# IBAM-MASTER ISIE

Projet réalisé dans le cadre du module« Développement de base de composants et services web – IBAM 2025 »


**GROUPE 2**

**SAWADOGO** Yves Dieudonné

**KABORE** W Cedric

**KYELEM** Christian W Thierry

**PROFESSEUR**

**Mr KY Oumar Fadel**


Année académique : 2024–2025

# Développement d'une application RESTful de gestion des ventes avec ETL et monitoring

PostgreSQL, MongoDB, Node.js, Prometheus & Grafana

# Résumé

Ce projet met en place une chaîne complète de traitement et d'exposition de données de ventes. Les données sont d'abord stockées dans une base relationnelle PostgreSQL, puis transférées vers MongoDB via un script ETL Python, qui calcule le montant total de chaque vente. Une API RESTful développée en Node.js / Express expose ces données sous forme de services CRUD. Enfin, l'API est instrumentée avec Prometheus et supervisée via un dashboard Grafana affichant les métriques techniques clés (requêtes HTTP, taux de requêtes, utilisation mémoire).

**Mots-clés :** PostgreSQL, MongoDB, ETL, REST, Node.js, Express, Prometheus, Grafana, Airflow.

# Introduction

Dans les architectures modernes, les données sont rarement consommées directement depuis la base de données brute. Elles sont souvent préparées par un processus ETL (Extraction, Transformation, Chargement), exposées via une API RESTful et surveillées grâce à des outils de monitoring. Ce projet illustre cette approche sur un cas de données de ventes.


À partir d'une base de données de ventes en PostgreSQL, l'objectif est de : **extraire et transformer les données vers MongoDB**, développer une API RESTful pour les exposer, tester cette API avec Postman et assurer un monitoring technique avec Prometheus et Grafana. Un DAG Apache Airflow est également fourni pour décrire le workflow ETL sous forme de tâches.

# Contexte et objectifs

## 2.1 Contexte pédagogique

Le travail est réalisé dans le cadre du module « Développement de base de composants et services web – IBAM 2025 », avec pour but de mettre en pratique la mise en place de services web et l'utilisation d'outils modernes autour des données.

## 2.2 Objectifs du projet

Les principaux objectifs du projet sont :

### ETL

Configurer un processus ETL pour extraire les données de ventes à partir d'une base PostgreSQL, les transformer et les charger dans MongoDB.

### API RESTful

Créer une API RESTful avec Node.js / Express pour exposer les données de ventes.

### CRUD

Développer des endpoints CRUD (Create, Read, Update, Delete) sur les ventes.

### Tests

Tester l'API à l'aide d'un client HTTP (Postman).

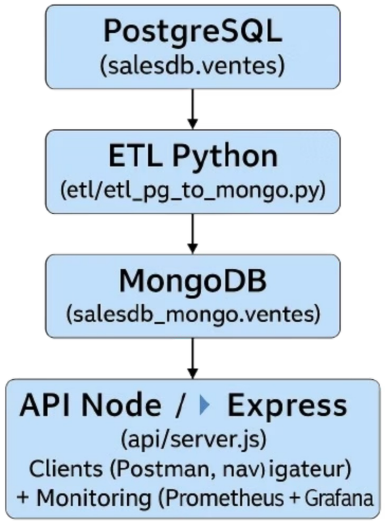
### Monitoring

Mettre en place un monitoring et des dashboards avec Prometheus et Grafana.

Les choix techniques incluent PostgreSQL et MongoDB pour le stockage, Python pour l'ETL, Node.js / Express pour l'API, et Prometheus / Grafana pour le monitoring. Un DAG Airflow est fourni en complément comme configuration de l'ETL.

# Architecture globale et technologies

L'architecture logique du projet peut être résumée ainsi :



