**MINISTERE DE L’ENSEIGNEMENT SUPERIEUR ET DE LA**

**RECHERCHE SCIENTIFIQUE ET DE L’INNOVATION**

(**M.E.S.R.S.I**)

------------------

**S**ecrétariat **G**énéral

-------------------------

**U**niversité **N**azi **B**ONI (**U.N.B**.)

-------------------------------

**É**cole **S**upérieure d’**I**nformatique (**E.S.I**)



**Master en Science de Données** **(M.S.D**)

**Option** : **S**cience de **D**onnées (**SD**)

**PROJET DE MACHINE LEARNING ET BIG DATA**

**THEME :**

**Création d’un pipe basé sur l’architecture**

d’un

**KAPPA**



|  |  |
| --- | --- |
| **Membres du Groupe :**  **KABRE Hamidou**  **LOMPO Abel**  **KIETENGA Laurant**  **TRAORE Oumar** | **ENSEIGNANT :**    **DRABO Constantin / Pr Sadouamouan MALO** |

**SOMMAIRE**

# 

[Introduction 3](#_Toc200290159)

[I. Architecture KAPPA 4](#_Toc200290160)

[II. Outils et technologies utilisées 10](#_Toc200290161)

[III. Implémentation de la Pipeline 12](#_Toc200290162)

[IV. Modèle de Machine Learning pour la prédiction 16](#_Toc200290163)

[V. Présentation des Résultats 25](#_Toc200290164)

[VI. Démonstration du système 28](#_Toc200290165)

[Conclusion 32](#_Toc200290166)

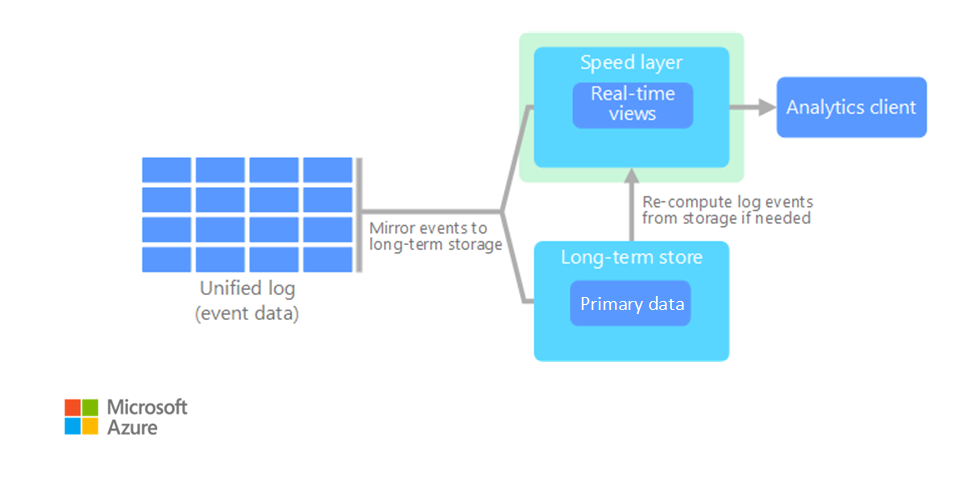
# Introduction

L’architecture Kappa est une approche moderne de traitement de données en continu. Elle repose sur une simplification du modèle Lambda, en supprimant la dualité batch/stream pour se concentrer uniquement sur les flux temps réel. Ce mémoire présente une mise en œuvre concrète de cette architecture appliquée à l’ingestion et à la restitution de données financières à l’aide des technologies Spring Boot, Apache Kafka, Polygon.io et Elasticsearch.

# Architecture KAPPA

1. Présentation de l’architecture Kappa

**L’architecture Kappa** est une alternative à l’architecture Lambda. C’est un modèle de traitement de données massives en temps réel, proposé par **Jay Kreps (co-fondateur de Kafka)**. Il a les mêmes objectifs de base que l’architecture Lambda, mais toutes les données transitent par un chemin unique via un système de traitement de flux. Aussi, contrairement à l’architecture Lambda qui combine un traitement en **batch** et un traitement en **Stream**, l’architecture Kappa repose exclusivement sur un **pipeline de streaming**.



*Figure 1 : Architecture Kappa*

1. Les différentes étapes de l’architecture kappa

2.1. La couche d’ingestion

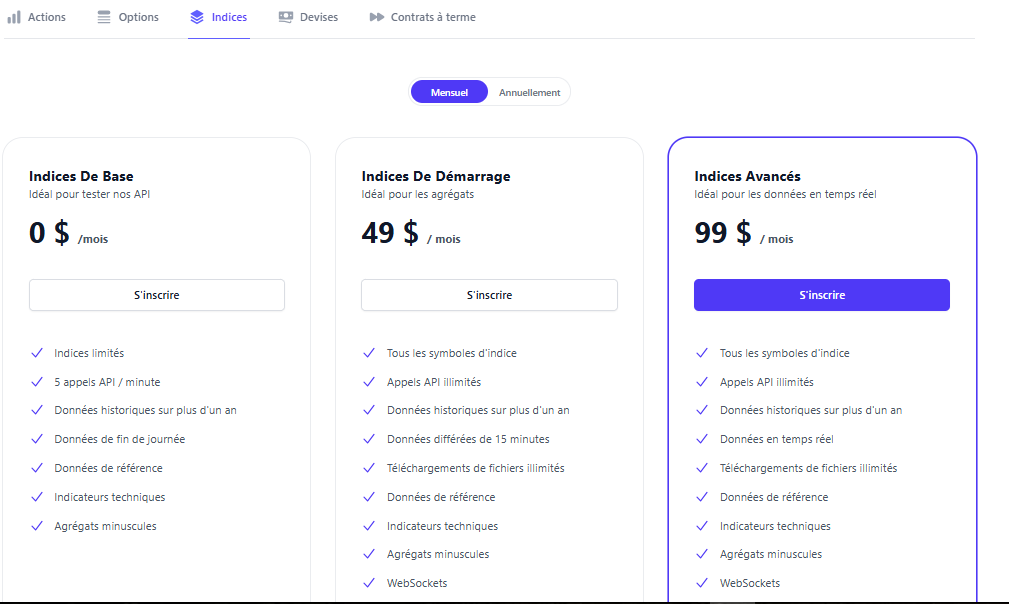
La couche d’ingestion représente la première étape de l’architecture Kappa. Son rôle principal est de récupérer les données brutes depuis les sources externes, puis de les transformer en flux exploitables en temps réel. Ces flux sont ensuite envoyés dans un système de messagerie Apache Kafka dans notre implémentation, où ils deviennent disponibles pour les couches suivantes.

Cette couche joue un rôle critique dans la qualité et la disponibilité des données en assurant une collecte fiable, continue et évolutive.

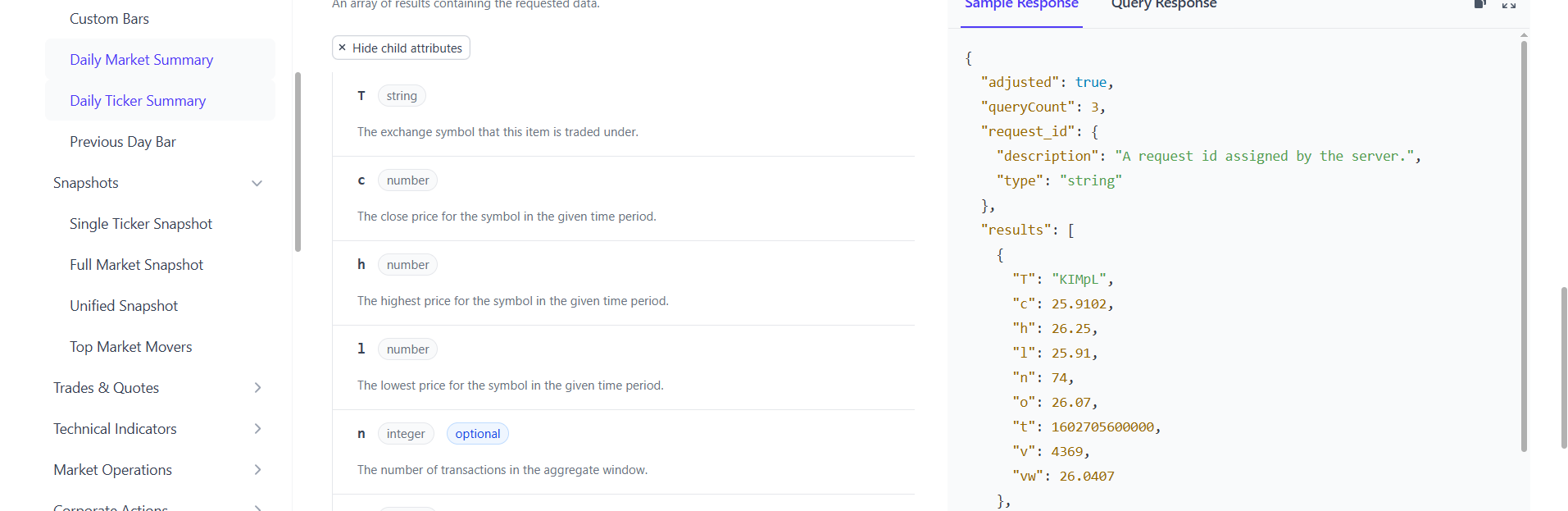
2.2. Les sources de données

Polygon.io est une plateforme d’API financières permettant d’accéder à une grande variété de données de marché telles que les prix des actions, les volumes échangés, et d’autres agrégats boursiers. Elle s’adresse principalement aux développeurs et analystes souhaitant intégrer des données financières dans des outils d’analyse, des tableaux de bord ou des modèles de machine learning.

Dans le cadre d’un compte gratuit (Free Tier), Polygon.io permet de consulter des données boursières américaines avec un décalage de 15 minutes. Ce niveau d’abonnement autorise l’utilisation d’endpoints essentiels tels que “/aggs/grouped/locale/us/market/stocks/{date}” que nous avons utiliser dans notre modèle.Cet endpoint offre un accès groupé aux données agrégées de toutes les actions cotées aux États-Unis pour une date précise.Aussi Ce point d’accès retourne pour chaque titre les informations de prix (open, high, low, close), le volume, le nombre de transactions, et le prix moyen pondéré. Ces données peuvent servir de base pour constituer des jeux de données destinés à l’analyse ou à des projets de prédiction via apprentissage automatique. Dans notre projet pour simuler la collecte des données sur le temps nous avons boucler les appels Api à partir du parametre de la date à la date présente afin de récolter toute les informations necessaire.



*Figure 2 : Choix du type de compte ou d’abonnement*



*Figure 3 : Résumé quotidien du marché*

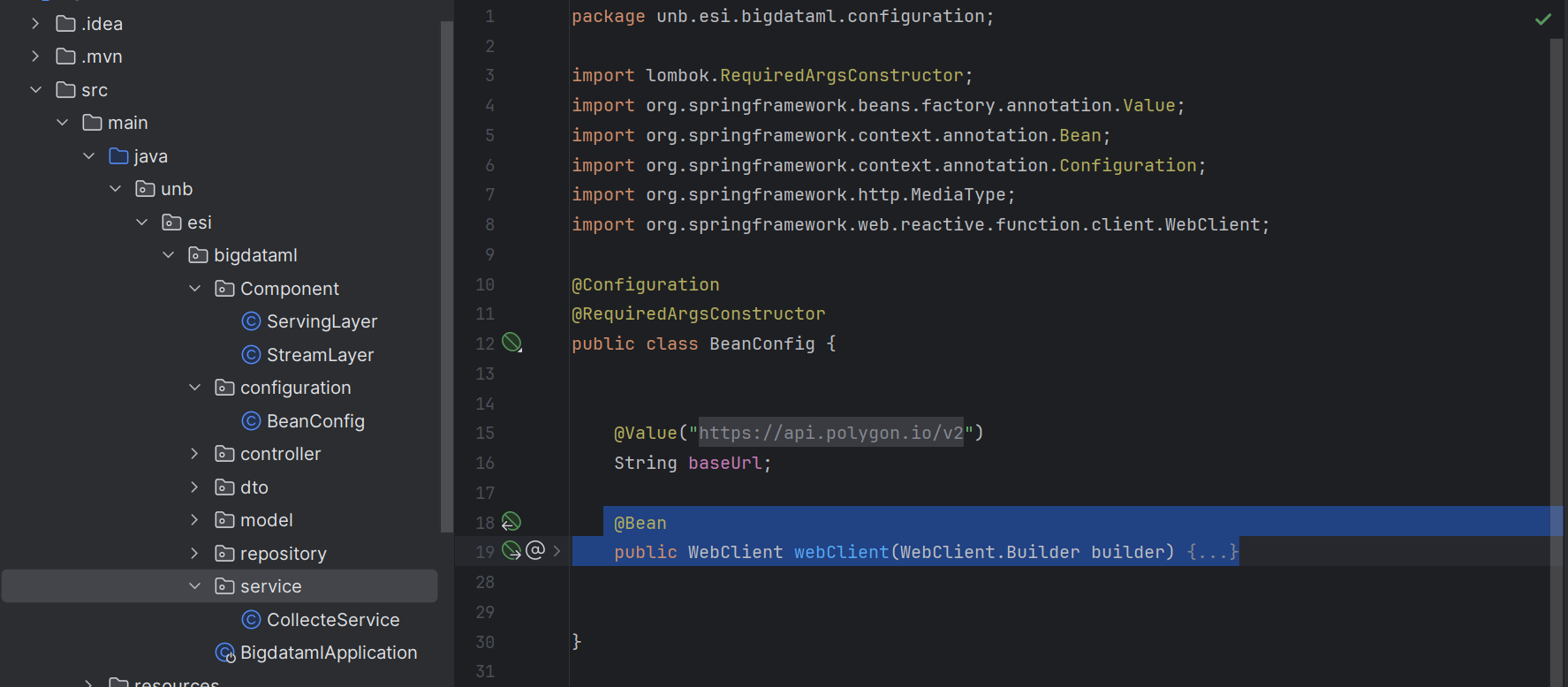
2.3. Le Service de collecte

La récupération des données depuis l’API dans notre projet est assurée par une Classe service appelé CollecteService. Ce service encapsule la logique d’appel à l’API distante, effectue les conversions nécessaires vers des objets métier de type record en java (ApiResponse), et assure la robustesse du processus de collecte (par exemple, en gérant les erreurs ou les appels nuls).

