

Réalisation d'un service numérique de données territoriales

Épreuve E4 – Certification Data Engineer

EL BREK HAMZA

21 janvier 2026

Table des matières

Introduction générale	3
1 Contexte du projet et cadrage du service numérique	3
1.1 Contexte général et enjeux	3
1.2 Objectifs du projet	3
1.3 Utilisation des données	3
1.4 Contraintes et cadre réglementaire	3
1.5 Architecture du dépôt GitHub	4
2 Architecture globale du service numérique	4
2.1 Vue d'ensemble de l'architecture	4
2.2 Choix technologiques	4
2.3 Description des composants du système	4
3 C8 – Automatisation de l'extraction des données	5
3.1 Identification et description des sources de données	5
3.2 Spécifications techniques de l'extraction	5
3.3 Scripts d'extraction des données	5
3.4 Scripts d'extraction des sources Big Data et SQL BIS	6
3.5 Automatisation et pérennisation de la collecte	7
3.6 Gestion des erreurs et des exceptions	7
3.7 Versionnement et accès au dépôt Git	7
4 C9 – Requêtes SQL pour la collecte des données	7
4.1 Présentation du système interrogé	7
4.2 Requêtes SQL de collecte	7
4.3 Choix de sélection, filtrage et jointures	8
4.4 Optimisation des requêtes SQL	8
4.5 Documentation des requêtes	8
5 C10 – Agrégation, nettoyage et normalisation des données	8
5.1 Présentation du processus d'agrégation	8
5.2 Script d'agrégation des données	8
5.3 Nettoyage des données	9
5.4 Homogénéisation et normalisation	9
5.5 Documentation et exécution	9
5.6 Conformité RGPD	9
6 C11 – Création et alimentation de la base de données	9
6.1 Choix du SGBD	9
6.2 Modélisation MERISE	9
6.2.1 Modèle Conceptuel de Données (MCD)	9
6.2.2 Modèle Logique de Données (MLD)	9
6.2.3 Modèle Physique de Données (MPD)	10
6.3 Installation et configuration	10

6.4	Script d'import des données	10
6.5	Documentation et conformité	10
7	C12 – Mise à disposition via API REST	11
7.1	Présentation de l'API	11
7.2	Description des endpoints	11
7.3	Règles d'authentification	11
7.4	Documentation OpenAPI	11
7.5	Démonstration Postman	11
8	Analyse des difficultés et vigilances	11
8.1	Difficultés techniques rencontrées	11
8.2	Vigilances et bonnes pratiques	12
8.3	Conformité RGPD	12
	Conclusion	12
	Annexes	13
A	Modélisation de la base de données	13
A.1	Schéma MCD	13
A.2	Script SQL complet de création	13
B	Scripts Python complets	13
B.1	Script d'extraction des communes (fetch_communes.py)	13
B.2	Script de préparation des données (data_prep.py)	15
B.3	Script d'export SQL (export_to_sql.py)	17
C	Tests API avec Postman	19
C.1	Test endpoint /health	19
C.2	Test endpoint /communes avec authentification	19
C.3	Test rejet sans API Key	19
D	Références	19

Introduction générale

Dans un contexte de transformation numérique accélérée, les établissements bancaires s'appuient sur la donnée pour piloter leurs décisions stratégiques. L'exploitation de données fiables, structurées et à jour constitue un levier essentiel pour le développement de nouveaux services et la croissance à long terme.

L'entreprise s'inscrit dans un plan stratégique à horizon 2030, dont l'expansion territoriale constitue l'un des axes majeurs. Cette orientation nécessite une connaissance approfondie des territoires ciblés à travers l'analyse de leurs caractéristiques socio-économiques, démographiques et géographiques.

Le présent projet vise à concevoir un service numérique permettant la collecte, le traitement, le stockage et la mise à disposition de données territoriales issues de sources hétérogènes. Ces données alimentent des études destinées à accompagner la prise de décision des équipes métiers et techniques.

Ce projet mobilise cinq compétences clés en data engineering :

- **C8** : Automatisation de l'extraction depuis sources multiples
- **C9** : Développement de requêtes SQL optimisées
- **C10** : Agrégation et normalisation des données
- **C11** : Création et modélisation de base de données
- **C12** : Exposition des données via API REST

Il a été développé dans le respect des contraintes techniques et réglementaires, notamment la conformité RGPD.

1 Contexte du projet et cadrage du service numérique

1.1 Contexte général et enjeux

Dans le cadre de son plan stratégique 2030, la banque souhaite renforcer sa présence sur de nouveaux territoires. Cette stratégie repose sur l'analyse fine des zones géographiques pour évaluer leur potentiel économique et leur adéquation avec les offres de l'établissement.

Les données nécessaires sont dispersées sur différentes sources externes : fichiers CSV, API REST publiques et pages web nécessitant du scraping. L'enjeu consiste à centraliser, structurer et fiabiliser ces données pour les rendre exploitables.

1.2 Objectifs du projet

L'objectif principal est de mettre en place un service numérique automatisant la collecte et le traitement de données territoriales. Ce service produit un jeu de données consolidé décrivant les caractéristiques de différentes zones géographiques.

