Bonnes pratiques générales

Avantages de SQLite en phase de test

Facilité de mise en place :

- Il est souvent préinstallé avec Python et d'autres environnements de développement, ce qui permet de l'utiliser immédiatement sans installations supplémentaires.

Moins de dépendances :

- En phase de test ou de développement, réduire les dépendances externes comme PostgreSQL peut rendre les environnements plus simples et moins coûteux à maintenir.

Économique :

- Pas besoin d'une infrastructure de base de données robuste ou coûteuse (comme un serveur PostgreSQL) pour exécuter des tests simples ou des développements initiaux.

4. Bonnes pratiques générales

Pourquoi PostgreSQL en production ?

Concurrence et performance :

- PostgreSQL est conçu pour gérer plusieurs connexions simultanées et offrir de meilleures performances pour des bases de données volumineuses ou des environnements multi-utilisateurs.
- SQLite, en revanche, est limité en termes de concurrence (il ne supporte pas bien les écritures simultanées).

Fonctionnalités avancées :

- PostgreSQL offre des fonctionnalités avancées telles que :
  - Support des transactions complexes et de l'ACID.
  - Types de données personnalisés, JSONB, géospatiaux (PostGIS), etc.
  - Réplication et haute disponibilité.

4. Bonnes pratiques générales

Pourquoi PostgreSQL en production ?

Scalabilité :

- Contrairement à SQLite, PostgreSQL peut gérer des bases de données massives et évoluer horizontalement ou verticalement selon les besoins.

Sécurité et contrôle d'accès :

- PostgreSQL propose des mécanismes de sécurité robustes, tels que les utilisateurs, les rôles et les contrôles d'accès granulaires.

Conformité au déploiement :

- En production, les systèmes doivent souvent être déployés sur des serveurs cloud ou des environnements distribués, où un moteur comme PostgreSQL est plus adapté.

4. Bonnes pratiques générales

Documentation et lisibilité du code

- **Documentation :**
  - *Docstrings* pour expliquer les fonctions et les classes.
  - *README.md* pour décrire le pipeline, ses dépendances, et comment l'exécuter
  - Format du code

Utiliser un style de code standard (ex : PEP8 pour Python).

4. Bonnes pratiques générales

Documentation et lisibilité du code

```
def clean_data(df: pd.DataFrame) -> pd.DataFrame:
    """
    Nettoie les données en supprimant les doublons et les valeurs manquantes.

    Args:
        df (pd.DataFrame): DataFrame d'entrée.

    Returns:
        pd.DataFrame: DataFrame nettoyé.
    """
    return df.dropna().drop_duplicates()
```

4. Bonnes pratiques générales

Gestion des erreurs et logs

- **Pourquoi logger ?**
  - Diagnostiquer les pannes, auditer les exécutions.
- **Bonnes pratiques :**
  - Utiliser le module logging plutôt que print().
  - Capturer les exceptions avec try/except et logger l’erreur.

4. Bonnes pratiques générales

Gestion des erreurs et logs

```
import logging
logging.basicConfig(filename='pipeline.log', level=logging.INFO)

try:
    df = pd.read_csv('data.csv')
except FileNotFoundError as e:
    logging.error(f"Fichier non trouvé : {e}", exc_info=True)
    raise
```