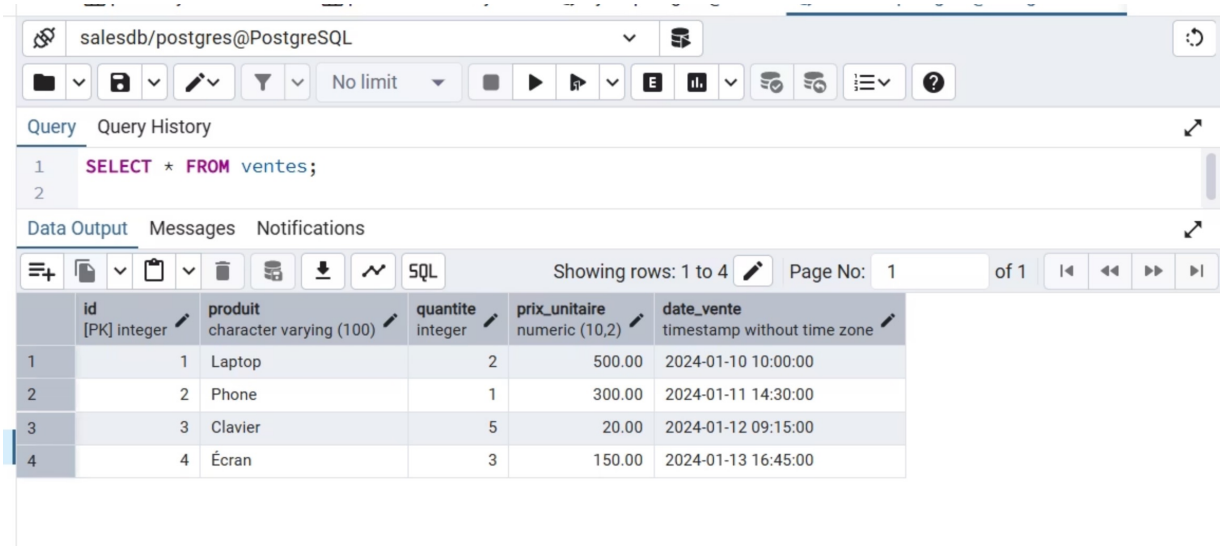

*[Figure 1 : Schéma d'architecture globale du projet]*

## 3.1 Technologies utilisées

- **PostgreSQL** : base de données relationnelle pour la table des ventes.
- **MongoDB** : base documentaire servant de source à l'API REST.
- **Python + psycopg2 + pymongo** : implémentation de l'ETL PostgreSQL → MongoDB.
- **Node.js + Express + Mongoose** : développement de l'API RESTful.
- **Prometheus** : collecte des métriques de l'API exposées sur /metrics.
- **Grafana** : visualisation des métriques dans un dashboard.
- **Airflow (configuration)** : description du workflow ETL sous forme de DAG.

# Préparation des données dans PostgreSQL

Une base PostgreSQL nommée `salesdb` a été créée, puis initialisée à l'aide du script `sql/init_salesdb.sql`. Ce script crée la table ventes et insère quelques lignes de test.