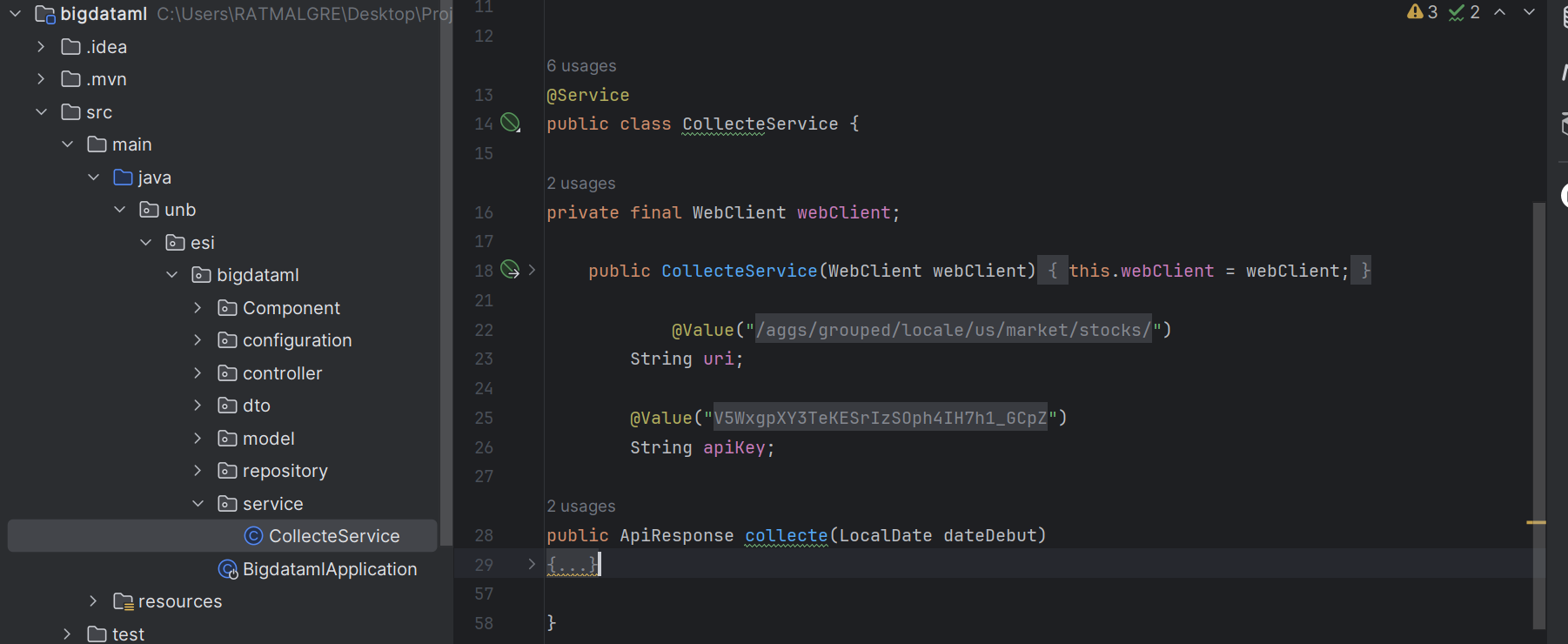
Le service de collecte est exécuté de manière régulière grâce à une tâche planifiée configurée avec l’annotation @Scheduled dans le composant **StreamLayer**, garantissant ainsi une collecte automatisée et continue toutes les 12 secondes. Cette tâche démarre à partir d’une date de début définie dans le fichier de configuration (properties).

À chaque exécution, le service récupère les données correspondant à une nouvelle journée, comprise entre la date de début et la date courante. Il est important de noter que le compte gratuit utilisé pour accéder à l’API Polygon.io limite la récupération des données aux 2 dernières années, excluant les données du jour en cours.

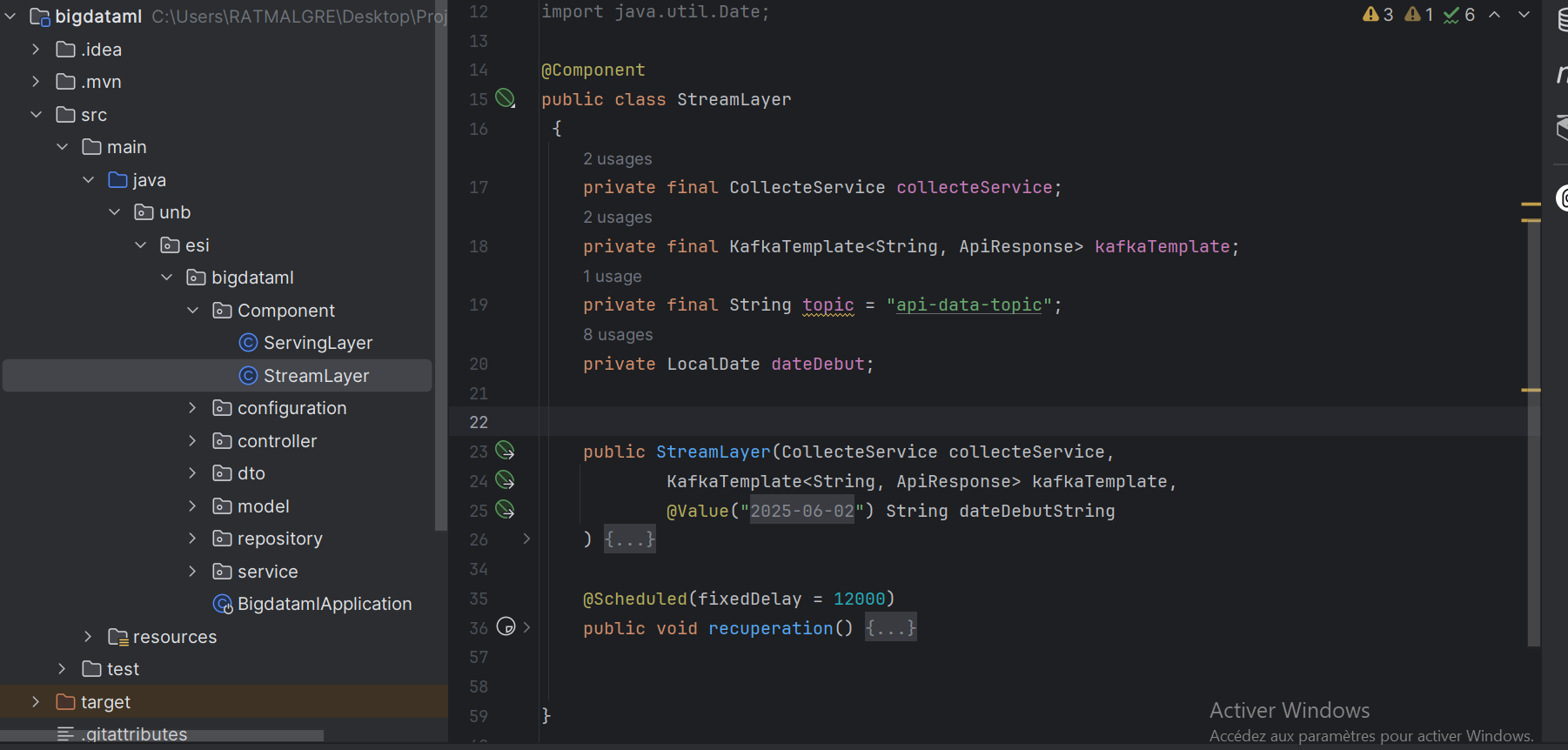
Pour effectuer les appels API, nous avons utilisé **WebClient** de Spring WebFlux, qui permet une communication non bloquante et performante avec le service distant. La configuration des en-têtes nécessaires à ces appels est centralisée dans un bean spécifique nommé **BeanConfig**, assurant ainsi une gestion claire et réutilisable des paramètres d’authentification et des autres headers requis.



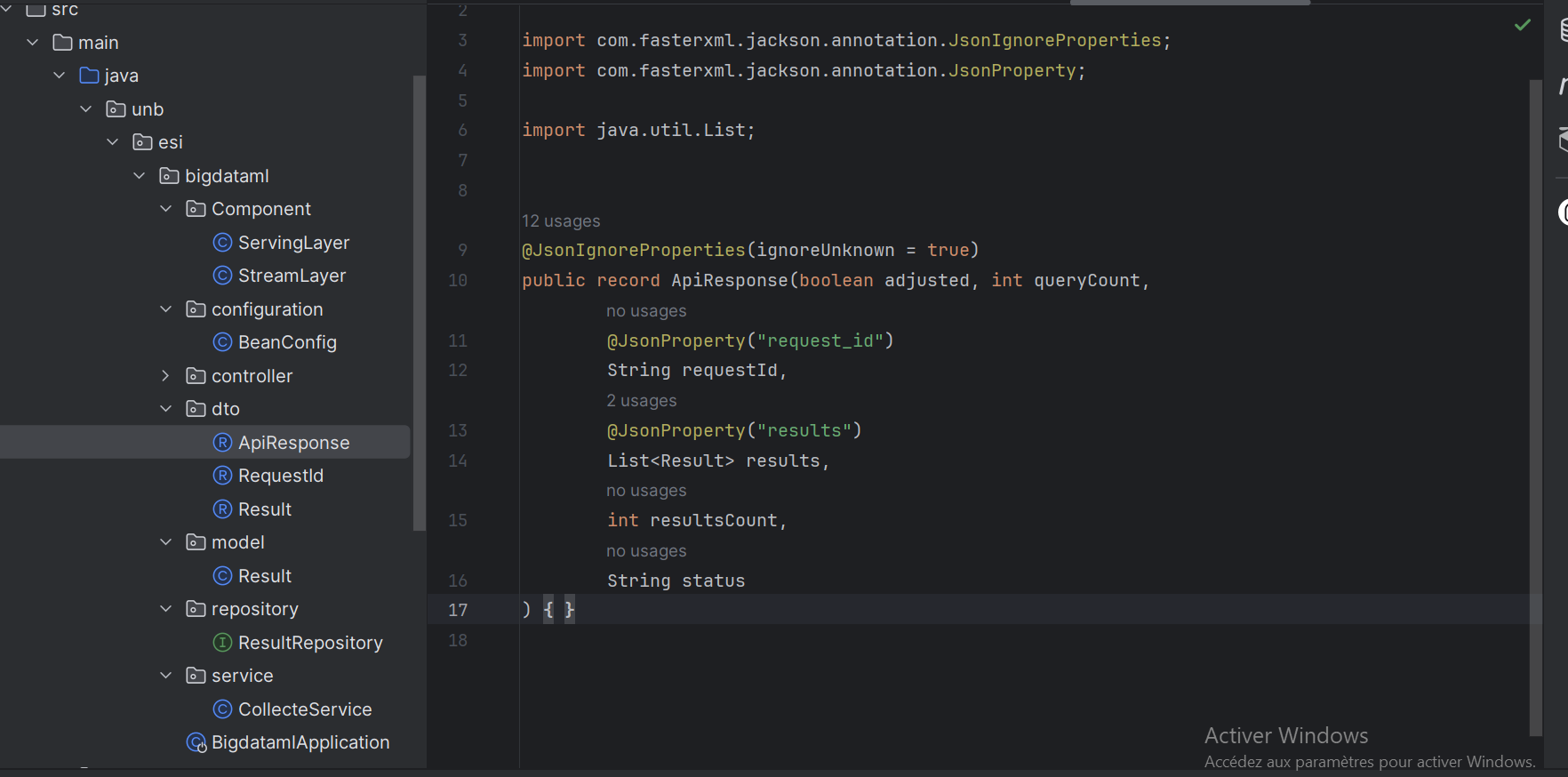
*Figure 4 : Appel de l’API avec WebClient*