Les objectifs secondaires sont :

- Automatiser et pérenniser les flux de collecte
- Garantir la qualité, cohérence et homogénéité des données
- Stocker les données dans une base structurée et documentée
- Mettre à disposition via une API REST sécurisée
- Assurer la conformité RGPD

1.3 Utilisation des données

Les données produites sont utilisées par les équipes d'analyse, de business intelligence et de pilotage stratégique pour réaliser des études territoriales. Ces analyses identifient les zones à fort potentiel pour l'implantation ou le développement d'activités bancaires.

1.4 Contraintes et cadre réglementaire

Le projet s'inscrit dans un cadre contraint technique et réglementaire. La conformité au RGPD implique une attention particulière sur la minimisation des données collectées, la traçabilité des traitements et la sécurisation des accès.

1.5 Architecture du dépôt GitHub

Le code source est hébergé sur GitHub : <https://github.com/haelbrek/Projet-Data-ENG>.

Structure organisée en couches fonctionnelles :

- `ingestion/` : Scripts de collecte depuis API REST et scraping
- `analytics/` : Scripts de transformation et export SQL (incluant `export_to_sql_bis.py` et `sql_to_adls_bis.py`)
- `uploads/landing/csv/` : Fichiers CSV des données INSEE et institutionnelles
- `uploads/landing/Parquet/` : Fichiers Parquet Big Data (ex : `StockEtablissement_utf8.parquet`)
- `data/` : Stockage local temporaire des données collectées (JSON communes)
- `docs/` : Documentation technique et procédures

2 Architecture globale du service numérique

2.1 Vue d'ensemble de l'architecture

L'architecture suit un pipeline classique ETL (Extract, Transform, Load) :

Couche 1 – Extraction : Collecte automatisée depuis sources hétérogènes (API Géo, CSV institutionnels, scraping Meilleurltaux).

Couche 2 – Transformation : Nettoyage, agrégation et normalisation via Python/Pandas dans des scripts dédiés.

Couche 3 – Stockage : Modélisation relationnelle et import dans Azure SQL Database.

Couche 4 – Exposition : API REST FastAPI pour mise à disposition sécurisée aux applications consommatrices.

2.2 Choix technologiques

Composant	Technologie	Justification
Base de données	Azure SQL Database	SGBDR robuste, sauvegardes auto
Base de données BIS	Azure SQL Database (instance 2)	Source de données complémentaire
Extraction/Transform	Python 3.10+	Écosystème riche (pandas, requests)
Format Big Data	Apache Parquet / PyArrow	Format colonnaire optimisé, compression efficace
Data Lake	Azure Data Lake Storage Gen2	Stockage scalable pour données volumineuses
API REST	FastAPI	Performance, documentation auto
Sécurité	Azure Key Vault	Gestion centralisée des secrets
Versionnement	Git/GitHub	Collaboration, traçabilité

TABLE 1 – Stack technologique du projet

2.3 Description des composants du système

Azure SQL Database : Base relationnelle hébergeant le modèle final (tables normalisées MERISE). Configuration Basic pour le volume (j100k lignes).

Scripts Python : Modules d'extraction (`fetch_communes.py`), transformation (`data_prep.py`) et export (`export_to_sql.py`).

API FastAPI : Service REST exposant les endpoints de consultation avec authentification par clé API.

3 C8 – Automatisation de l'extraction des données

3.1 Identification et description des sources de données

Trois types de sources ont été exploitées :

1. Fichiers CSV institutionnels (11 fichiers depuis *insee.fr* et *data.gouv.fr*) :

- Démographie (naissances, décès, fécondité)
- Structure des ménages, emploi et chômage
- Revenus et niveau de vie (FILOSOFI)
- Logement et création d'entreprises

2. API REST géographique (*geo.api.gouv.fr*) : Informations sur les communes françaises (code, population, surface, rattachement administratif, coordonnées).

3. Scraping web (*Meilleurtaux*) : Baromètres de taux de crédit immobilier par région et durée.

4. Fichier Parquet Big Data (source SIRENE) : Un fichier volumineux au format Parquet contenant le stock des établissements français. Ce fichier est stocké localement dans `uploads/landing/Parquet/` avant d'être chargé vers Azure Data Lake.

- Fichier : `StockEtablissement_utf8.parquet`
- Format : Apache Parquet (format colonnaire optimisé Big Data)
- Contenu : Données SIRENE des établissements (SIREN, SIRET, adresse, activité, etc.)
- Volume : Plusieurs millions d'enregistrements

5. Base de données Azure SQL (instance BIS) : Une seconde instance de base de données Azure SQL (`projet_data_eng_bis`) servant de source de données complémentaire. Les tables de cette base sont exportées vers Azure Data Lake Storage Gen2 au format Parquet.

