**PROJET**

**ECOSYSTEMES BIG DATA**

**Introduction**

L'avènement du Big Data a révolutionné la manière dont les entreprises et les organisations gèrent et tirent des insights de quantités massives de données. En accord avec la demande croissante pour des solutions scalables et efficaces pour gérer, traiter et analyser d'énormes ensembles de données, notre projet se concentre sur l'utilisation de systèmes distribués Open Source.

Dans le cadre d'un cas pratique au sein du secteur bancaire, notre projet vise à simuler le traitement de transactions bancaires en temps réel. Nous cherchons à mettre en place une infrastructure capable de gérer efficacement les flux de données en temps réel, de les stocker de manière sécurisée et d'en extraire des insights utiles pour les agents bancaires. Pour atteindre cet objectif, nous allons utiliser Kafka, une plateforme de streaming distribuée, pour la gestion des données en temps réel. Kafka nous permettra de collecter, traiter et distribuer les flux de données de manière efficace et fiable. Les données collectées en temps réel seront ensuite stockées dans une base de données SQLite. SQLite est une base de données relationnelle légère et autonome, idéale pour notre cas d'utilisation, car elle offre une intégration facile. Enfin, nous utiliserons ces données stockées pour créer des tableaux de bord de visualisation. Ces tableaux de bord permettront aux agents bancaires de suivre en temps réel les différentes transactions effectuées par les clients. Ils offriront une vue d'ensemble intuitive et personnalisable des données, permettant aux agents de prendre des décisions éclairées et de fournir un service clientèle de qualité.

Dans les sections suivantes de ce rapport, nous détaillerons les différentes étapes de mise en œuvre de notre projet, en mettant en évidence les technologies utilisées, les défis rencontrés et les solutions apportées.

**Lien de notre dépôt GitHub (avec les instructions pour l’installation du projet) : https://github.com/ritmosky/Projet-Big-Data-ECE-Kafka-Streaming**

Une image contenant texte, capture d’écran, Police, diagramme

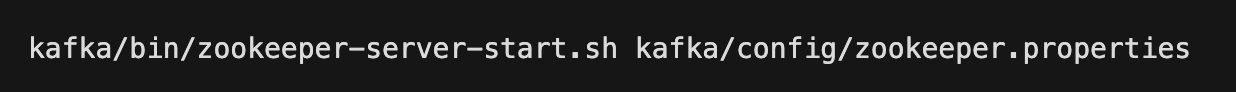
Description générée automatiquement

**Architecture de notre projet**

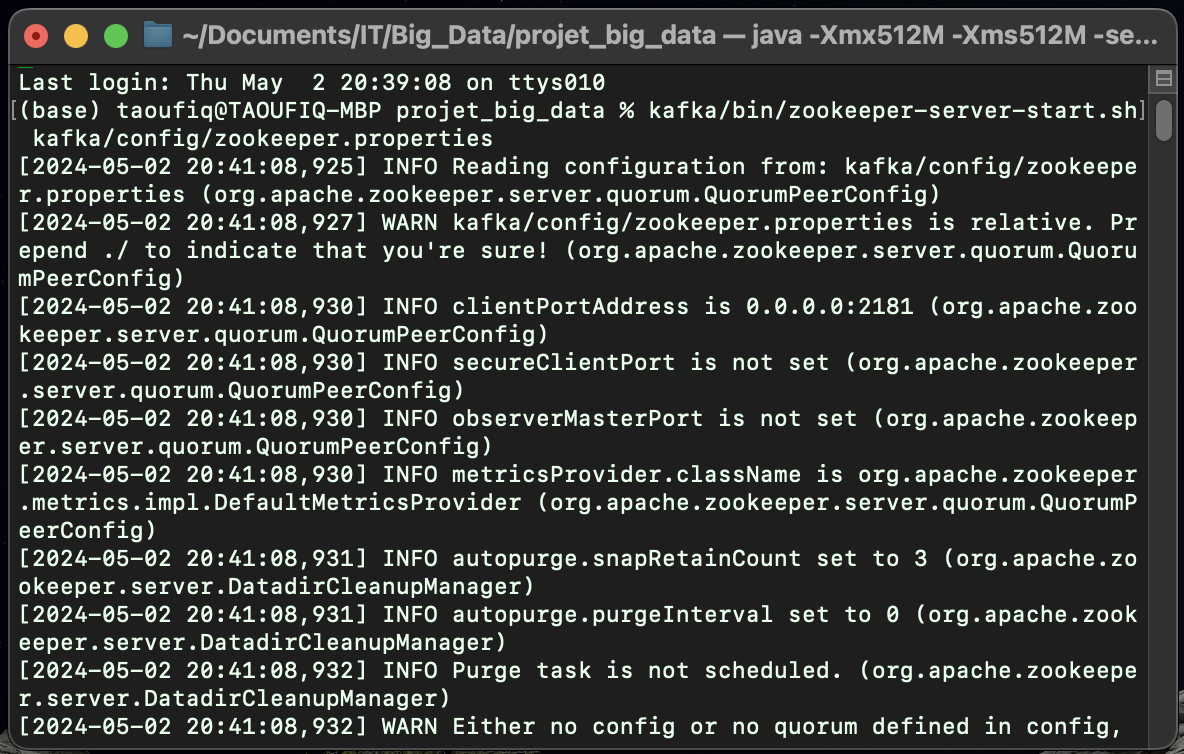
**I. Mise en place de l’environnement KAFKA**