*Figure 5 : Service de collecte*



*Figure 6 : Le StreamLayer*



*Figure 7 : L’API response*

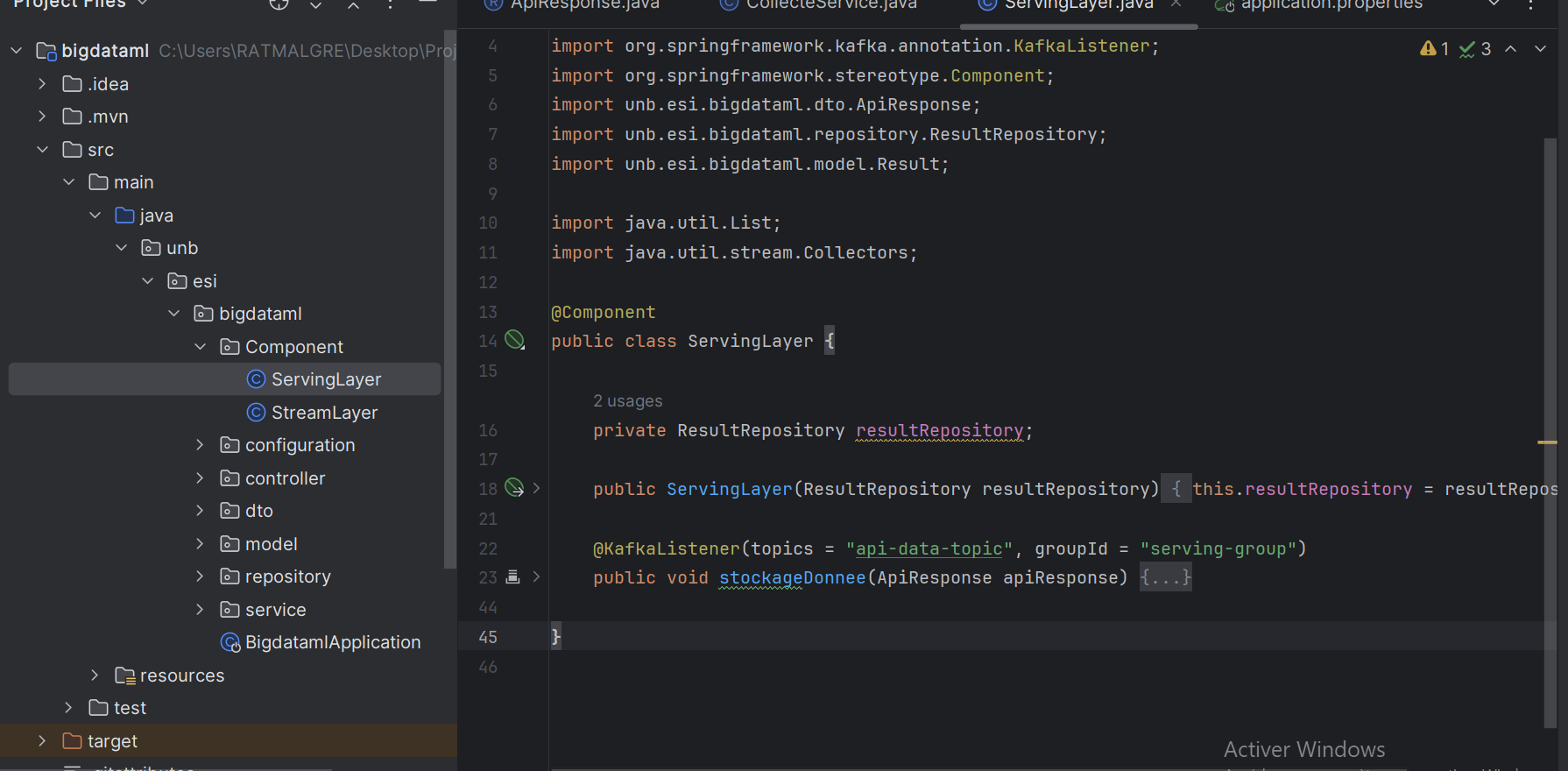
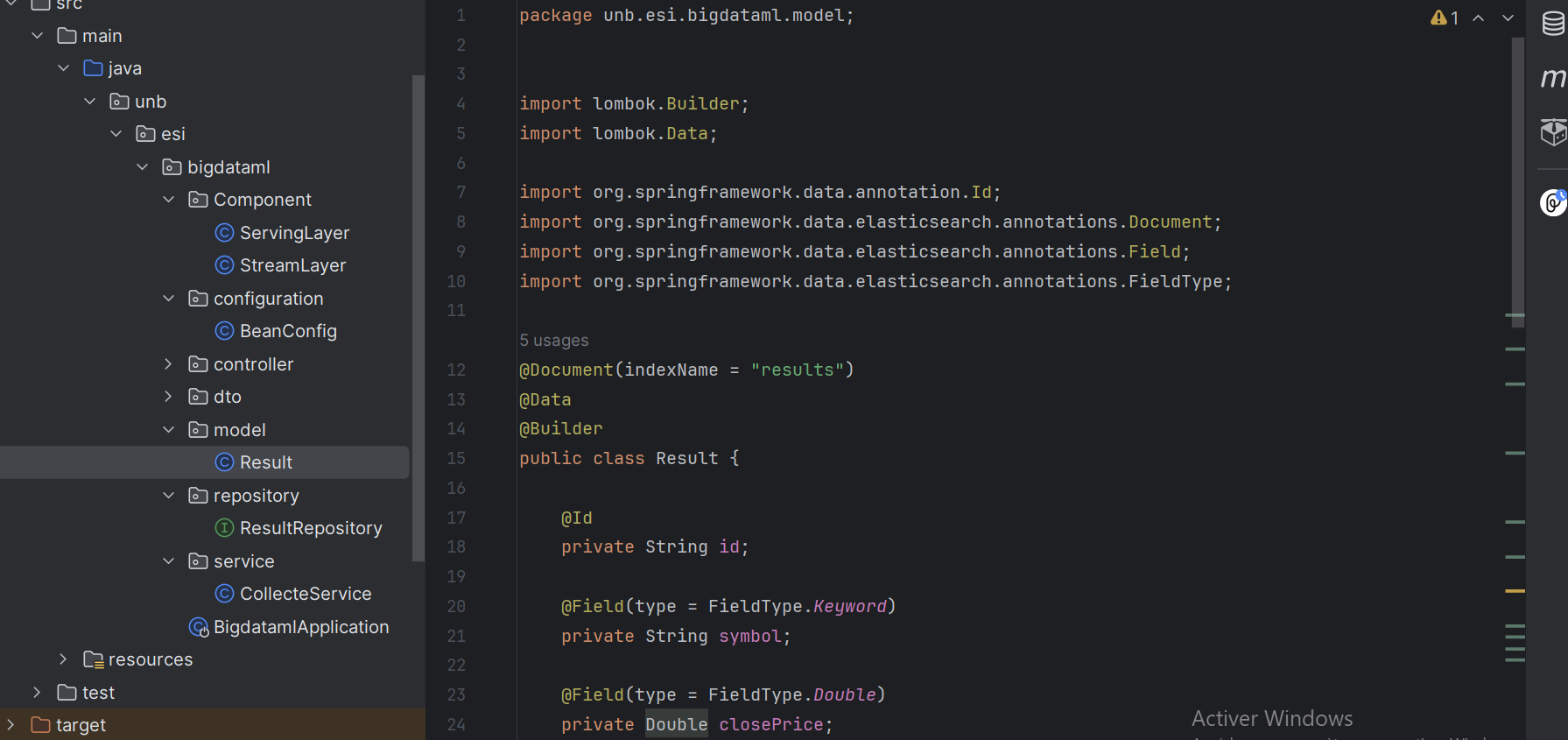
2.4. Envoi des données dans Kafka (topic)

Une fois les données récupérées via CollecteService, elles sont transmises à Apache Kafka via un producteur Kafka (KafkaTemplate) dont les configurations sont dans properties. Les données sont publiées dans un topic Kafka nommé “api-data-topic”, servant de canal de diffusion pour l’ensemble du système. La classe Component StreamLayer dans notre projet est la classe qui s’occupe de ses aspects.

2.5. La couche de traitement (Serving Layer)

Dans le cadre de l’architecture Kappa, la Serving Layer, ou couche de restitution, représente la dernière étape du pipeline de traitement. Elle a pour rôle d’ingérer les flux de données issus d’Apache Kafka, sans transformation majeure, pour les stocker dans un moteur d’indexation et de recherche, ici Elasticsearch. Cette couche permet ainsi de rendre les données brutes ou faiblement transformées immédiatement consultables,

Dans notre projet, cette couche est incarnée par la classe ServingLayer, un composant Spring annoté avec @Component et @KafkaListener, ce qui lui permet d’écouter continuellement les événements diffusés dans le topic Kafka nommé “api-data-topic”. Lorsqu’un message est reçu, il est désérialisé en un objet métier de type ApiResponse. Les enregistrements contenus dans la réponse (results) sont ensuite extraits, transformés en instances de la classe Result, puis persistés dans l’index Elasticsearch results grâce à l’interface ResultRepository.  
Cette implémentation répond à plusieurs objectifs fonctionnels : consommer les flux issus de Kafka, extraire les informations pertinentes des messages (symboles boursiers, prix, volumes, etc.), les mapper en entités persistables, puis les stocker de manière efficace pour permettre un accès rapide et performant aux données. Les propriétes du consumer sont definie dans properties



*Figure 8 : ServingLayer*

# Outils et technologies utilisées

Dans ce projet, nous avons utilisé un ensemble cohérent de technologies issues de l’écosystème du **Big Data** et du **Machine Learning,** dans une approche temps réel conforme à **l’architecture Kappa**.

* **Polygon.io**

C’est un fournisseur de flux de données boursières (API financière en temps réel) accessible en API REST (ou WebSocket pour le streaming). Les données financières sont récupérées en temps et utilisées pour l’apprentissage.

* **Spring Boot** **et** **Python**

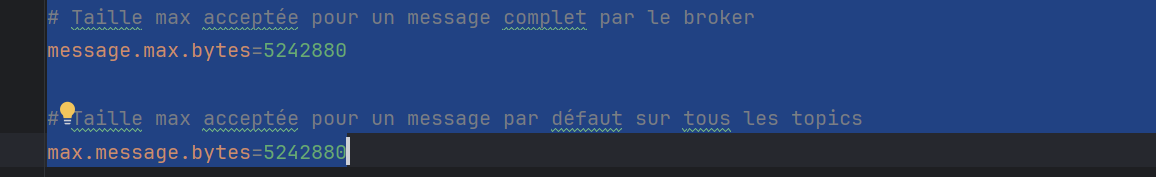
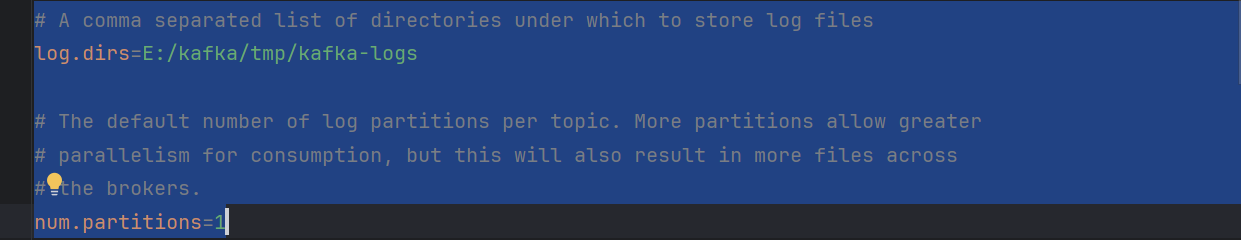
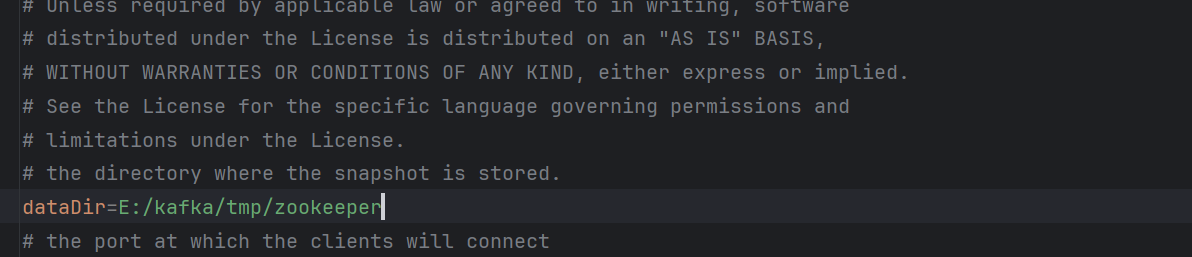
Ce sont les langages de programmation utilisés dans ce projet. Le Framework Spring boot est utilisé pour l’implémentation tandis que python est utilisé pour la modélisation du modèle de machine learning.

* **Installation de Apache Kafka et exécution en local**

Apache Kafka est un système de messagerie distribué conçu pour gérer des flux de données en temps réel. Il joue un rôle central dans l’architecture Kappa en assurant la communication entre la couche d’ingestion et la couche de restitution.

Pour installer Apache Kafka en local sur Windows, il faut tout d’abord se rendre sur le site officiel à l’adresse https://kafka.apache.org/downloads afin de télécharger la dernière version stable. Après extraction de l’archive dans un répertoire dédié, il convient de configurer les fichiers zookeeper.properties et server.properties situés dans le dossier config selon les besoins spécifiques du système.

Une fois la configuration terminée, ouvrez une invite de commandes en mode administrateur, puis positionnez-vous dans le dossier d’installation de Kafka. Lancez ensuite le service ZooKeeper en exécutant la commande suivante : “.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties” Dans une nouvelle fenêtre de terminal, lancez le serveur Kafka avec la commande : : ” .\bin\windows\kafka-server-start.bat .\config\server.properties”

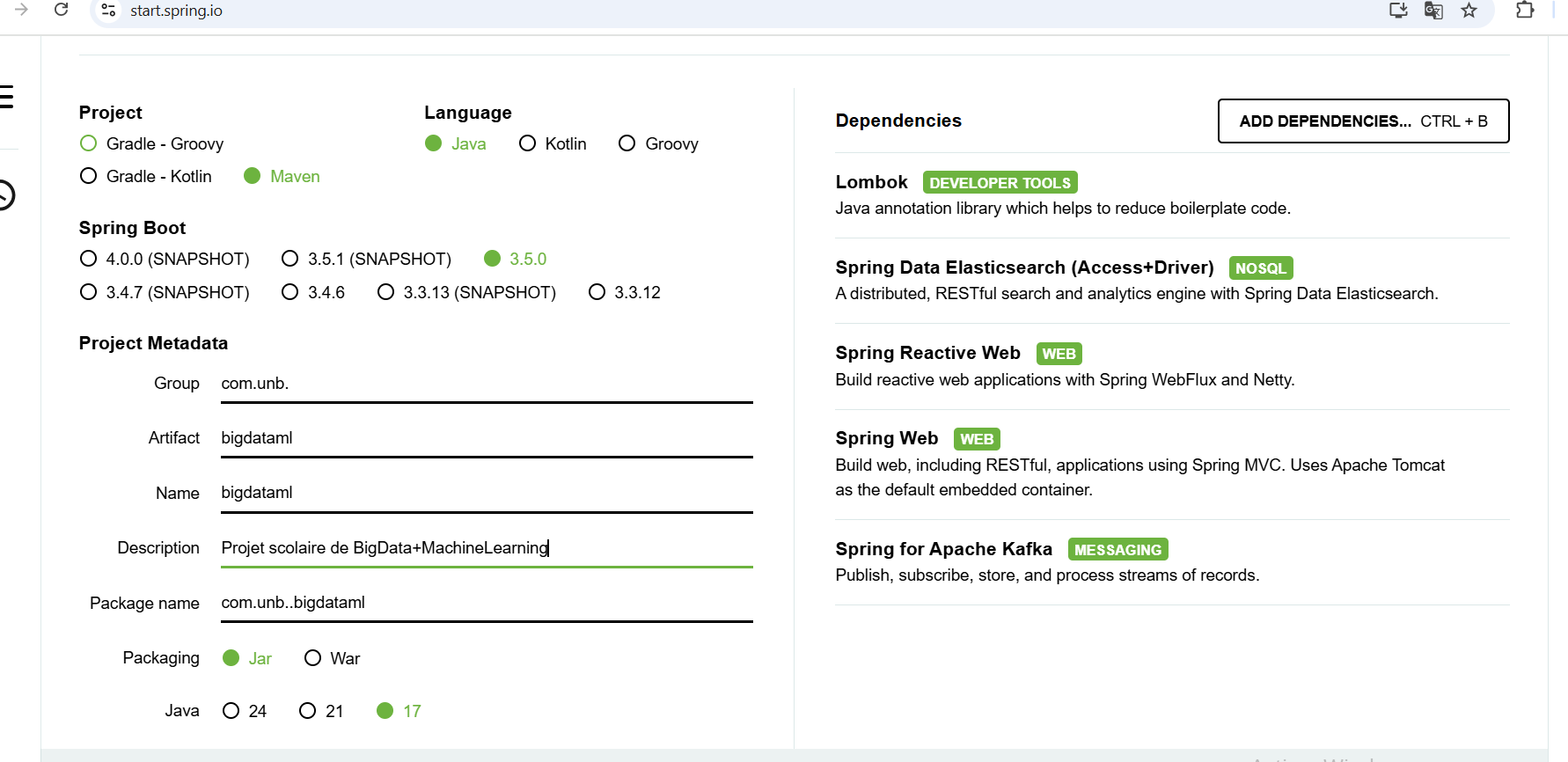


*Figure 9 : Exécution local de Kafka*

* **Elasticsearch installer en local**

# Implémentation de la Pipeline

La création du projet a débuté par l’utilisation de **Spring Initializr**, un outil en ligne accessible à l’adresse <https://start.spring.io>, qui permet de générer rapidement la structure d’un projet Spring Boot personnalisé. L’interface intuitive de l’outil a permis de configurer les principales caractéristiques du projet.