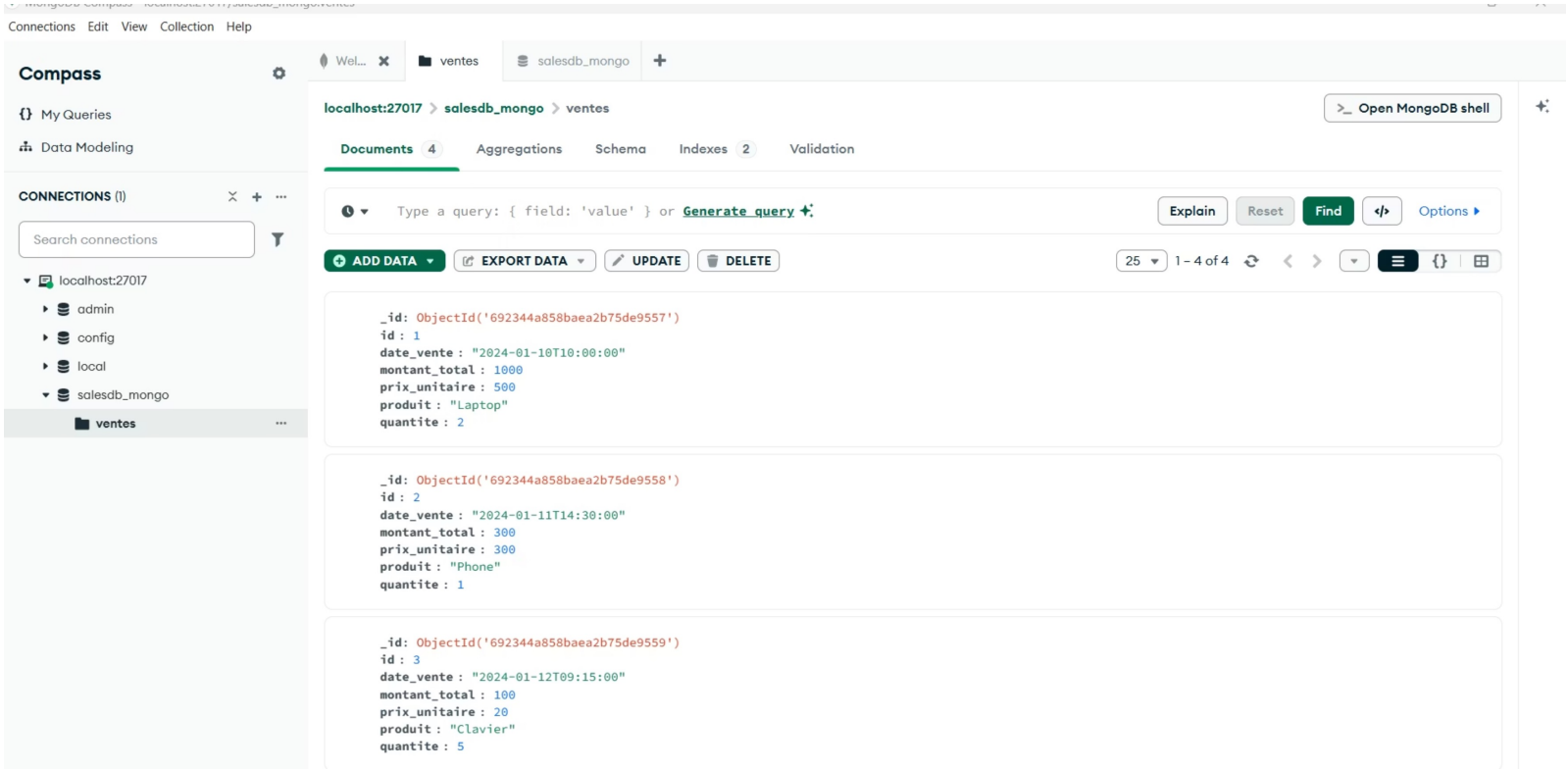

| | id [PK] integer | produit character varying (100) | quantite integer | prix_unitaire numeric (10,2) | date_vente timestamp without time zone |
|---|---|---|---|---|---|
| 1 | 1 | Laptop | 2 | 500.00 | 2024-01-10 10:00:00 |
| 2 | 2 | Phone | 1 | 300.00 | 2024-01-11 14:30:00 |
| 3 | 3 | Clavier | 5 | 20.00 | 2024-01-12 09:15:00 |
| 4 | 4 | Écran | 3 | 150.00 | 2024-01-13 16:45:00 |

[Figure 2 : Capture de la table ventes dans PostgreSQL]

# Processus ETL PostgreSQL → MongoDB

## 5.1 Script ETL Python

Le script `etl/etl_pg_to_mongo.py` implémente les trois étapes ETL : **extraction** des ventes depuis PostgreSQL, **transformation** (calcul du montant_total) et **chargement** des données dans MongoDI

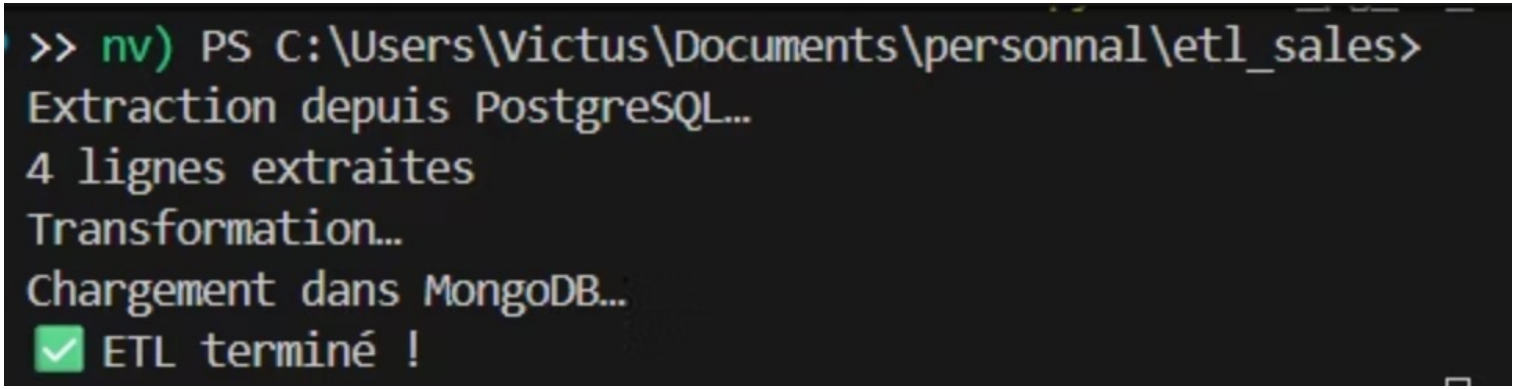



*[Figure 3 : Capture MongoDB Compass montrant la collection salesdb_mongo.ventes avec montant_total.]*