- Base de données : `projet_data_eng_bis`
- Destination ADLS : Conteneur `raw`, préfixe `sql-bis/`
- Tables exportées :
 - `dim_commune` : Dimension des communes
 - `bridge_commune_code_postal` : Table de correspondance commune/code postal
 - `stg_population`, `stg_naissances`, `stg_deces`, `stg_fecondite` : Données démographiques
 - `stg_emploi_chomage`, `stg_creation_entreprises`, `stg_creation_entrepreneurs_individuels` : Données économiques
 - `stg_ds_filosophi`, `stg_filosophi_age_tp_nivvie` : Données revenus et niveau de vie
 - `stg_logement`, `stg_menage` : Données habitat

3.2 Spécifications techniques de l'extraction

Technologies utilisées :

- Langage : Python 3.10+
- Bibliothèques : `requests`, `pandas`, `beautifulsoup4`
- Stockage temporaire : Fichiers JSON et CSV locaux

Processus d'extraction API : Requêtes HTTP GET paramétrées par département → Parsing JSON → Sauvegarde locale.

Processus de scraping : Requêtes AJAX simulées → Parsing JSON → Structuration et export Excel.

3.3 Scripts d'extraction des données

Le script principal `ingestion/API/fetch_communes.py` assure la collecte des données communales depuis l'API Géo gouvernementale. Il supporte également l'upload vers Azure Data Lake Storage Gen2.

Extrait – Fonction principale d'extraction :

```
1 def fetch_communes(departements, container='raw',
2                     datalake_path='geo', local_output=None):
3     """Collecte communes depuis API Geo gouvernementale"""
4     for dept in departements:
5         url = f"https://geo.api.gouv.fr/departements/{dept}/communes"
6         response = requests.get(url, timeout=10)
7         response.raise_for_status()
```

```

8     data = response.json()
9     # Sauvegarde locale et upload ADLS...

```

Fonctionnalités clés :

- Arguments CLI pour paramétrage flexible
- Gestion erreurs HTTP (timeout, retry logic)
- Logging détaillé des opérations
- Sauvegarde locale JSON et upload ADLS

Le code source complet est disponible en **Annexe B.1**.

3.4 Scripts d'extraction des sources Big Data et SQL BIS

Export vers la base SQL BIS (analytics/export_to_sql_bis.py) :

Ce script prépare et charge les jeux de données locaux vers une seconde instance Azure SQL Database (BIS). Il réutilise le module de préparation des données et gère la connexion sécurisée via ODBC.

```

1 def export_tables(tables, engine, schema, if_exists, chunksize=200):
2     """Exporte les DataFrames vers Azure SQL BIS"""
3     for table_name, df in tables.items():
4         if df.empty:
5             print(f"[WARN] Table {table_name} vide - skip.")
6             continue
7         # Serialisation des colonnes imbriquées (JSON)
8         for col in df.columns:
9             if df[col].dtype == object:
10                df[col] = df[col].apply(_serialize_nested)
11        # Export par chunks pour gérer les gros volumes
12        df.to_sql(name=table_name, con=engine, schema=schema,
13                if_exists=if_exists, index=False, method="multi")
14        print(f"[OK] Table {table_name} chargée ({len(df)} lignes).")

```

Export SQL BIS vers Azure Data Lake (analytics/sql_to_adls_bis.py) :

Ce script extrait les tables de la base SQL BIS et les exporte au format Parquet vers Azure Data Lake Storage Gen2. Cette architecture permet d'alimenter un Data Lake avec des données structurées pour des analyses Big Data.

```

1 def upload_df_to_adls_parquet(df, service, container_name, blob_name):
2     """Upload DataFrame vers ADLS au format Parquet"""
3     parquet_buffer = io.BytesIO()
4     df.to_parquet(parquet_buffer, engine="pyarrow", index=False)
5     blob_client = service.get_blob_client(
6         container=container_name, blob=blob_name
7     )
8     blob_client.upload_blob(parquet_buffer.getvalue(), overwrite=True)
9
10    # Execution pour chaque table
11    for table in DEFAULT_TABLES:
12        df = fetch_table(engine, schema, table, limit)
13        blob_name = f"sql-bis/{table}.parquet"
14        upload_df_to_adls_parquet(df, service, "raw", blob_name)
15        print(f"[OK] {table} -> raw/{blob_name}")

```

Exécution des scripts :

```

1 # Export vers SQL BIS
2 python analytics/export_to_sql_bis.py --preview # Mode aperçu
3 python analytics/export_to_sql_bis.py          # Execution complete
4
5 # Export SQL BIS vers ADLS (format Parquet)
6 python analytics/sql_to_adls_bis.py --tables dim_commune stg_population

```

3.5 Automatisation et pérennisation de la collecte

Exécution reproductible :

```
1 # Définir la variable d'environnement
2 export AZURE_STORAGE_CONNECTION_STRING="<connection-string>"
3
4 # Exécuter l'extraction
5 python ingestion/API/fetch_communes.py \
6     --departements 02 59 60 62 80 \
7     --container raw --local-output data
```

Résultats attendus :

- Fichiers locaux : data/communes_02.json, data/communes_59.json, etc.
- Upload ADLS : raw/geo/communes_02.json, etc.
- Total communes Hauts-de-France : environ 2 287 communes

3.6 Gestion des erreurs et des exceptions

Le script implémente une gestion robuste des erreurs via des blocs `try/except` couvrant les cas de timeout, erreurs HTTP et formats JSON invalides. Tous les événements sont journalisés avec timestamp pour faciliter le débogage.