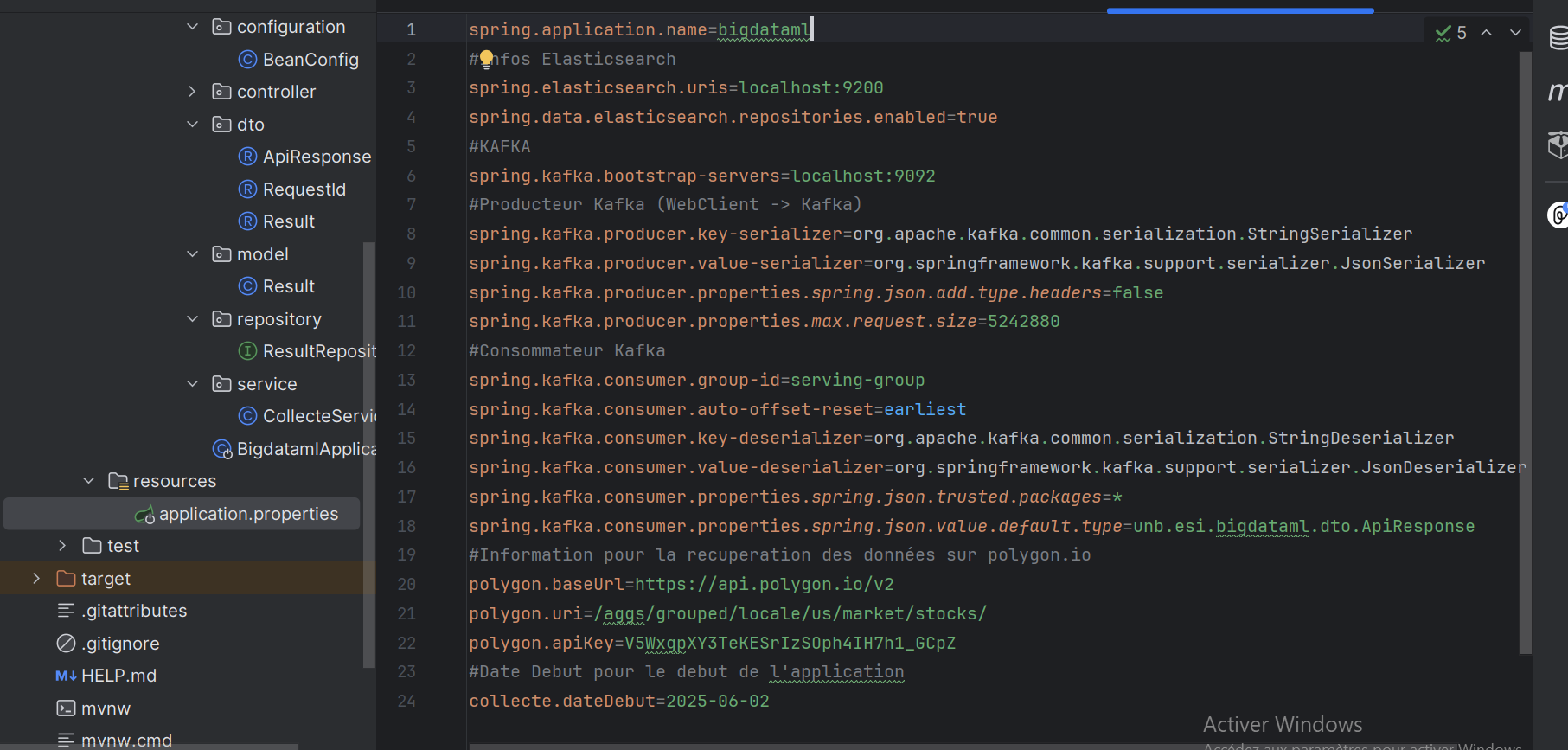
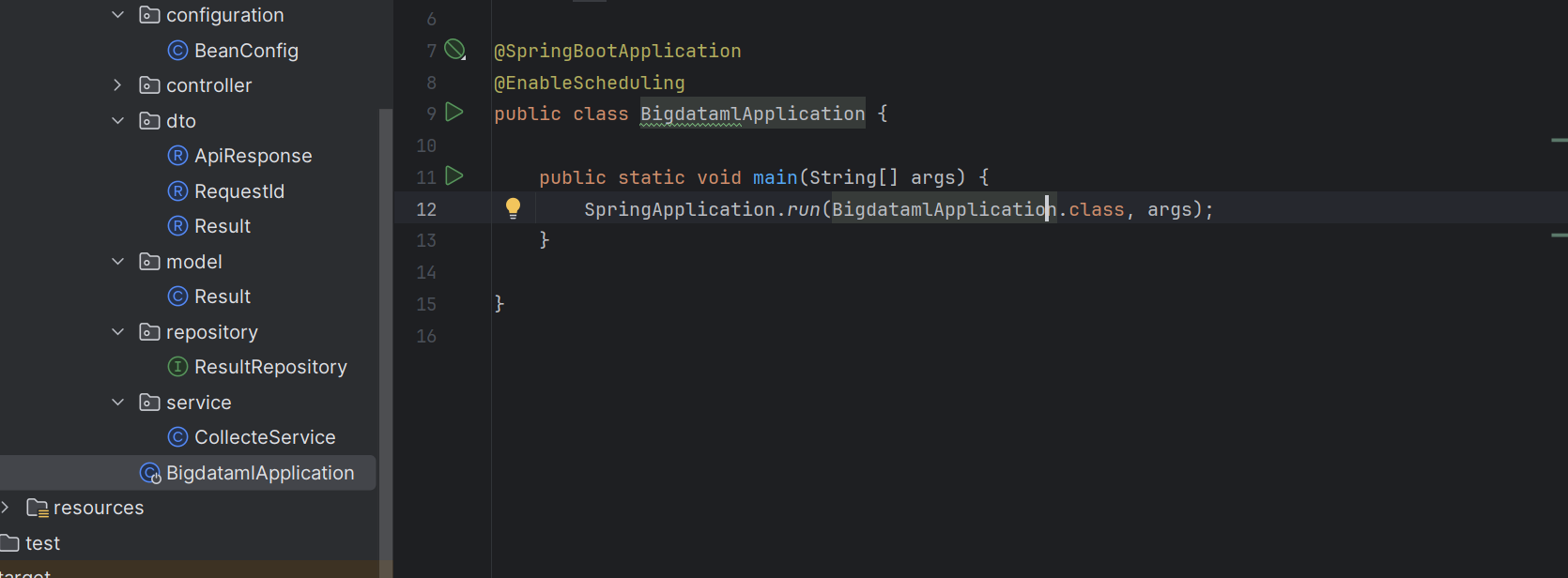
*Figure 10 : Création et configuration*

Nous avons opté pour **Maven** comme système de gestion de projet, en sélectionnant le langage **Java** et une version stable de **Spring Boot** (par exemple, la 3.2.x). Les métadonnées du projet ont ensuite été définies, avec le groupId com.monprojet, l’artifact streaming-app, et le nom de package com.monprojet.streaming. Le projet a été configuré pour générer un **fichier JAR** exécutable, compatible avec la version **Java 17**, assurant ainsi une bonne prise en charge des dernières fonctionnalités du framework.

Concernant les dépendances, plusieurs modules essentiels ont été intégrés dès la génération du projet. **Spring Web** a été ajouté afin de permettre la création de services REST pour faire un test avec Postman. et l’utilisation du client WebClient pour les appels API. **Spring Kafka** a été inclus pour la gestion des flux de données avec Apache Kafka, tandis que **Spring Data Elasticsearch** a été utilisé pour l’indexation et la recherche des données dans un moteur Elasticsearch. D’autres dépendances utilitaires comme **Lombok** (pour la génération automatique de code) et **Spring Boot DevTools** (pour accélérer le développement grâce au rechargement à chaud) ont également été sélectionnées.

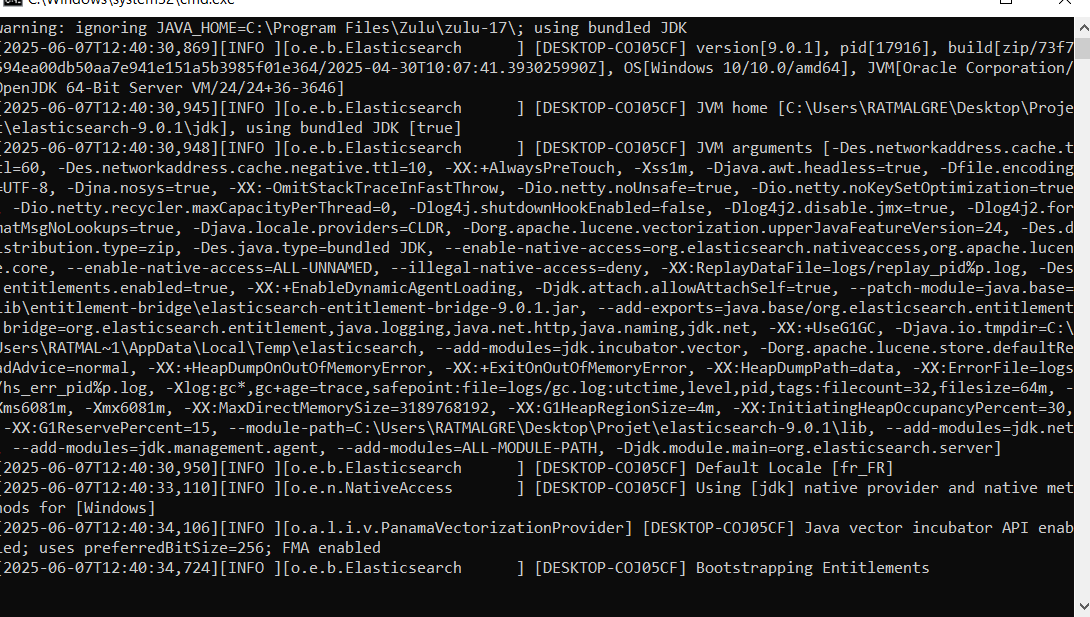
Une fois tous ces paramètres définis, le projet a été généré via le bouton "Generate", ce qui a entraîné le téléchargement d’une archive .zip contenant l’ensemble de la structure du projet. Cette archive a ensuite été extraite et importée dans un environnement de développement intégré **IntelliJ IDEA**. À l’ouverture du projet, **Maven** a automatiquement résolu et téléchargé l’ensemble des dépendances spécifiées dans le fichier pom.xml.

Le projet était alors prêt à être lancé. La classe principale générée, annotée avec @SpringBootApplication, contient une méthode main() permettant de démarrer l’application localement en tant que serveur Spring Boot. Ce point d’entrée offre la base pour construire les composants métiers de notre système de traitement de données en temps réel.