L'interaction entre ZooKeeper et Kafka est essentielle pour assurer le bon fonctionnement et la haute disponibilité de Kafka. ZooKeeper agit comme un service de coordination et de gestion de configuration distribué, tandis que Kafka l'utilise pour diverses tâches, notamment la gestion des brokers Kafka, le suivi des partitions et des topics, ainsi que pour la gestion des groupes de consommateurs.

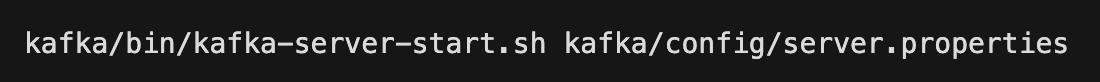
**1. Nous lançons le service ZooKeeper avec la commande :**

****

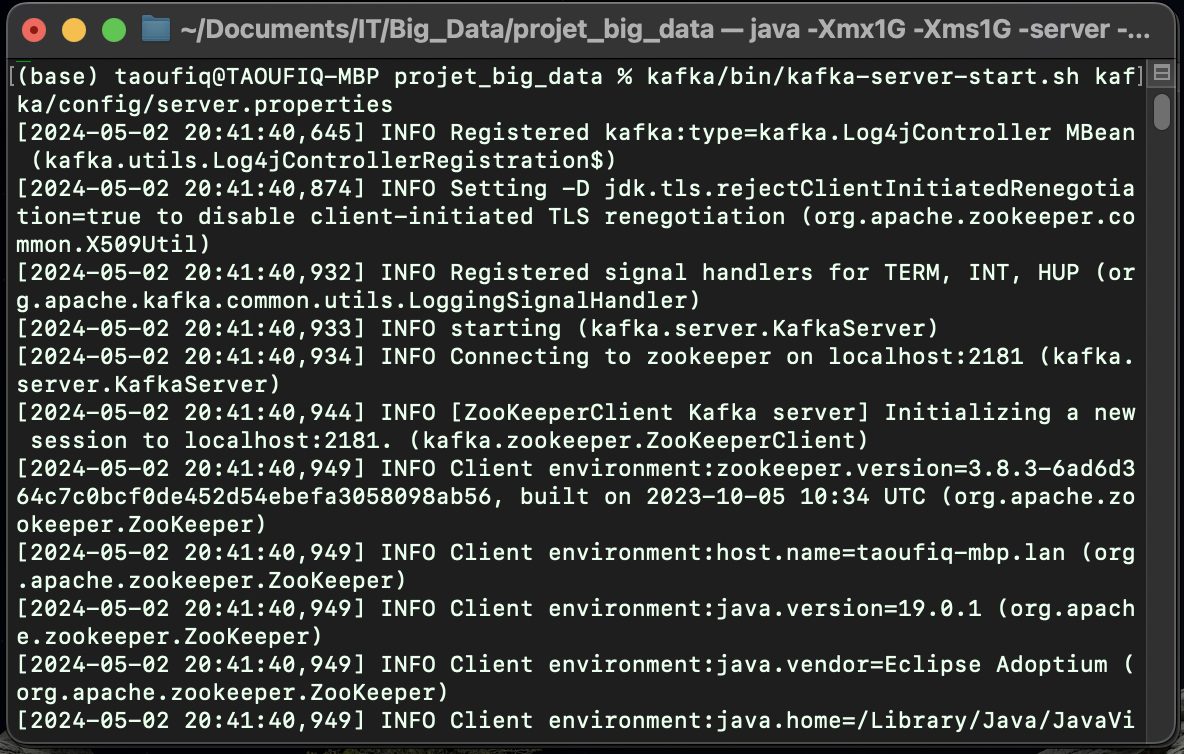
Ainsi, ZooKeeper démarre et commence à écouter les demandes des clients sur le port spécifié dans le fichier de configuration « **zookeeper.properties** ». Il agit comme un service de coordination. Il conserve un état en mémoire distribué, qui est utilisé par les brokers Kafka pour stocker des métadonnées telles que la configuration, les partitions et les topics.