# Processus ETL PostgreSQL → MongoDB

## 5.2 Exécution de l'ETL

L'ETL est exécuté dans un environnement virtuel Python, après installation des dépendances (psycopg2-binary, pymongo).
L'exécution lit les ventes, calcule le montant_total et insère les documents dans MongoDB.




*[Figure 4 : Capture de la console montrant l'exécution réussie du script ETL.]*

# Processus ETL PostgreSQL → MongoDB

## 5.3 Description du DAG Airflow

Le fichier `airflow/etl_ventes_to_mongo_dag.py` décrit le même workflow sous forme de DAG Airflow, avec trois tâches Python : extract, transform et load. Ce DAG pourrait être exécuté dans une instance Airflow pour automatiser l'ETL (par exemple avec @daily).

```python
from datetime import datetime
from airflow import DAG
from airflow.operators.python import PythonOperator

import psycopg2
from pymongo import MongoClient

# ---------- CONFIG POSTGRES ----------
PG_CONFIG = {
    "host": "localhost",
    "port": 5432,
    "database": "salesdb",      # ta base Postgres
    "user": "postgres",         # ton user Postgres
    "password": "Wpharell2000@",  # <-- le même que dans ton script
}

# ---------- CONFIG MONGO ----------
MONGO_URI = "mongodb://localhost:27017"
MONGO_DB = "salesdb_mongo"
MONGO_COLLECTION = "ventes"
```

[Figure 5] : Configiuration de postgress

```python
def extract_from_postgres(**context):
    """Task Airflow : lire les ventes depuis PostgreSQL."""
    conn = psycopg2.connect(**PG_CONFIG)
    cur = conn.cursor()
    cur.execute("""
        SELECT id, produit, quantite, prix_unitaire, date_vente
        FROM ventes
    """)
    rows = cur.fetchall()
    cur.close()
    conn.close()

    ventes = []
    for r in rows:
        vente = {
            "id": r[0],
            "produit": r[1],
            "quantite": r[2],
            "prix_unitaire": float(r[3]),
            "date_vente": r[4].isoformat()
        }
        ventes.append(vente)

    # le return va dans XCom automatiquement
    return ventes
```

[Figure 6] : Extraction des données depuis PostGres

# Processus ETL PostgreSQL → MongoDB

## 5.3 Description du DAG Airflow

Le fichier `airflow/etl_ventes_to_mongo_dag.py` décrit le même workflow sous forme de DAG Airflow, avec trois tâches Python : extract, transform et load. Ce DAG pourrait être exécuté dans une instance Airflow pour automatiser l'ETL (par exemple avec @daily).

```python
def transform_ventes(**context):
    """Task Airflow : ajouter montant_total."""
    ti = context["ti"]
    ventes = ti.xcom_pull(task_ids="extract")

    for v in ventes:
        v["montant_total"] = v["quantite"] * v["prix_unitaire"]

    return ventes


def load_into_mongo(**context):
    """Task Airflow : charger dans MongoDB."""
    ti = context["ti"]
    ventes = ti.xcom_pull(task_ids="transform")

    client = MongoClient(MONGO_URI)
    db = client[MONGO_DB]
    col = db[MONGO_COLLECTION]

    # upsert par id pour éviter les doublons
    for v in ventes:
        col.update_one(
            {"id": v["id"]},
            {"$set": v},
            upsert=True
        )

    client.close()
```

[Figure 7] :

Fonction transform_ventes -> Traitement des données pour les rendre recevable par mongo DB

Fonction load_into_mongo -> Chargement des données dans Mongo DB

```python
# --------- Définition du DAG ---------
with DAG(
    dag_id="etl_ventes_to_mongo",
    start_date=datetime(2024, 1, 1),
    schedule_interval="@daily",   # une fois par jour
    catchup=False,
    tags=["etl", "ventes"],
) as dag:

    extract_task = PythonOperator(
        task_id="extract",
        python_callable=extract_from_postgres,
    )

    transform_task = PythonOperator(
        task_id="transform",
        python_callable=transform_ventes,
    )

    load_task = PythonOperator(
        task_id="load",
        python_callable=load_into_mongo,
    )

    # Ordre des tâches
    extract_task >> transform_task >> load_task
```

[Figure 6] : Définition du DAG

# API RESTful de gestion des ventes

## 6.1 Structure du projet API

Le projet Node.js / Express est organisé avec un fichier principal `server.js`, un modèle Mongoose Sale et des routes regroupées dans `routes/salesRoutes.js`.
L'API se connecte à MongoDB (salesdb_mongo, collection ventes).