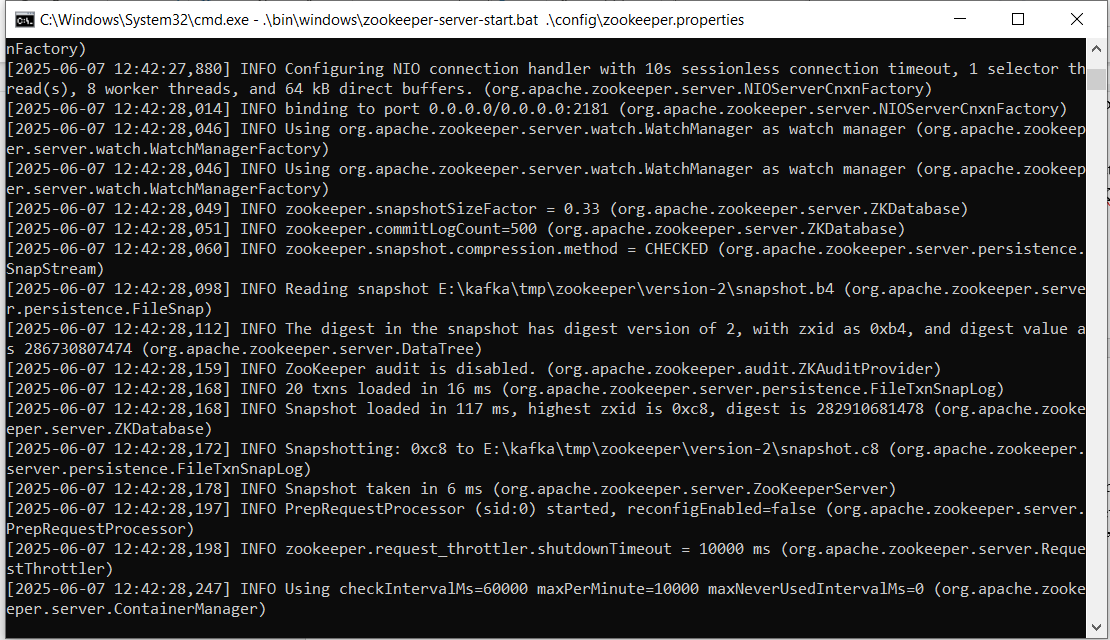


*Figure 11 : Structures du projet*

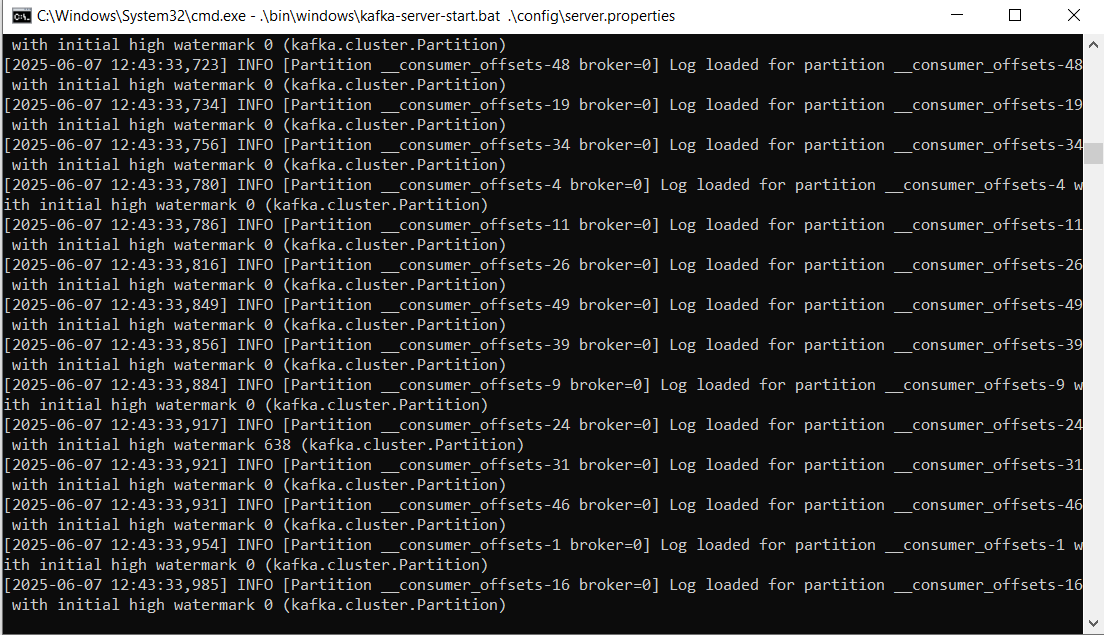
Plusieurs étapes complémentaires ont été mises en œuvre, comme l’illustrent les images ci-dessous :



*Figure 12 : Lancement du serveur Elasticsearch*



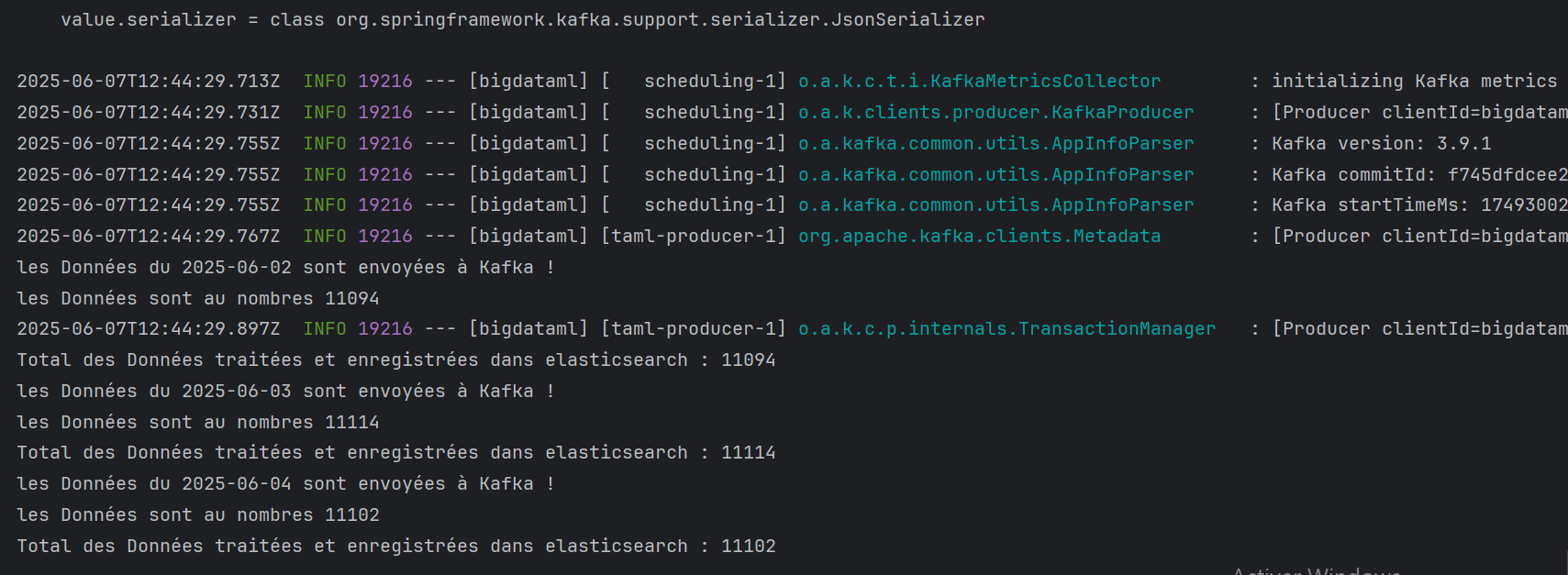
*Figure 13 : Lancement de Kafka : ZooKeeper*



*Figure 14 : Lancement du serveur Kafka*



*Figure 15 : Lancement de Spring Boot*



*Figure 16 : Résultat Invite de Commande*

# Modèle de Machine Learning pour la prédiction

L’objectif principal du modèle de machine learning est de prédire le **prix de clôture (closePrice)** d’un actif financier en fonction du **volume de transactions** observé. Cette tâche de régression permet d’anticiper la valeur d’un actif à partir d’une donnée continue, ce qui est particulièrement utile dans un contexte de surveillance en temps réel des marchés.

1. **Préparation des données**

* Source des Données

Les données financières utilisées dans ce projet sont collectées à partir de la plateforme Polygon.io. Ces données sont ensuite transmises dans ElasticSearch, via un pipeline Kappa (Stream + serving layer).

* **Pipeline de collecte**

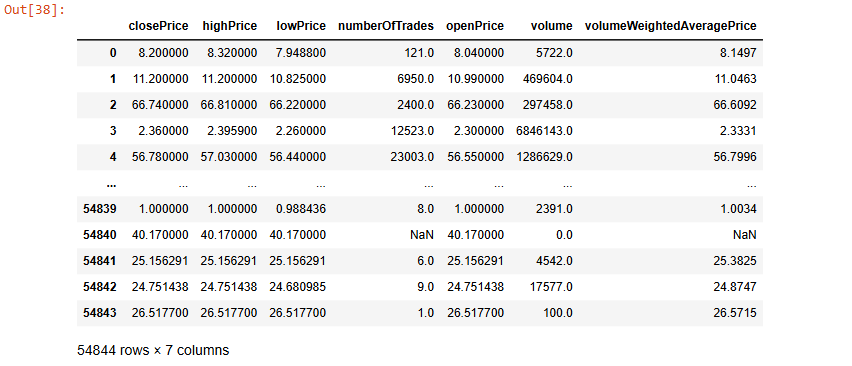
- **Stream Layer :** collecteur de données en temps réel via API Polygon.io

- **Serving Layer:** stockage dans un index Elasticsearch nommé "results"

- **Langage utilisé :** Python (utilisation de la bibliothèque elasticsearch)

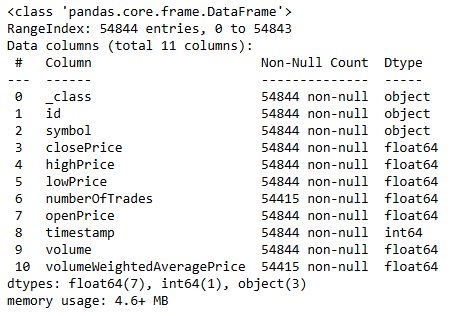
* **Description des variables**

|  |  |
| --- | --- |
| **Champ** | **Description** |
| **symbol** | Symbole boursier |
| **closePrice** | Prix de clôture |
| **highPrice** | Prix le plus haut atteint durant la journée |
| **lowPrice** | Prix le plus bas de la journée |
| **openPrice** | Prix d'ouverture |
| **volume** | Volume échangé |
| **numberOfTrades** | Nombre de transactions |
| **volumeWeightedAveragePrice** | Prix moyen pondéré par le volume |
| **timestamp** | Timestamp Unix (millisecondes) |

****

*Figure 17 : Visualisation du jeu de donné*

* **Analyse initiale**

****

*Figure 18 : Analyse des différentes variables*

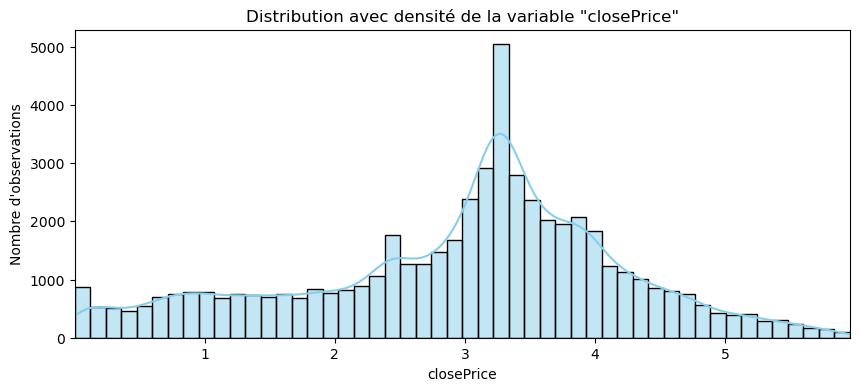
Les données sont numériquement cohérentes, avec des types bien définis (float64, int64, object). Deux variables (numberOfTrades et volumeWeightedAveragePrice) présentent 429 valeurs manquantes. Il conviendra de traiter ces valeurs lors de la phase de préparation des données (par imputation ou suppression). Les autres colonnes sont complètes, ce qui est favorable à l’analyse statistique et la modélisation.

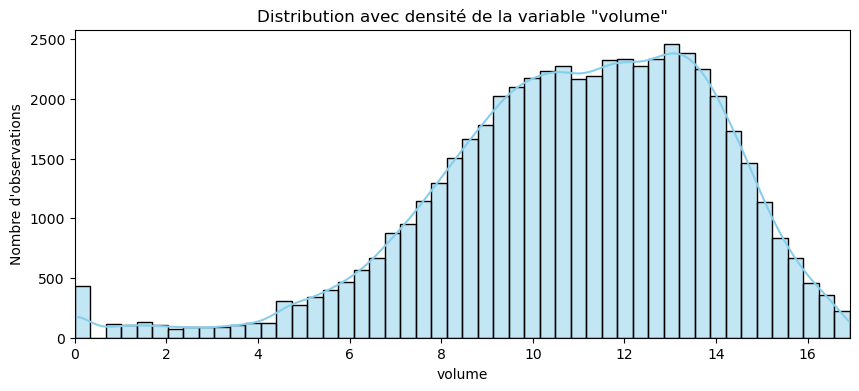
* **Statistique descriptive**

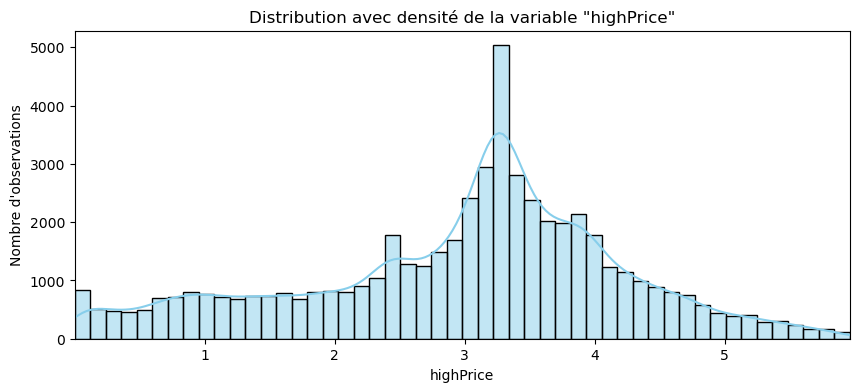


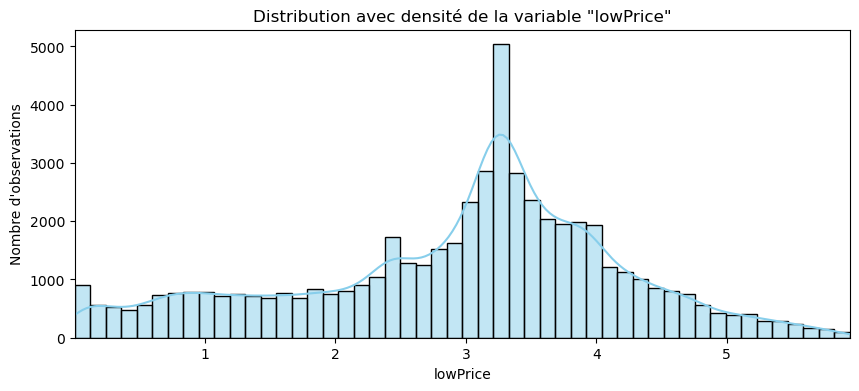
*Figure 19 : Les statistique des variables*