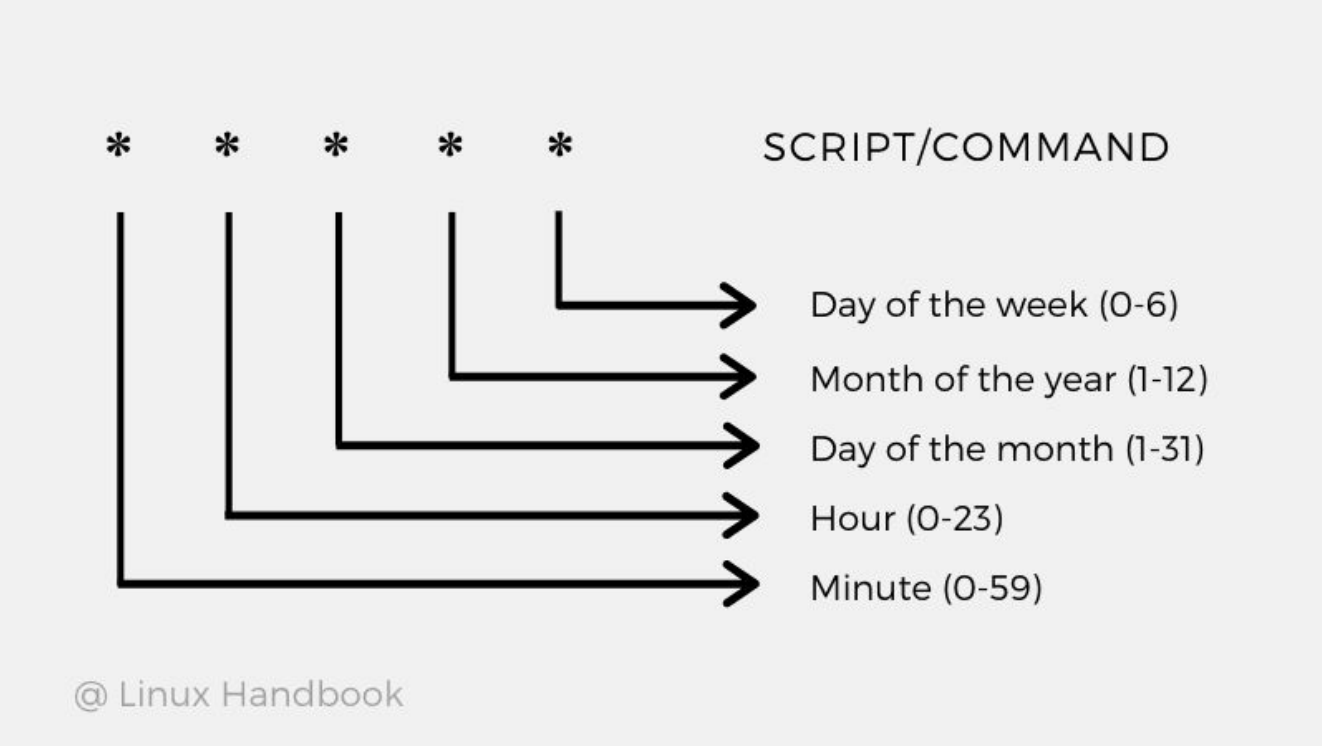
4. Bonnes pratiques générales

Automatisation avec des outils adaptés

- **Cron :**
  - Pour des tâches simples et périodiques (ex : backup quotidien).
  - Exemple de syntaxe : `0 3 * * * /path/to/script.py` (exécution à 3h du matin).
- **Orchestrateurs modernes :**
  - *Dagster*: Pour des workflows complexes avec dépendances.
  - *Prefect* : Alternative plus flexible et moderne.

4. Bonnes pratiques générales

Automatisation avec des outils adaptés



4. Bonnes pratiques générales

Tests et validation des données

Types de tests :

- *Tests unitaires* (avec `pytest`) : Vérifier des fonctions individuelles.
- *Tests d'intégration* : Valider des workflows complets.
- *Validation des données* (avec `pandera` ou `great_expectations`) : S'assurer que les données respectent un schéma.

```
def test_clean_data():
    df_input = pd.DataFrame({'col': [1, None, 3]})
    df_output = clean_data(df_input)
    assert df_output.shape[0] == 2  # Vérifie que les NaN sont supprimés
```

4. Bonnes pratiques générales

Gestion des secrets et sécurité

- **À ne jamais faire :**
  - Stocker des mots de passe ou clés API en clair dans le code.
- **Solutions :**
  - Variables d'environnement (ex : os.getenv('DB_PASSWORD')).
  - Outils de gestion de secrets : *Vault*, *AWS Secrets Manager*.

```
from dotenv import load_dotenv
load_dotenv()  # Charge les variables depuis un fichier .env
db_password = os.getenv('DB_PASSWORD')
```

4. Bonnes pratiques générales

Optimisation des coûts et des performances

- **Optimisation des coûts cloud :**
  - Arrêter les instances inutilisées (ex : AWS EC2, BigQuery slots).
  - Utiliser des stockages à froid (ex : AWS S3 Glacier) pour les archives.
- **Optimisation du code :**
  - Utiliser le parallélisme (ex : multiprocessing, Dask).
  - Éviter les requêtes SQL coûteuses (ex : SELECT *).

4. Bonnes pratiques générales

Versionnage et collaboration

- **Git :**
  - Versionner le code, les configurations, et les requêtes SQL.
  - Utiliser des branches (feature, bugfix) pour chaque nouvelle feature
- **Fichier .gitignore :**
  - Exclure les fichiers sensibles (.env, data/, *.log).

4. Bonnes pratiques générales

Conteneurisation et déploiement

- **Docker :**
  - Empaqueter le pipeline et ses dépendances pour un déploiement reproductible.
- **Orchestration :**
  - *Kubernetes* : Pour scaler les pipelines sur des clusters.

Pour plus tard…

4. Bonnes pratiques générales

En résumé, les 10 commandements des pipelines (et même plus !)

- Tu modulariseras ton code.
- Tu documenteras chaque fonction.
- Tu ne stockeras jamais de secrets en clair.
- Tu testeras avant de déployer.
- Tu surveilleras tes pipelines.
- Tu choisiras les bons outils pour le bon usage.
- Tu optimiseras tes coûts cloud.
- Tu versionneras tout.
- Tu conteneuriseras pour la reproductibilité.
- Tu apprendras de tes erreurs.

4. Bonnes pratiques générales