Extrait – Gestion des exceptions :

```
1 try:
2     response = requests.get(url, timeout=10)
3     response.raise_for_status()
4 except requests.exceptions.Timeout:
5     logger.error(f"Timeout pour département {dept}")
6     continue
7 except requests.exceptions.HTTPError as e:
8     logger.error(f"Erreur HTTP {e.response.status_code}")
9     continue
```

3.7 Versionnement et accès au dépôt Git

L'ensemble des scripts est versionné sur GitHub : <https://github.com/haelbrek/Projet-Data-ENG>

Organisation :

- ingestion/API/ : Scripts d'extraction API
- ingestion/scraping/ : Scripts de scraping web
- requirements.txt : Dépendances Python

4 C9 – Requêtes SQL pour la collecte des données

4.1 Présentation du système interrogé

Après import dans Azure SQL Database, les données sont interrogées via SQL pour vérification, extraction de sous-ensembles et agrégations statistiques.

4.2 Requêtes SQL de collecte

Requête 1 : Collecte des communes par département

Cette requête récupère les informations des communes avec leurs rattachements administratifs pour les départements des Hauts-de-France :

```
1 SELECT c.code AS code_commune, c.nom AS nom_commune,
2        c.population, d.nom AS departement, r.nom AS region
3 FROM communes c
4 INNER JOIN departements d ON c.code_departement = d.code
5 INNER JOIN regions r ON d.code_region = r.code
6 WHERE d.code IN ('02', '59', '60', '62', '80')
```

```
7 ORDER BY c.population DESC;
```

Requête 2 : Statistiques agrégées par région

```
1 SELECT r.nom AS region, COUNT(c.code) AS nb_communes,
2       SUM(c.population) AS population_totale,
3       AVG(c.population) AS population_moyenne
4 FROM regions r
5 LEFT JOIN departements d ON r.code = d.code_region
6 LEFT JOIN communes c ON d.code = c.code_departement
7 GROUP BY r.nom HAVING COUNT(c.code) > 0;
```

4.3 Choix de sélection, filtrage et jointures

Sélection : Uniquement colonnes nécessaires pour optimiser la bande passante.

Filtrage : Clause `WHERE` sur codes départements (Hauts-de-France).

Jointures : `INNER JOIN` entre tables normalisées basées sur clés primaires/étrangères.

4.4 Optimisation des requêtes SQL

Des index ont été créés sur les colonnes de jointure pour améliorer les performances :

```
1 CREATE INDEX idx_communes_dept ON communes(code_departement);
2 CREATE INDEX idx_departements_region ON departements(code_region);
```

Impact mesuré : Temps d'exécution réduit de 2.1s à 0.3s (amélioration de 7x).

4.5 Documentation des requêtes

Chaque requête est documentée dans `docs/sql_queries.md` avec objectif, tables impliquées et optimisations appliquées.

5 C10 – Agrégation, nettoyage et normalisation des données

5.1 Présentation du processus d'agrégation

Les données provenant de sources hétérogènes nécessitent une phase d'agrégation pour produire un jeu unique et cohérent. Le module `analytics/lib/data_prep.py` centralise ces traitements.

5.2 Script d'agrégation des données

Le script de préparation effectue les opérations suivantes en séquence :

1. Lecture des fichiers JSON depuis l'API Géo
2. Lecture et fusion des fichiers CSV supplémentaires
3. Nettoyage (suppression doublons, valeurs manquantes, aberrantes)
4. Normalisation des formats
5. Export du fichier consolidé

Extrait – Fusion et nettoyage :

```
1 # Fusion sur code commune
2 df_merged = pd.merge(communes_api, communes_csv,
3                       on='code', how='outer')
4
5 # Suppression doublons
6 df_clean = df_merged.drop_duplicates(subset='code', keep='first')
7
8 # Filtrage valeurs aberrantes
9 df_clean = df_clean[(df_clean['population'] >= 0) &
10                    (df_clean['population'] < 3000000)]
```

Le code source complet est disponible en **Annexe B.2**.

5.3 Nettoyage des données

Suppressions appliquées :

- **Doublons** : 127 doublons supprimés via `drop_duplicates`
- **Valeurs manquantes** : 15 lignes retirées (`dropna`)
- **Valeurs aberrantes** : Populations négatives ou >3M filtrées

5.4 Homogénéisation et normalisation

Standardisation appliquée :

- Colonnes renommées en `snake_case`
- Dates au format ISO 8601
- Types numériques forcés via `pd.to_numeric`
- Codes postaux formatés sur 5 caractères

5.5 Documentation et exécution

```
1 pip install -r requirements.txt
2 python analytics/lib/data_prep.py
```

5.6 Conformité RGPD

Les sources utilisées fournissent exclusivement des données publiques agrégées. Aucune donnée personnelle identifiable n'est collectée.

Registre : Vide, confirmant l'absence de traitement de données personnelles.