**2. Une fois que ZooKeeper est en cours d'exécution, nous ouvrons un nouveau terminal pour démarrer le service Kafka Broker à l'aide de la commande :**



Le broker Kafka démarre et commence à écouter les requêtes des « **Producer** » et des « **Consumer** » de données sur le port spécifié dans le fichier de configuration « server.properties ». Pendant son fonctionnement, le broker Kafka communique régulièrement avec ZooKeeper pour mettre à jour son état, signaler les partitions disponibles et maintenir la cohérence des données distribuées.



Une fois que tous les services auront été lancés avec succès, nous aurons un environnement Kafka de base opérationnel et prêt à l'emploi.

**II. Création de Topic pour stocker les évènements**

Kafka est une plateforme de streaming d'événements distribuée qui permet de lire, écrire, stocker et traiter des événements en temps réel. Avant de pouvoir écrire nos premiers événements dans Kafka, nous devons créer un sujet, « **Topic** ». La création d'un Topic dans Kafka est une étape essentielle avant de pouvoir écrire ou consommer des événements. Les Topics permettent de regrouper les événements de manière logique et de les traiter de manière séparée selon les besoins de l'application.

**1. On crée un nouveau topic nommé « project\_bigdata » en lançant la commande suivante dans un nouveau terminal**

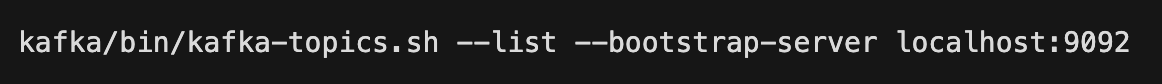
****

--create : indique que nous voulons créer un nouveau sujet.

--topic ProjectBigData : spécifie le nom du nouveau sujet, dans ce cas "ProjectBigData".

--bootstrap-server localhost:9092 : indique l'adresse du serveur Kafka que nous utilisons pour la gestion des requêtes. Dans cet exemple, Kafka est exécuté localement sur le port 9092.

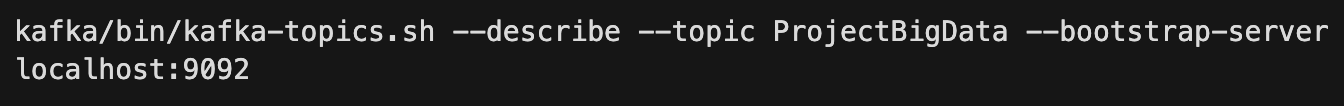
Lorsque cette commande est exécutée, Kafka crée un nouveau sujet appelé "**ProjectBigData**" avec les paramètres par défaut, prêt à recevoir des événements. Nous pouvons vérifier la création en tapant la commande :



Une image contenant texte, capture d’écran, multimédia, logiciel

Description générée automatiquement

**2. On peut également afficher les détails du nouveau topic crée avec la commande :**



Une image contenant texte, Police, capture d’écran

Description générée automatiquement

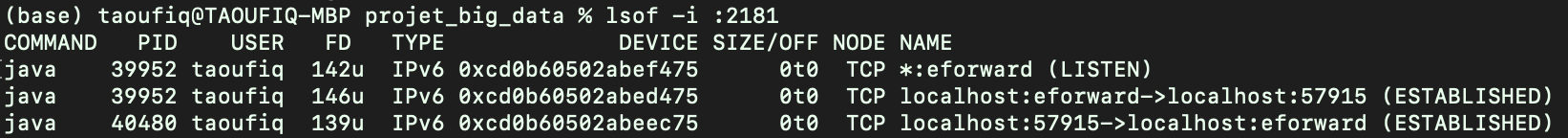
**3. Pendant le développement du projet, il arrivait parfois que les services ZooKeeper et Kafka Broker ne se lançaient pas.**

En effectuant des recherches, nous nous sommes rendu compte que cela était dû à des conflits de ports. En effet, avant de lancer les services (ZooKeeper et Kafka), il faut vérifier que les ports qu'ils utilisent ne soient pas déjà utilisés par d'autres processus car cela pourrait engendrer des conflits. Si plusieurs processus écoutent sur le même port, il faudra kill ces processus.