## 6.2 Endpoints CRUD

Les endpoints principaux sont :

### GET /

vérifie que l'API est en ligne.

### GET /ventes

retourne la liste de toutes les ventes.

### GET /ventes/:id

détail d'une vente spécifique.

### POST /ventes

création d'une nouvelle vente.

### PUT /ventes/:id

mise à jour d'une vente existante.

### DELETE /ventes/:id

suppression d'une vente.

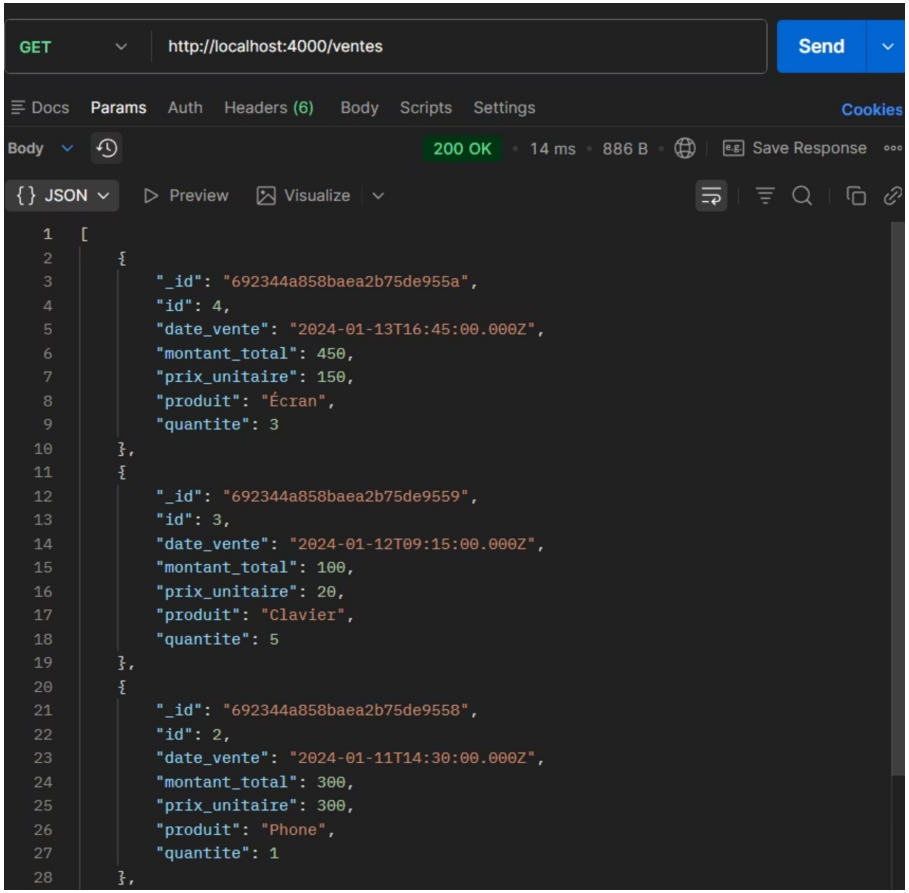
# API RESTful de gestion des ventes

## 6.1 Structure du projet API

Le projet Node.js / Express est organisé avec un fichier principal `server.js`, un modèle Mongoose Sale et des routes regroupées dans `routes/salesRoutes.js`.
L'API se connecte à MongoDB (salesdb_mongo, collection ventes).

## 6.3 Tests avec Postman

Les endpoints ont été testés à l'aide de Postman. Par exemple, GET /ventes retourne un tableau de ventes, tandis qu'un POST /ventes crée une nouvelle vente à partir d'un JSON contenant produit, quantite, prix_unitaire et date_vente.




[Figure 6 : Capture Postman – réponse de GET /ventes.]