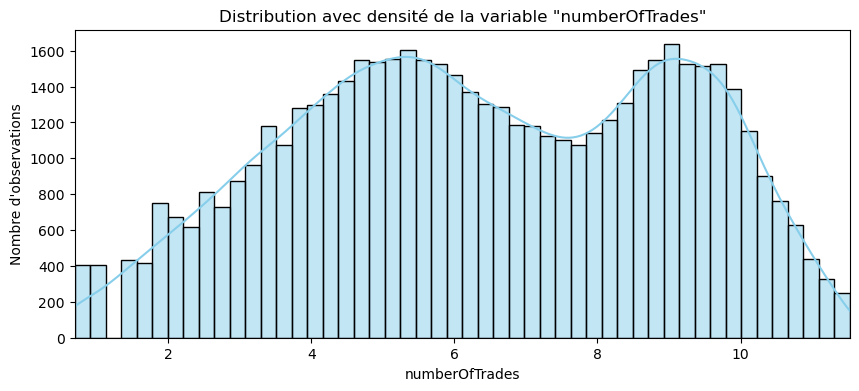
L’analyse des variables closePrice (prix de clôture) et volume (volume échangé) révèle une forte hétérogénéité et une distribution très asymétrique à droite. Le prix de clôture présente une moyenne de 7 118,63 mais une médiane bien plus basse (23,98), avec des valeurs extrêmes allant de 0,0019 à 384 millions, indiquant la présence de valeurs aberrantes ou de titres très atypiques. De même, le volume échangé varie considérablement, avec une moyenne de plus de 1,3 million mais une médiane de seulement 77 641,5, et un maximum dépassant 626 millions. Ces écarts soulignent l’importance de traitements statistiques robustes (comme la normalisation ou la transformation logarithmique) pour limiter l’influence des outliers dans les analyses ultérieures.

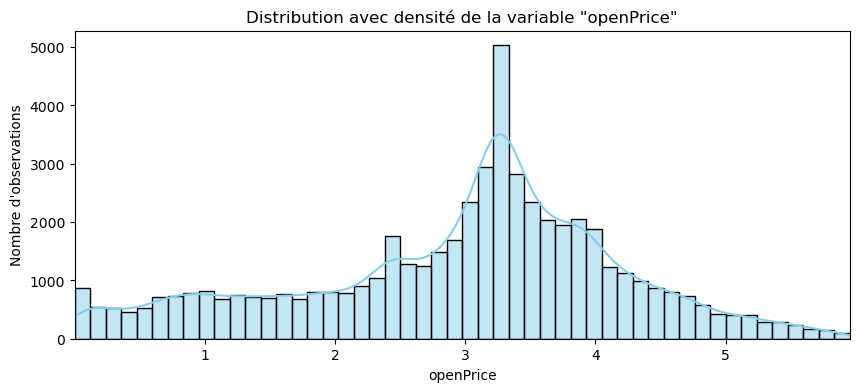


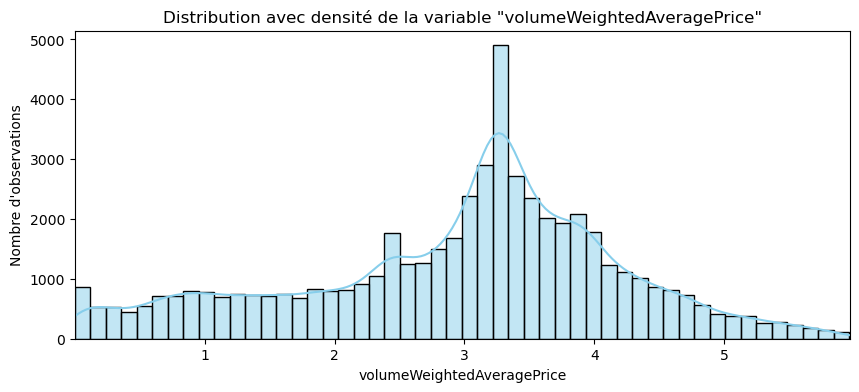




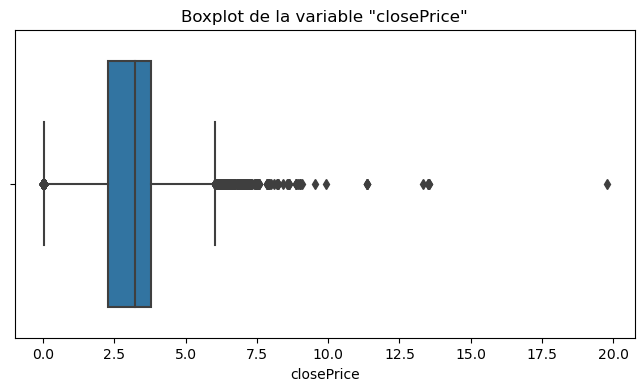


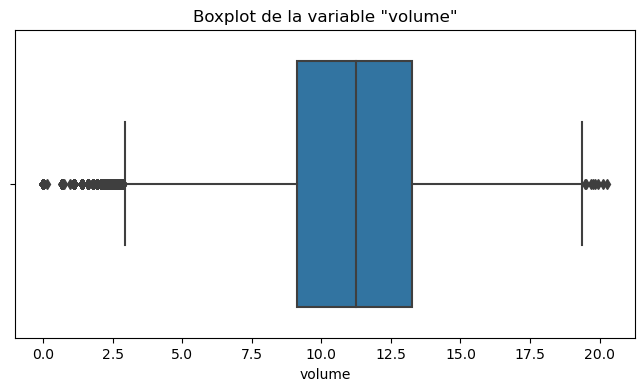




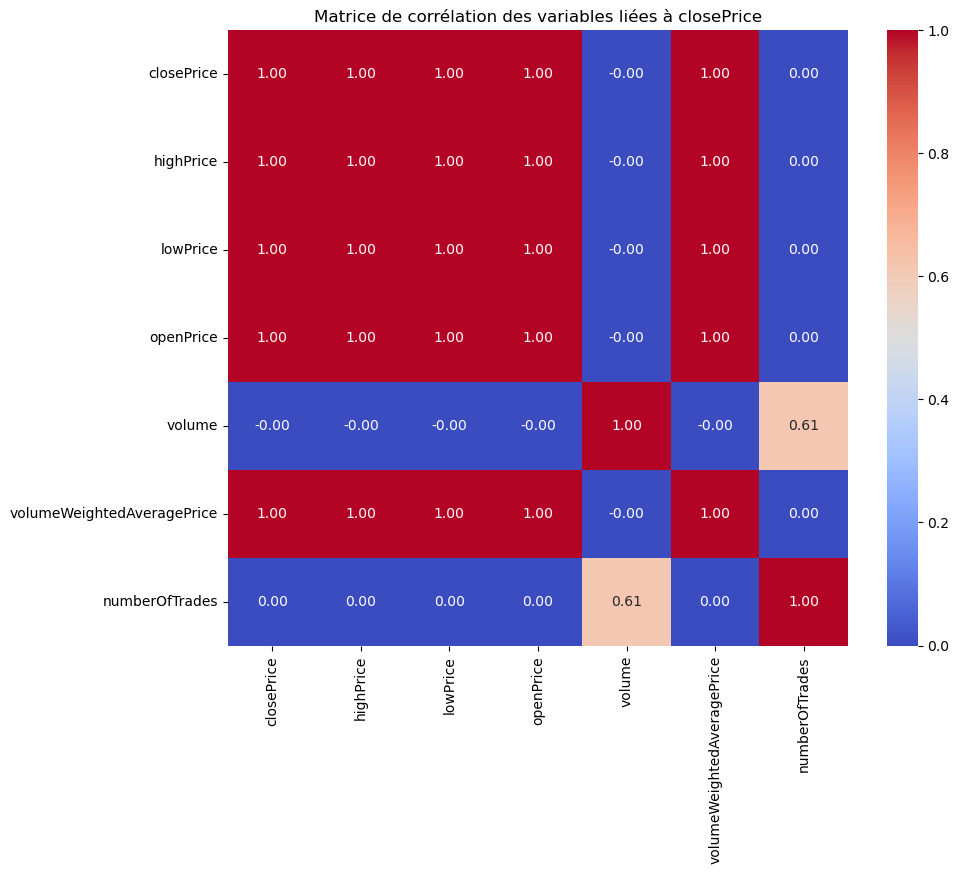


*Figure 20 : Distribution des différentes variables avec densité*





*Figure 21 : Distribution des variables closePrice et volume*



*Figure 22 : Visualisation de la corrélation des variables liées a closePrice*

1. **Choix des algorithmes de modélisation**

Dans ce projet de régression visant à prédire la variable closePrice à partir du volume, nous avons testé une large gamme de 13 algorithmes de modélisation répartis en différentes familles :

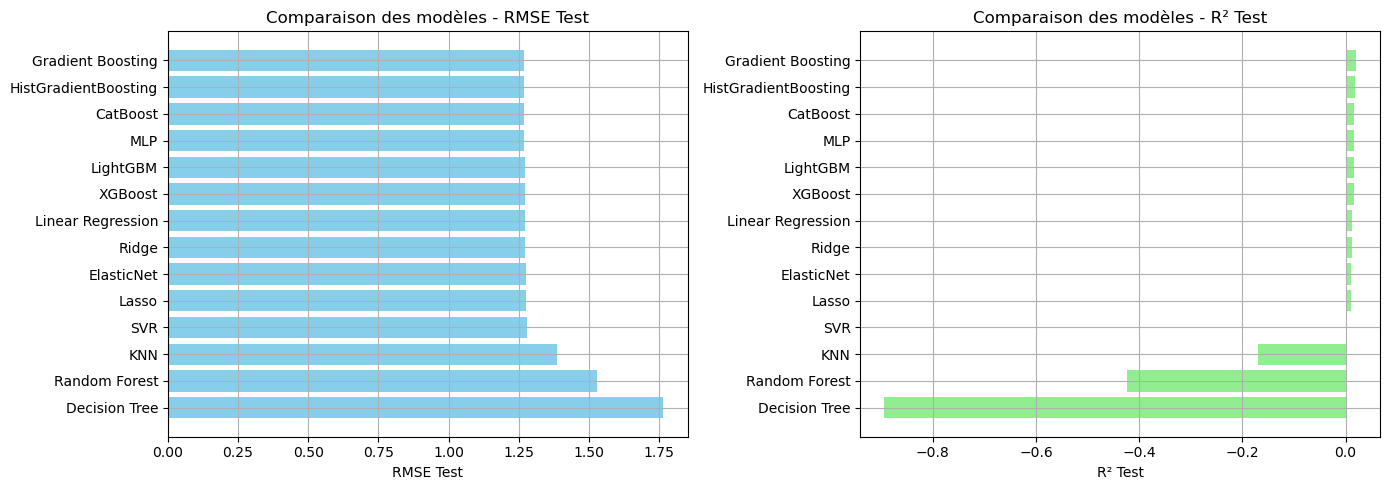
* Linear Regression
* Lasso
* Elastic Net
* Support Vector Regression (SVR)
* K-Nearest Neighbors (KNN)
* Decision Tree
* Random Forest
* Gradient Boosting
* Histogram-Based Gradient Boosting
* XGBoost
* LightGBM
* CatBoost
* Multi-layer Perceptron (MLP)

Cette sélection diversifiée permet de comparer différentes approches afin d’identifier le modèle le plus performant pour notre problématique de prédiction.

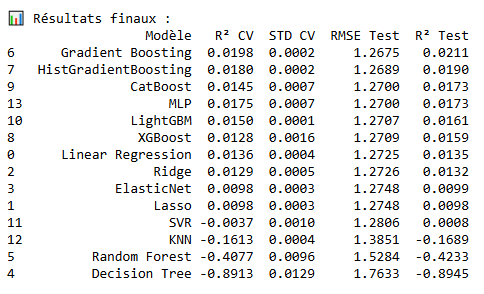
1. **Entraînement et évaluation des modèles**

Après avoir mis les données en bonne forme, et découper en ensemble d’apprentissage, et de test, nous passons à la phase d’entraînement.

Pour évaluer les performances des modèles, nous avons utilisé des métriques couramment employées dans ce type d’analyse : l’erreur absolue moyenne (MAE), la racine carrée de l’erreur quadratique moyenne (RMSE), et le coefficient de détermination (R²).



*Figure 23 : Analyse et comparaison des différents modèles*



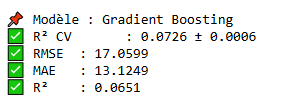
*Figure 24 : Résultat de performance des différents modèles*

Pour les deux métriques, l'algorithme Gradient Boosting s’est révélé être le plus performant.