Pour lister les processus écoutant sur un port, on tape :

Une image contenant texte, Police, capture d’écran, Graphique

Description générée automatiquement



On peut identifier l’identifiant « PID » du processus qu’on désire terminer puis taper la commande :

Une image contenant texte, Police, Graphique, noir

Description générée automatiquement

|  |  |
| --- | --- |
| **Service** | **Port utilisé** |
| ZooKeeper | 2181 |
| Kafka Broker | 9092 |

**III. Création de Producer et de Consumer**

Un client Kafka communique avec les brokers Kafka via le réseau. Une fois reçus, les brokers stockent les événements de manière durable et tolérante aux pannes.

Pour écrire des événements dans le sujet, nous utiliserons un « **Producer** » Kafka qui envoie les données vers les brokers Kafka. Comme les événements sont stockés de manière durable dans Kafka, ils peuvent être lus autant de fois et par autant de « **Consumer** » que nécessaire. Cette architecture permet une communication asynchrone et distribuée entre les producteurs et les consommateurs, garantissant une gestion efficace et fiable des flux de données en temps réel.

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

Dans le cas de notre projet, le « Producer » et le « Consumer » ont été codés en Python.

Le « Consumer » se charge de créer et réinitialiser la base de données dans un premier temps. Ensuite il s’abonne au Topic crée afin de lire les évènements.

Le « Producer », lui, se charge de transmettre les données au Topic. D’abord il transmet les données présentent dans notre fichier csv (contenant l’historique de quelques transactions). Ensuite il lance une application « **Tkinter** » au travers de laquelle l’utilisateur pourra se connecter à un compte et effectuer des transactions qui seront transmises en temps réels au Topic. Puis, il lance une interface web développée grâce à l’API Python « **Streamlit** » sur le lien [**http://localhost:8501**](http://localhost:8501) . On pourra observer les dashboards sur cette interface web et rafraichir la page à chaque fois que des transactions auront été renseignées grâce à l’interface Tkinter. Le « Producer » se connecte donc à la base de données afin de mettre à jour les affichages dans les applications Tkinter et Streamlit.

**Tout au long du projet, la base de données manipulée a pour schéma :**

**Une image contenant texte, Police, capture d’écran

Description générée automatiquement**

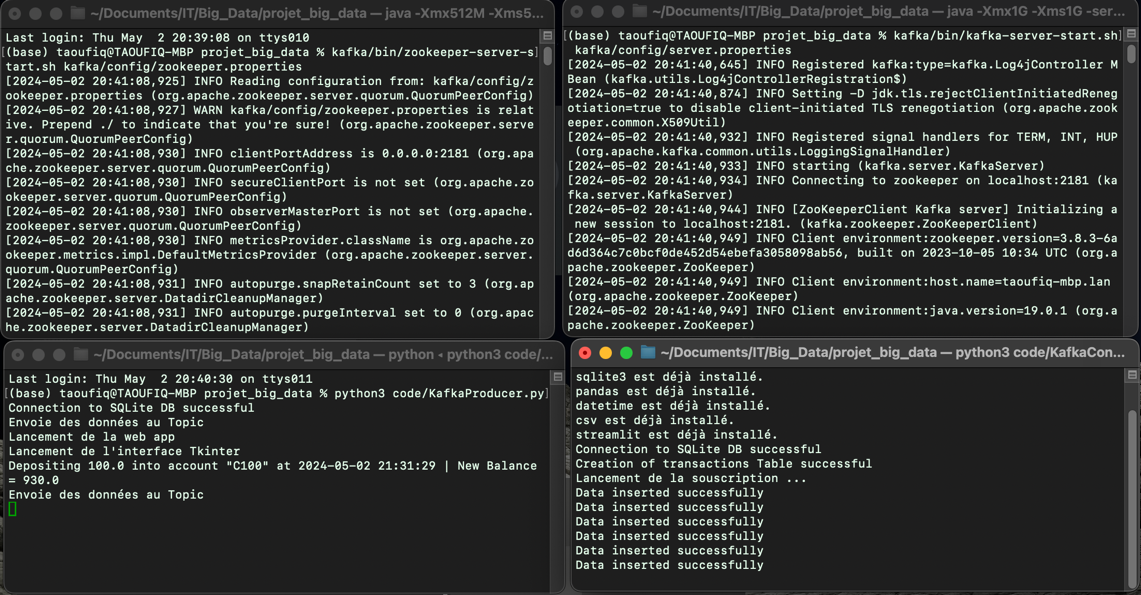
**Affichage de tous les 4 terminaux (nécessaires au projet):**