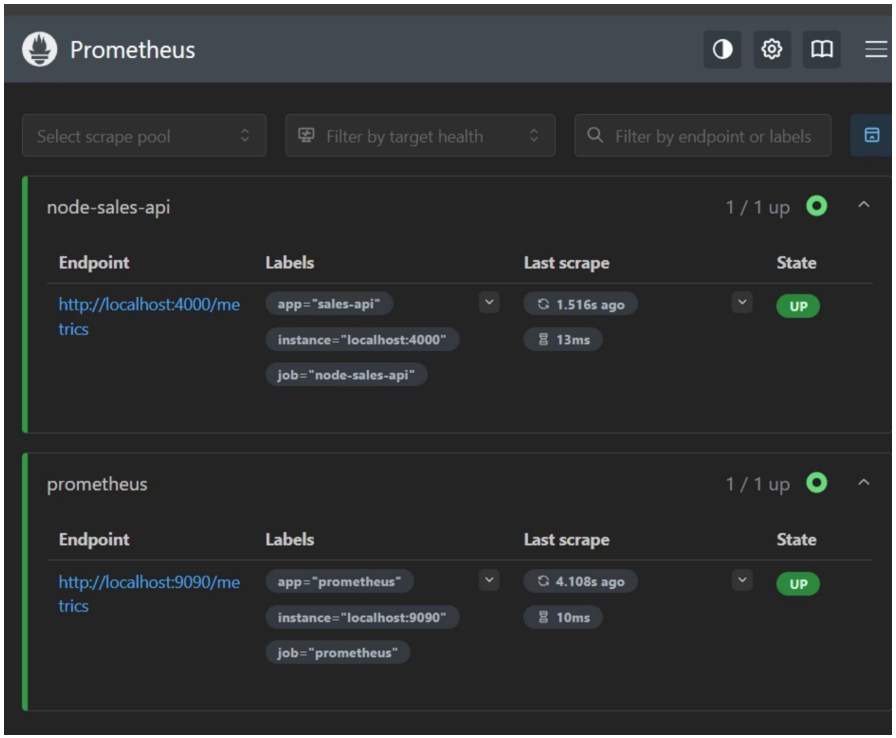
# Monitoring avec Prometheus et Grafana

## 7.1 Exposition des métriques

Un middleware `express-prom-bundle` a été ajouté à l'API pour exposer des métriques au format Prometheus sur `/metrics`. Ces métriques incluent notamment les compteurs et histogrammes de requêtes HTTP, ainsi que l'utilisation mémoire et CPU du processus Node.

## 7.2 Prometheus

Prometheus est configuré via `monitoring/prometheus.yml` avec un job `node-sales-api` qui scrute les métriques de l'API sur localhost:4000.
L'interface de Prometheus permet de vérifier que le job est bien UP.




*[Figure 7 : Capture Prometheus – page Targets montrant node-sales-api en UP.]*

# Monitoring avec Prometheus et Grafana

## 7.1 Exposition des métriques

Un middleware `express-prom-bundle` a été ajouté à l'API pour exposer des métriques au format Prometheus sur `/metrics`. Ces métriques incluent notamment les compteurs et histogrammes de requêtes HTTP, ainsi que l'utilisation mémoire et CPU du processus Node.

## 7.3 Dashboard Grafana

Grafana utilise Prometheus comme datasource et un dashboard dédié a été créé pour visualiser les métriques de l'API. Les panels principaux affichent : **le nombre de requêtes par méthode et code HTTP**, le taux de requêtes par seconde et la mémoire utilisée par le processus Node.


[Figure 8 : Capture du dashboard Grafana complet (3 panels principales).]

# Discussion et conclusion

Le projet atteint les objectifs fixés : les données sont stockées dans PostgreSQL, transformées et chargées dans MongoDB via un ETL Python, exposées par une API RESTful Node.js / Express et monitorées avec Prometheus et Grafana. Un DAG Airflow complète la solution en décrivant le workflow ETL sous forme de tâches planifiables.

## Améliorations possibles

### API enrichie

Ajout de filtres et de pagination sur l'API

### Sécurité

Mise en place d'une authentification

### Monitoring avancé

Enrichissement du dashboard Grafana avec des métriques supplémentaires

### Automatisation

Exécution réelle du DAG Airflow dans un environnement Linux ou Docker

> Ce projet illustre ainsi une petite architecture de données proche d'un cas réel, combinant stockage relationnel, base NoSQL, services web, ETL et observabilité.