# Présentation des Résultats

Avec le meilleur modèle obtenu (Gradient Boosting), nous l’avons entraîné sur tout le jeu de données d’apprentissage pour l’évaluer une seule fois sur le jeu de données de test, et nous obtenons les résultats suivants :

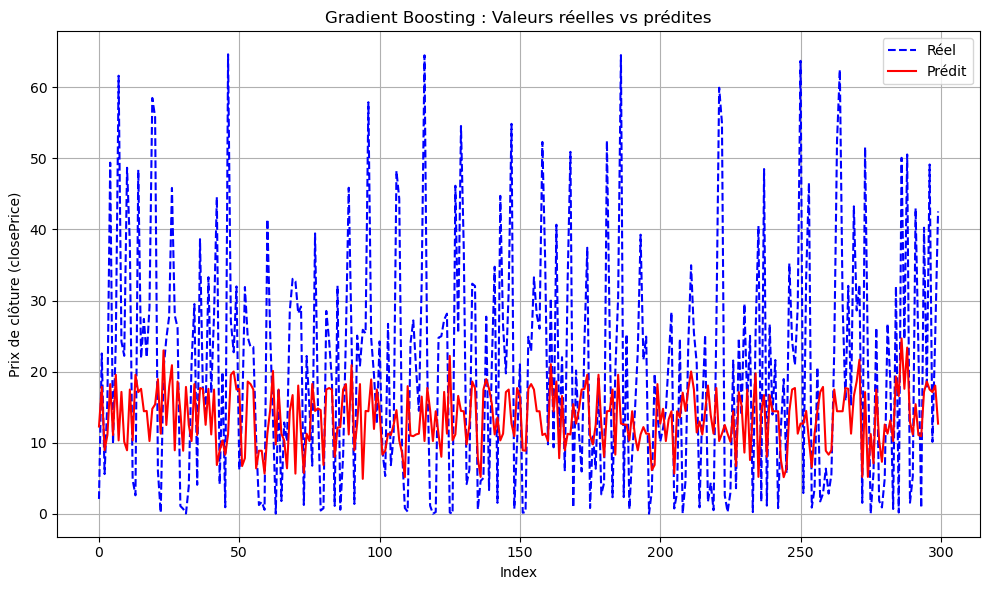
|  |  |  |
| --- | --- | --- |
| **MAE** | **RMSE** | **R2** |
| 13.1249 | 17.0599 | 0.0651 |



*Figure 25 : Performances du meilleur modèle sur les données de test*

Ces résultats indiquent que le modèle explique à peine 6.5% de la variance de la variable cible closePrice, et que l’erreur moyenne absolue reste élevée.

* **Visualisation de la performance**



*Figure 26 : Visualisation des performances*

* **Interprétation des faibles performances**

La faible performance du modèle est principalement due à la faible corrélation entre la variable explicative volume et la variable cible closePrice. En d'autres termes, même après transformation, le volume à lui seul ne permet pas d'expliquer efficacement les variations du prix de clôture.

* **Tentatives d’amélioration via Feature Engineering**

Pour compenser cette faiblesse, nous avons mis en place plusieurs techniques de feature engineering sur le volume :

* Transformation logarithmique : log(volume)
* Puissance 2 : (log(volume)) ²
* Puissance 3 : (log(volume)) ³
* Racine carrée : √(log(volume))

L’objectif était de capturer d’éventuelles relations non linéaires entre **volume** et **closePrice**. Malgré cela, le gain en performance est resté marginal.

* **Vers un meilleur modèle prédictif de *closePrice***

Conformément aux exigences du projet, la première version du modèle de prédiction a été développée en utilisant uniquement le volume échangé (volume) comme variable explicative. Cette approche, bien qu’alignée sur la consigne initiale du cahier des charges du projet, s’est révélée insuffisante en termes de performance. En effet, le prix de clôture d'une action est influencé par une multiplicité de facteurs liés à la dynamique du marché, et ne saurait être expliqué de manière satisfaisante par le seul volume.

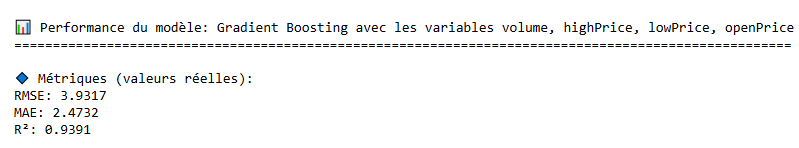
Pour répondre à cette limitation et construire un modèle plus robuste et explicatif, nous avons proposé une amélioration simple mais efficace : enrichir l’ensemble des variables explicatives utilisées pour entraîner le modèle. Ainsi, en complément du volume, nous avons intégré des variables clés :

* ***openPrice*** (prix d'ouverture)
* ***highPrice*** (prix le plus haut de la journée)
* ***lowPrice*** (prix le plus bas de la journée)

Ces variables, directement issues de l’environnement de cotation journalier, apportent un contexte plus complet et permettent de mieux comprendre la formation du closePrice.

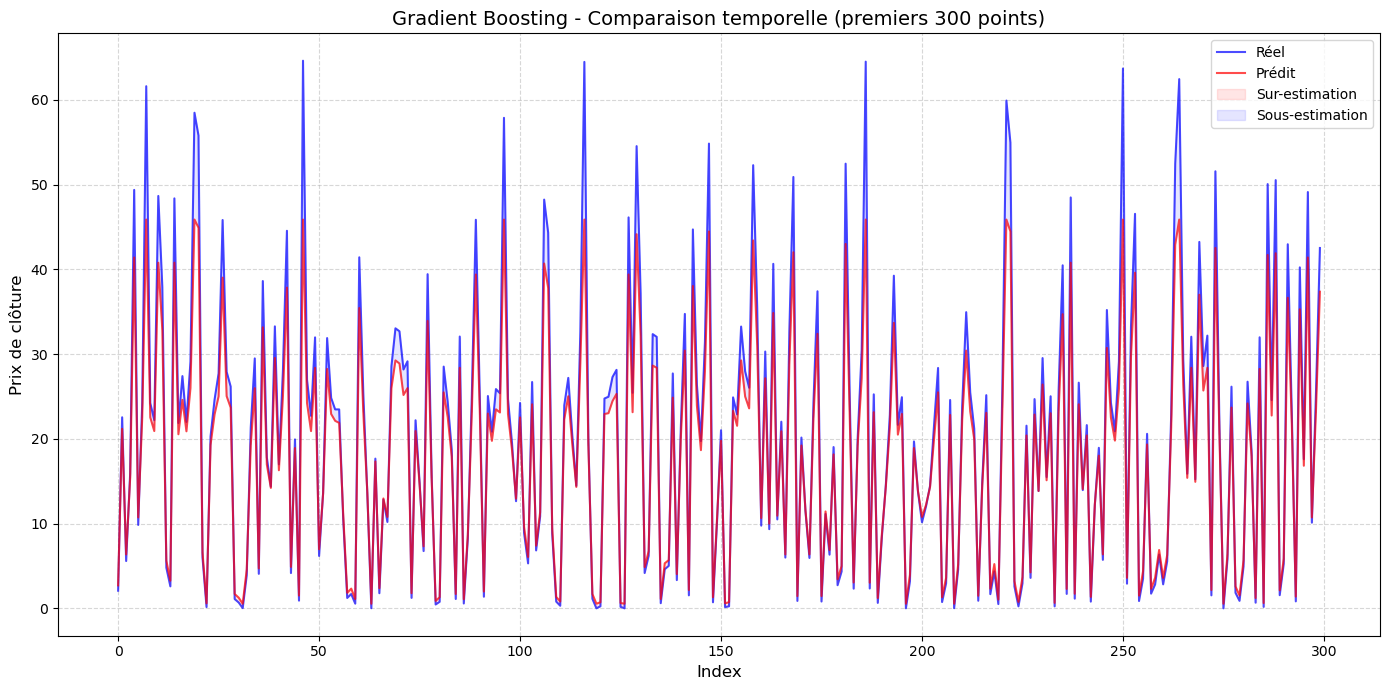
* **Résultats obtenus avec le nouveau jeu de variables**

|  |  |  |
| --- | --- | --- |
| **MAE** | **RMSE** | **R2** |
| 2.4732 | 3.9317 | 0.9391 |

****

Ces résultats révèlent que le modèle explique plus de 93 % de la variance du prix de clôture réel, ce qui constitue une performance très satisfaisante dans un contexte de prédiction financière. Le faible écart entre les valeurs prédites et les valeurs réelles démontre la pertinence de cette stratégie d'enrichissement.

* **Visualisation de la performance**



*Figure 27 : Visualisation des performances*

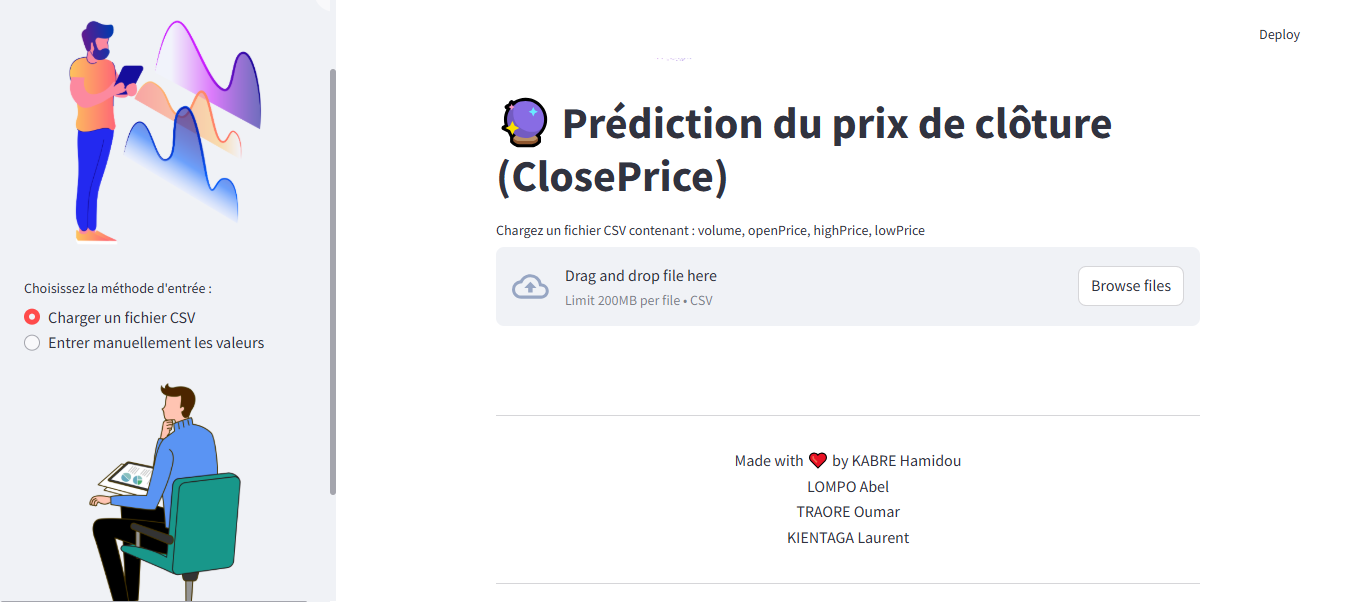
La comparaison visuelle entre les valeurs réelles et valeurs prédites montre une bonne cohérence temporelle. Le modèle capte fidèlement les tendances, tout en gardant des écarts d'erreur faibles et équilibrés entre les sur- et sous-estimations.

# Démonstration du système

Dans le but de valoriser le modèle de prédiction développé au cours du projet, nous avons décidé, de manière autonome, de concevoir une petite application illustrant concrètement le fonctionnement du système. Cette section présente, à travers une série de captures d’écran, les principales fonctionnalités de cette application.

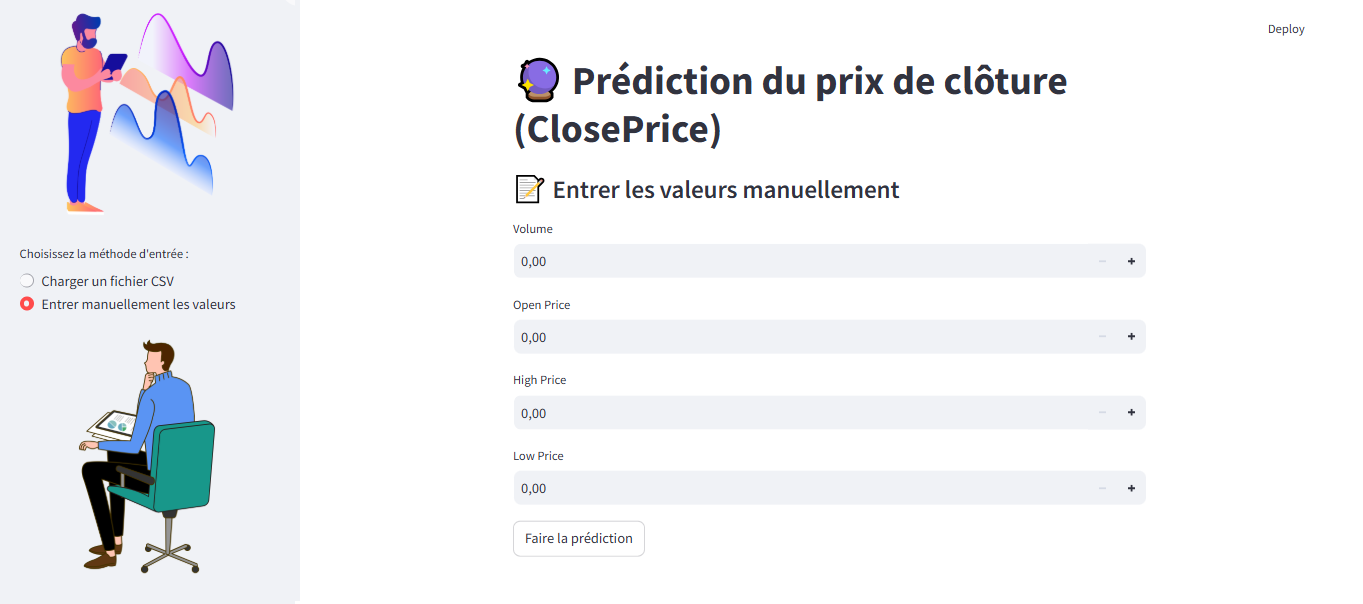


*Figure 28 : Page d’accueil du système*

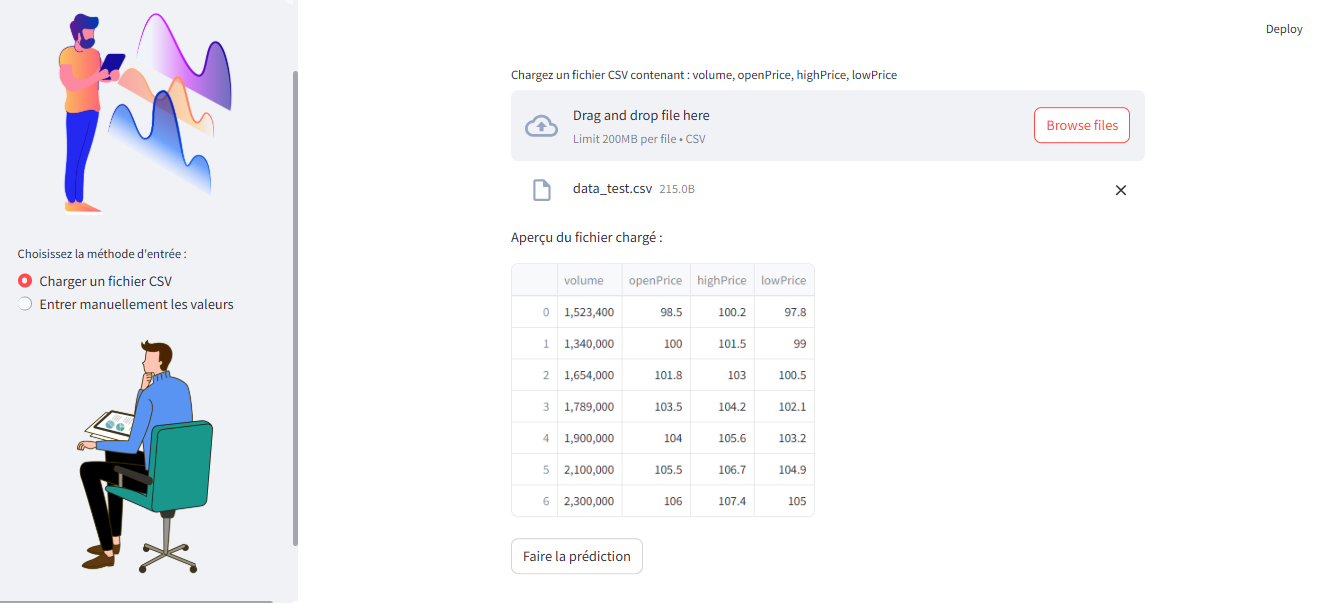


*Figure 29 : Visualisation de la fonctionnalité « charger un fichier »*

L’utilisateur peut, depuis la page d’accueil, charger un fichier CSV ou saisir manuellement des données pour la prédiction.