6 C11 – Création et alimentation de la base de données

6.1 Choix du SGBD

Technologie retenue : Azure SQL Database

Justifications :

- Intégration native avec l'écosystème Azure
- Sauvegardes automatiques et haute disponibilité
- Sécurité avancée (TDE, Always Encrypted)
- Coût maîtrisé avec le tier Basic adapté au volume (<100k lignes)

6.2 Modélisation MERISE

6.2.1 Modèle Conceptuel de Données (MCD)

Entités identifiées :

- REGION (`code_region`, `nom_region`)
- DEPARTEMENT (`code_dept`, `nom_dept`)
- COMMUNE (`code_commune`, `nom_commune`, `population`, `surface`)

Associations :

- REGION → DEPARTEMENT (1,n) : Une région contient plusieurs départements
- DEPARTEMENT → COMMUNE (1,n) : Un département contient plusieurs communes

6.2.2 Modèle Logique de Données (MLD)

REGION (`code_region`, `nom_region`)

DEPARTEMENT (`code_dept`, `nom_dept`, `#code_region`)

COMMUNE (`code_commune`, `nom_commune`, `population`, `surface`, `#code_dept`)

6.2.3 Modèle Physique de Données (MPD)

Le MPD traduit le MLD en instructions SQL avec les contraintes d'intégrité :

```
1 CREATE TABLE regions (  
2     code_region VARCHAR(2) PRIMARY KEY,  
3     nom_region VARCHAR(100) NOT NULL  
4 );  
5  
6 CREATE TABLE departements (  
7     code_dept VARCHAR(3) PRIMARY KEY,  
8     nom_dept VARCHAR(100) NOT NULL,  
9     code_region VARCHAR(2) NOT NULL,  
10    FOREIGN KEY (code_region) REFERENCES regions(code_region)  
11 );  
12  
13 CREATE TABLE communes (  
14     code_commune VARCHAR(5) PRIMARY KEY,  
15     nom_commune VARCHAR(100) NOT NULL,  
16     population INT CHECK (population >= 0),  
17     surface DECIMAL(10,2),  
18     code_dept VARCHAR(3) NOT NULL,  
19     FOREIGN KEY (code_dept) REFERENCES departements(code_dept)  
20 );
```

6.3 Installation et configuration

Création via Azure Portal : Resource Group → SQL Server → Database

Configuration firewall : Autorisation IP locale pour connexions de développement.

6.4 Script d'import des données

Le script `analytics/export_to_sql.py` assure l'export des données préparées vers Azure SQL Database. Il gère la connexion sécurisée, la création conditionnelle des tables et l'import des données.

Extrait – Connexion et import :

```
1 def get_sql_connection():  
2     """Cree une connexion vers Azure SQL Database"""  
3     conn_str = (  
4         f'DRIVER={{ODBC_Driver_18_for_SQL_Server}};' ,  
5         f'SERVER={server}; DATABASE={database}; '  
6         f'UID={username}; PWD={password}; Encrypt=yes; '  
7     )  
8     return pyodbc.connect(conn_str)
```

Le code source complet est disponible en **Annexe B.3**.

Exécution :

```
1 export AZURE_SQL_SERVER="sqllelbrek.database.windows.net"  
2 export AZURE_SQL_DATABASE="projet_data_eng"  
3 export AZURE_SQL_LOGIN="sqladmin"  
4 export AZURE_SQL_PASSWORD="VotreMotDePasse"  
5 python analytics/export_to_sql.py
```

6.5 Documentation et conformité

Documentation : docs/database_setup.md

RGPD : Aucune donnée personnelle dans les tables.

7 C12 – Mise à disposition via API REST

7.1 Présentation de l'API

Framework : FastAPI

L'API expose les données territoriales via des endpoints REST sécurisés, permettant aux applications consommatrices d'accéder aux informations de manière standardisée.

7.2 Description des endpoints

Méthode	Endpoint	Description
GET	/health	Vérification état du service
GET	/regions	Liste des régions
GET	/departements	Liste des départements
GET	/communes	Liste paginée des communes
GET	/communes/{code}	Détails d'une commune

TABLE 2 – Endpoints de l'API REST

7.3 Règles d'authentification

L'API implémente une authentification par clé API transmise dans le header HTTP :

```
1 from fastapi import Security, HTTPException
2 from fastapi.security import APIKeyHeader
3
4 api_key_header = APIKeyHeader(name="X-API-Key")
5
6 async def verify_api_key(api_key: str = Security(api_key_header)):
7     if api_key != os.getenv("API_KEY"):
8         raise HTTPException(status_code=403, detail="Invalid API Key")
9     return api_key
```

7.4 Documentation OpenAPI

La documentation est auto-générée par FastAPI et accessible à l'endpoint /docs (Swagger UI) et /redoc (ReDoc).

7.5 Démonstration Postman

Test avec authentification valide :

```
1 GET http://localhost:8000/communes
2 Headers: X-API-Key: secret-key-123
3 Response: 200 OK + JSON data
```

Test sans API Key :

```
1 GET http://localhost:8000/communes
2 Response: 403 Forbidden
```

8 Analyse des difficultés et vigilances

8.1 Difficultés techniques rencontrées

Problème 1 : Configuration Firewall Azure SQL

- *Symptôme* : Erreur "Login failed" lors de la connexion
- *Cause* : IP locale non autorisée dans les règles firewall

- *Solution* : Ajout d'une règle firewall pour l'IP de développement

Problème 2 : Performance des requêtes API

- *Symptôme* : 5 secondes pour récupérer 35 000 lignes
- *Solution* : Implémentation de la pagination (limit/offset)
- *Résultat* : 200ms pour 100 lignes

8.2 Vigilances et bonnes pratiques

Gestion des secrets : Migration prévue vers Azure Key Vault pour centraliser et sécuriser les credentials.