1 = terminal pour le service ZooKeeper

2 = terminal pour le service Kafka Broker

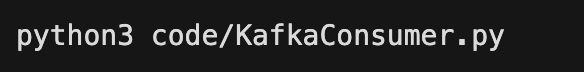
3 = terminal pour le Producer

4 = terminal pour le Consumer





**1. Lancement du Consumer**



Nous allons d’abord lancer le « Consumer » avant le « Producer » afin de pouvoir observer la lecture des données par ce dernier.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

**2. Lancement du Producer**

Une image contenant Police, texte, Graphique, noir

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

**3. Affichage du Producer et Consumer**

Une image contenant texte, Logiciel multimédia, logiciel, capture d’écran

Description générée automatiquement

**4. Lancement de l’interface Tkinter**

**Affichage de l’interface Tkinter au lancement :**

Une image contenant texte, capture d’écran, logiciel, multimédia

Description générée automatiquement Une image contenant texte, capture d’écran, logiciel, affichage

Description générée automatiquement

**Affichage de l’interface Tkinter après une transaction :**

On observe qu’après la transaction de 1OO€, le montant de la balance sur l’image de droite a changé. Elle est passée de 830 à 930, ce qui veut dire qu’un dépôt a été effectué. Cet affichage fonctionne correctement car nous avons connecté l’application à la base de données afin d’enregistrer les modifications et effectuer les mises à jour.

Dans ce processus, les données récupérées via l’interface graphique sont envoyées dans le Topic grâce au Producer. Ensuite le Consumer s’abonne au Topic, récupère les données et les charge dans la base de données. Enfin, le Producer se reconnecte à la base de données pour récupérer les données et ainsi changer l’affichage.

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

**Affichage du Producer et du Consumer après une transaction :**

On peut également observer que de nouvelles lignes se sont rajoutées sur nos 2 terminaux. A gauche (côté Producer) on notifie que la transaction a été effectuée. A droite (côté Consumer) on notifie que les données ont bien été insérées dans la base de données.

Une image contenant texte, Logiciel multimédia, logiciel, Logiciel de graphisme

Description générée automatiquement

**5. Lancement de l’interface web Streamlit**

**Interface web Streamlit (affichage des transactions) :**

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

**Interface web Streamlit (dashboard transaction par compte) :**

Une image contenant capture d’écran, texte, logiciel, Logiciel multimédia

Description générée automatiquement

**Interface web Streamlit (dashboard balance par compte) :**

Une image contenant capture d’écran, ligne, Tracé, logiciel

Description générée automatiquement

**Conclusion**

En conclusion, notre projet Big Data a été une exploration passionnante des technologies distribuées et des pratiques modernes de gestion des données en temps réel. Nous avons pu mettre en œuvre une architecture robuste utilisant Kafka comme plateforme de streaming d'événements, ZooKeeper pour la coordination et la gestion de configuration, ainsi que SQLite pour le stockage des données. En simulant le traitement de transactions bancaires en temps réel, notre projet a démontré l'efficacité et la fiabilité des solutions Open Source dans la gestion des flux de données massifs. Grâce à Kafka, nous avons pu garantir la disponibilité et la durabilité des événements, tout en permettant une lecture multiple par des consommateurs multiples.

Une deuxième difficulté que nous avons rencontré lors de nos phases de tests finaux est la gestion de toutes les dépendances relatives au projet notamment installation de Java, nécessaire au lancement de Kafka ou encore l’installation de Python. A un moment, pour régler ce soucis nous avons exploré l’idée d’utiliser « **Docker** » afin de gérer ces dépendances. Toutefois nous avons également rencontré quelques difficultés pour faire cela et nous avons donc poursuivi sur notre idée initiale.

Une autre a été la gestion des packages Python. Lorsque nous testions le projet sur d’autres ordinateurs, il nous arrivait d’avoir des erreurs relatives à l’import de certaines bibliothèques malgré notre script d’installation automatique de packages. En effet, il est généralement conseillé de lancer tout projet Python avec un environnement virtuel dédié. Nous avons essayé d’en mettre un en place mais nous cela nous aurait pris beaucoup plus de temps que prévu, également il aurait été plus facile de le faire par le biais de Docker.

En somme, notre projet a démontré le potentiel transformateur des technologies Big Data et plus particulièrement dans le secteur bancaire, offrant des solutions innovantes pour améliorer l'efficacité opérationnelle, optimiser la prise de décision et offrir un service clientèle de qualité supérieure.