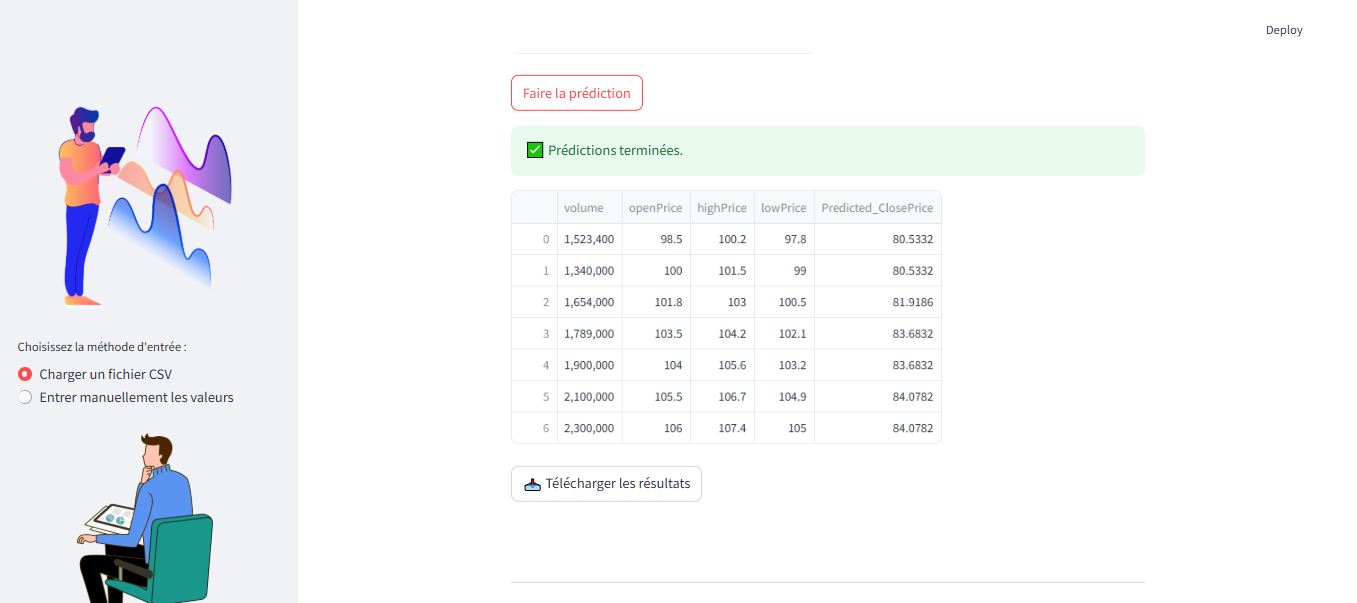


*Figure 30 : Visualisation de la fonctionnalité « Entrer manuellement les valeurs »*



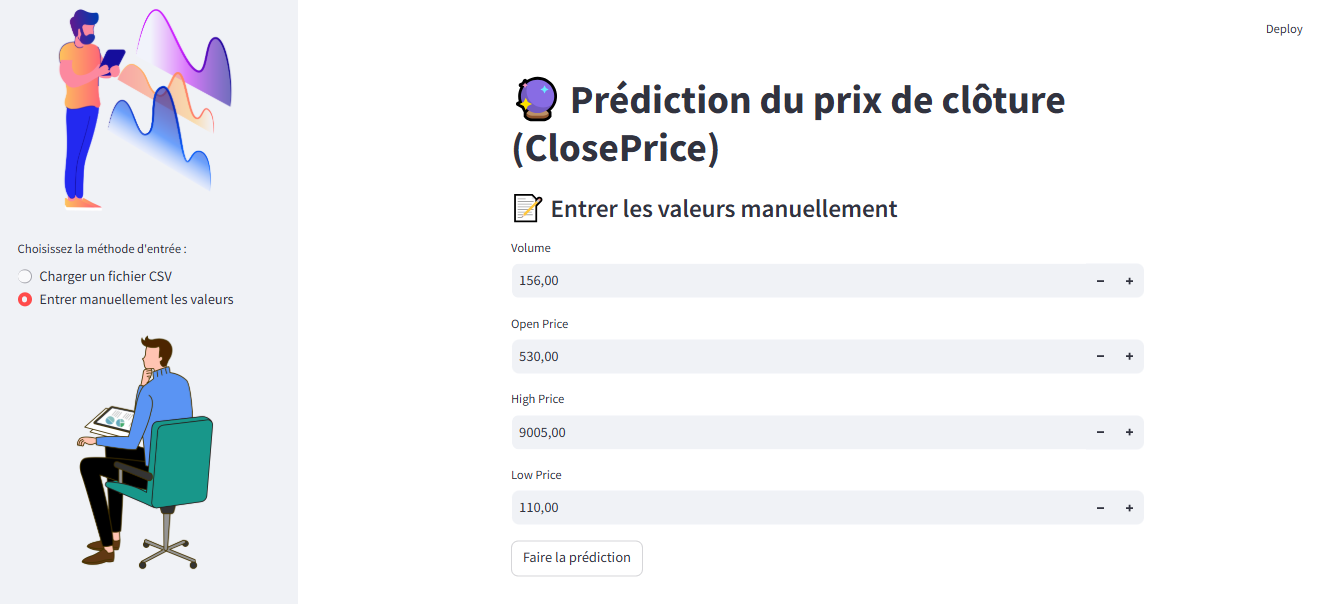
*Figure 31 : Chargement d’un fichier csv contenant des valeurs pour la prédiction*

Dans cette figure, nous remarquons qu’un fichier CSV a été charger par l’utilisateur pour faire la prédiction. Ces variables et ces valeurs sont visible dans l’aperçu du fichier chargé.



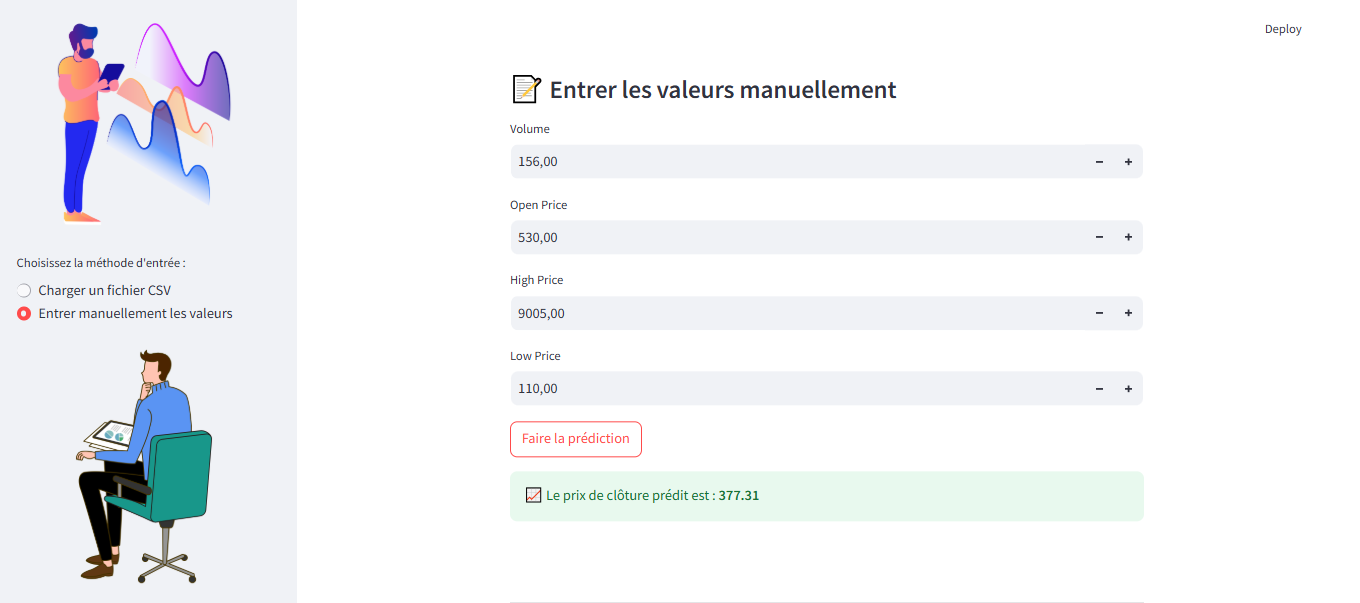
*Figure 32 : Prédiction des valeurs de closePrice en fonction des valeurs du fichier fournit*

Après le chargement du fichier CSV, la prédiction de la variable cible a été effectuée à partir des valeurs fournies. Les résultats sont visibles dans l’image ci-dessus et peuvent être téléchargés par l’utilisateur.



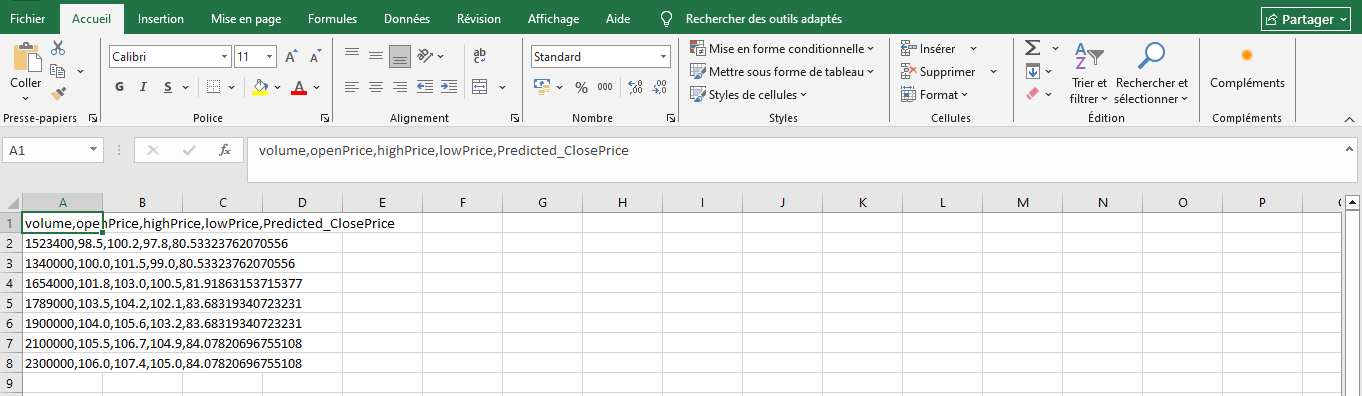
*Figure 33 : Saisis des valeurs par l’utilisateur pour la prédiction*

Ensuite l’utilisateur a également le choix de saisir manuellement les valeurs pour la prédiction. Dans ce cas, des valeurs ont été saisit pour une éventuelle prédiction.



*Figure 34 : Prédiction des valeurs de closePrise à partir des valeurs saisit*

La valeur prédite, obtenue à partir des données saisies, est affichée, illustrant les bonnes performances et la fiabilité du modèle.



*Figure 35 : Exemples des données du fichier csv utilisé pour la prédiction*

# Conclusion

Ce projet a permis de mettre en œuvre une solution complète de traitement de données financières en temps réel en s’appuyant sur l’architecture Kappa. En exploitant des outils robustes tels qu’Apache Kafka pour le streaming, Spring Boot pour l’orchestration des flux, Polygon.io pour la collecte des données financières, et Elasticsearch pour l’indexation et la visualisation, nous avons pu construire un pipeline efficace du flux de données jusqu’à l’analyse.

L’intégration d’un modèle de Machine Learning a permis d’apporter une valeur ajoutée en prédisant la variable "closePrice" à partir du volume des transactions. La création d'une interface utilisateur pour illustrer les fonctionnalités a également facilité la démonstration de bout en bout du système.

Ce projet démontre la pertinence de l’architecture Kappa dans les contextes où la réactivité aux données temps réel est cruciale, tout en restant plus simple à maintenir qu’une architecture Lambda. Des pistes d'amélioration peuvent inclure l’ajout de mécanismes d’alerte en cas de fluctuations importantes, l’enrichissement des données, ou encore le déploiement sur des infrastructures cloud pour assurer une scalabilité optimale.