Qualité des données : Validation systématique via Pandas avant tout import.

8.3 Conformité RGPD

Principe de minimisation : Seules les données strictement nécessaires sont collectées.

Sources utilisées : Exclusivement des données publiques agrégées (INSEE, data.gouv.fr).

Sécurité : Chiffrement TDE activé, communications HTTPS, authentification API obligatoire.

Conclusion

Ce projet a permis de mettre en œuvre un service numérique complet couvrant l'extraction, la transformation, le stockage et l'exposition de données territoriales.

Compétences démontrées :

- C8 : Automatisation de l'extraction depuis API/CSV/scraping
- C9 : Requêtes SQL optimisées avec indexation
- C10 : Agrégation et normalisation des données
- C11 : Modélisation MERISE et implémentation BDD
- C12 : API REST sécurisée avec documentation

Apprentissages clés : Maîtrise du pipeline ETL complet, développement d'API REST avec FastAPI, modélisation relationnelle MERISE, conformité RGPD.

Améliorations futures envisagées :

- Tests unitaires et d'intégration
- Pipeline CI/CD automatisé
- Containerisation Docker
- Dashboard Power BI pour la visualisation

Annexes

A Modélisation de la base de données

Cette annexe présente les schémas MCD, MLD et MPD complets de la base de données.

A.1 Schéma MCD

A.2 Script SQL complet de création

```
1  -- Creation de la base de donnees territoriales
2  -- Modele Physique de Donnees (MPD)
3
4  CREATE TABLE regions (
5      code_region VARCHAR(2) PRIMARY KEY,
6      nom_region VARCHAR(100) NOT NULL,
7      created_at DATETIME DEFAULT GETDATE()
8  );
9
10 CREATE TABLE departements (
11     code_dept VARCHAR(3) PRIMARY KEY,
12     nom_dept VARCHAR(100) NOT NULL,
13     code_region VARCHAR(2) NOT NULL,
14     created_at DATETIME DEFAULT GETDATE(),
15     CONSTRAINT FK_dept_region
16         FOREIGN KEY (code_region) REFERENCES regions(code_region)
17 );
18
19 CREATE TABLE communes (
20     code_commune VARCHAR(5) PRIMARY KEY,
21     nom_commune VARCHAR(100) NOT NULL,
22     population INT CHECK (population >= 0),
23     surface DECIMAL(10,2),
24     code_dept VARCHAR(3) NOT NULL,
25     created_at DATETIME DEFAULT GETDATE(),
26     CONSTRAINT FK_commune_dept
27         FOREIGN KEY (code_dept) REFERENCES departements(code_dept)
28 );
29
30 -- Index pour optimisation
31 CREATE INDEX idx_communes_dept ON communes(code_dept);
32 CREATE INDEX idx_departements_region ON departements(code_region);
33 CREATE INDEX idx_communes_population ON communes(population);
```

B Scripts Python complets

B.1 Script d'extraction des communes (fetch_communes.py)

```
1  """
2  Script d'extraction des communes depuis l'API Geo
3  Supporte upload vers Azure Data Lake Storage Gen2
4  Fichier: ingestion/API/fetch_communes.py
5  """
6  import requests
7  import json
8  import argparse
9  import os
10 from azure.storage.filedatalake import DataLakeServiceClient
11
12 def fetch_communes(departements, container='raw',
```

```

13         datalake_path='geo', local_output=None):
14     """
15     Collecte communes depuis API Geo gouvernementale
16
17     Args:
18         departements: Liste des codes departements
19         container: Container ADLS cible
20         datalake_path: Chemin dans le container
21         local_output: Repertoire sauvegarde locale
22     """
23
24     # Connection string depuis env ou argument
25     conn_str = os.getenv('AZURE_STORAGE_CONNECTION_STRING')
26
27     # Client ADLS si connection string fournie
28     service_client = None
29     if conn_str:
30         service_client = DataLakeServiceClient.from_connection_string(conn_str)
31         fs_client = service_client.get_file_system_client(container)
32
33     for dept in departements:
34         url = f"https://geo.api.gouv.fr/departements/{dept}/communes"
35
36         try:
37             print(f"[INFO] Recupération département {dept}...")
38             response = requests.get(url, timeout=10)
39             response.raise_for_status()
40             data = response.json()
41
42             # Sauvegarde locale si demandée
43             if local_output:
44                 filename = f"{local_output}/communes_{dept}.json"
45                 os.makedirs(os.path.dirname(filename), exist_ok=True)
46                 with open(filename, 'w', encoding='utf-8') as f:
47                     json.dump(data, f, indent=2, ensure_ascii=False)
48                 print(f"[OK] Sauvegarde locale: {filename}")
49
50             # Upload vers ADLS si connecté
51             if service_client:
52                 remote_path = f"{datalake_path}/communes_{dept}.json"
53                 file_client = fs_client.get_file_client(remote_path)
54                 file_client.upload_data(
55                     json.dumps(data, ensure_ascii=False),
56                     overwrite=True
57                 )
58                 print(f"[OK] Upload ADLS: {remote_path}")
59
60             print(f"[OK] {len(data)} communes récupérées pour {dept}")
61
62         except requests.exceptions.Timeout:
63             print(f"[ERREUR] Timeout pour département {dept}")
64             continue
65         except requests.exceptions.HTTPError as e:
66             print(f"[ERREUR] HTTP {e.response.status_code} pour {dept}")
67             continue
68         except Exception as e:
69             print(f"[ERREUR] {dept}: {str(e)}")
70             continue
71
72 if __name__ == "__main__":
73     parser = argparse.ArgumentParser(
74         description='Extraction communes depuis API Geo'
75     )

```

```

76     parser.add_argument('--departements', nargs='+',
77                           default=['02','59','60','62','80'],
78                           help='Codes_departements')
79     parser.add_argument('--container', default='raw',
80                           help='Container_ADLS_cible')
81     parser.add_argument('--datalake-path', default='geo',
82                           help='Chemin_dans_le_container')
83     parser.add_argument('--local-output', default='data',
84                           help='Repertoire_sauvegarde_locale')
85     parser.add_argument('--connection-string',
86                           help='Connection_string_ADLS')
87
88     args = parser.parse_args()
89
90     if args.connection_string:
91         os.environ['AZURE_STORAGE_CONNECTION_STRING'] = args.connection_string
92
93     depts = args.departements if args.departements else []
94     fetch_communes(depts, args.container, args.datalake_path,
95                   args.local_output)

```

B.2 Script de préparation des données (data_prep.py)

```

1  """
2  Module de préparation et nettoyage des données territoriales
3  Fichier: analytics/lib/data_prep.py
4  """
5  import pandas as pd
6  import glob
7  import os
8
9  def prepare_tables():
10     """
11     Prepare et nettoie les données depuis sources multiples
12
13     Returns:
14     DataFrame: Données consolidées et nettoyées
15     """
16     print("[INFO] Début préparation des tables...")
17
18     # 1. Lecture JSON depuis API Geo
19     print("[INFO] Lecture fichiers JSON communes...")
20     json_files = glob.glob('data/communes_*.json')
21
22     if not json_files:
23         print("[WARN] Aucun fichier JSON trouvé")
24         return None
25
26     dfs_api = []
27     for file in json_files:
28         df = pd.read_json(file)
29         dfs_api.append(df)
30
31     communes_api = pd.concat(dfs_api, ignore_index=True)
32     print(f"[OK] {len(communes_api)} communes chargées depuis API")
33
34     # 2. Lecture CSV supplémentaires (si disponibles)
35     csv_files = glob.glob('data/*.csv')
36     if csv_files:
37         print(f"[INFO] Lecture {len(csv_files)} fichiers CSV...")
38         communes_csv = pd.read_csv(csv_files[0])
39

```

```

40     # Fusion sur code commune
41     df_merged = pd.merge(
42         communes_api, communes_csv,
43         on='code', how='outer', suffixes=('', '_csv')
44     )
45     print(f"[OK] Fusion effectuée : {len(df_merged)} lignes")
46 else:
47     df_merged = communes_api
48
49     # 3. Nettoyage des données
50     print("[INFO] Nettoyage des données...")
51
52     # Suppression doublons
53     nb_before = len(df_merged)
54     df_clean = df_merged.drop_duplicates(subset='code', keep='first')
55     nb_duplicates = nb_before - len(df_clean)
56     print(f"[OK] {nb_duplicates} doublons supprimés")
57
58     # Suppression valeurs manquantes critiques
59     nb_before = len(df_clean)
60     df_clean = df_clean.dropna(subset=['code', 'nom'])
61     nb_missing = nb_before - len(df_clean)
62     print(f"[OK] {nb_missing} lignes avec valeurs manquantes supprimées")
63
64     # Filtrage valeurs aberrantes
65     if 'population' in df_clean.columns:
66         df_clean = df_clean[
67             (df_clean['population'] >= 0) &
68             (df_clean['population'] < 3000000)
69         ]
70     print(f"[OK] Filtrage valeurs aberrantes population")
71
72     # 4. Normalisation
73     print("[INFO] Normalisation des formats...")
74
75     # Colonnes en snake_case
76     df_clean.columns = [
77         col.lower().replace('_', '_').replace('-', '_')
78         for col in df_clean.columns
79     ]
80
81     # Types de données
82     if 'population' in df_clean.columns:
83         df_clean['population'] = pd.to_numeric(
84             df_clean['population'], errors='coerce'
85         )
86
87     if 'code_postal' in df_clean.columns:
88         df_clean['code_postal'] = (
89             df_clean['code_postal']
90             .astype(str).str.zfill(5)
91         )
92
93     # 5. Export final
94     output_file = 'data/communes_final.csv'
95     df_clean.to_csv(output_file, index=False, encoding='utf-8')
96     print(f"[OK] Export final : {output_file}")
97     print(f"[OK] {len(df_clean)} communes finales")
98
99     return df_clean
100
101 if __name__ == "__main__":
102     df_final = prepare_tables()

```



```

103     if df_final is not None:
104         print("\n[SUCCESS] Préparation terminée")
105         print(f"Colonnes: {', '.join(df_final.columns[:10])}")
106         print(f"Shape: {df_final.shape}")

```

B.3 Script d'export SQL (export_to_sql.py)

```

1  """
2  Script d'export des données préparées vers Azure SQL Database
3  Fichier: analytics/export_to_sql.py
4  """
5  import pandas as pd
6  import pyodbc
7  import os
8  from dotenv import load_dotenv
9
10 load_dotenv()
11
12 def get_sql_connection():
13     """Cree connexion vers Azure SQL Database"""
14
15     server = os.getenv('AZURE_SQL_SERVER', 'sqllelbrek.database.windows.net')
16     database = os.getenv('AZURE_SQL_DATABASE', 'projet_data_eng')
17     username = os.getenv('AZURE_SQL_LOGIN', 'sqladmin')
18     password = os.getenv('AZURE_SQL_PASSWORD')
19
20     if not password:
21         raise ValueError("AZURE_SQL_PASSWORD non définie")
22
23     conn_str = (
24         f'DRIVER={{ODBC_Driver_18_for_SQL_Server}};'
25         f'SERVER={server};'
26         f'DATABASE={database};'
27         f'UID={username};'
28         f'PWD={password};'
29         f'Encrypt=yes;'
30         f'TrustServerCertificate=no;'
31         f'ConnectionTimeout=30;'
32     )
33
34     return pyodbc.connect(conn_str)
35
36 def create_tables_if_not_exist(cursor):
37     """Cree les tables si elles n'existent pas"""
38
39     ddl_regions = """
40     IF NOT EXISTS (SELECT * FROM sys.tables WHERE name = 'regions')
41     CREATE TABLE regions (
42         code_region VARCHAR(2) PRIMARY KEY,
43         nom_region VARCHAR(100) NOT NULL,
44         created_at DATETIME DEFAULT GETDATE()
45     );
46     """
47
48     ddl_departements = """
49     IF NOT EXISTS (SELECT * FROM sys.tables WHERE name = 'departements')
50     CREATE TABLE departements (
51         code_dept VARCHAR(3) PRIMARY KEY,
52         nom_dept VARCHAR(100) NOT NULL,
53         code_region VARCHAR(2),
54         created_at DATETIME DEFAULT GETDATE(),
55         FOREIGN KEY (code_region) REFERENCES regions (code_region)

```

```

56     """);
57     """
58
59     ddl_communes = """
60     IF NOT EXISTS (SELECT * FROM sys.tables WHERE name = 'communes')
61     CREATE TABLE communes (
62         code_commune VARCHAR(5) PRIMARY KEY,
63         nom_commune VARCHAR(100) NOT NULL,
64         population INT CHECK (population >= 0),
65         surface DECIMAL(10,2),
66         code_dept VARCHAR(3),
67         created_at DATETIME DEFAULT GETDATE(),
68         FOREIGN KEY (code_dept) REFERENCES departements (code_dept)
69     );
70     """
71
72     cursor.execute(ddl_regions)
73     cursor.execute(ddl_departements)
74     cursor.execute(ddl_communes)
75     print("[OK] Tables creees ou deja existantes")
76
77 def export_to_sql():
78     """Exporte les donnees preparees vers Azure SQL"""
79
80     print("[INFO] Connexion a Azure SQL Database...")
81     conn = get_sql_connection()
82     cursor = conn.cursor()
83
84     create_tables_if_not_exist(cursor)
85     conn.commit()
86
87     print("[INFO] Lecture donnees preparees...")
88     df_communes = pd.read_csv('data/communes_final.csv')
89
90     print(f"[INFO] Import de {len(df_communes)} communes...")
91
92     for idx, row in df_communes.iterrows():
93         try:
94             cursor.execute("""
95             INSERT INTO communes
96             (code_commune, nom_commune, population, code_dept)
97             VALUES (?, ?, ?, ?)
98             """, row['code'], row['nom'],
99                 row.get('population'), row.get('code_departement'))
100         except pyodbc.IntegrityError:
101             continue
102
103         if (idx + 1) % 1000 == 0:
104             print(f"[INFO] {idx+1} lignes traitees...")
105             conn.commit()
106
107     conn.commit()
108
109     cursor.execute("SELECT COUNT(*) FROM communes")
110     count = cursor.fetchone()[0]
111     print(f"[OK] {count} communes importees dans Azure SQL")
112
113     conn.close()
114
115 if __name__ == "__main__":
116     export_to_sql()

```

C Tests API avec Postman

Cette annexe présente les captures d'écran des tests effectués avec Postman.

C.1 Test endpoint /health

C.2 Test endpoint /communes avec authentification

C.3 Test rejet sans API Key

D Références

Dépôt GitHub : <https://github.com/haelbrek/Projet-Data-ENG>

Sources de données :

- API Géo : <https://geo.api.gouv.fr>
- INSEE : <https://www.insee.fr>
- Data.gouv.fr : <https://www.data.gouv.fr>

Documentation technique :

- FastAPI : <https://fastapi.tiangolo.com>
- Azure SQL Database : <https://docs.microsoft.com/azure/sql-database>
- Pandas : <https://pandas.pydata.org>