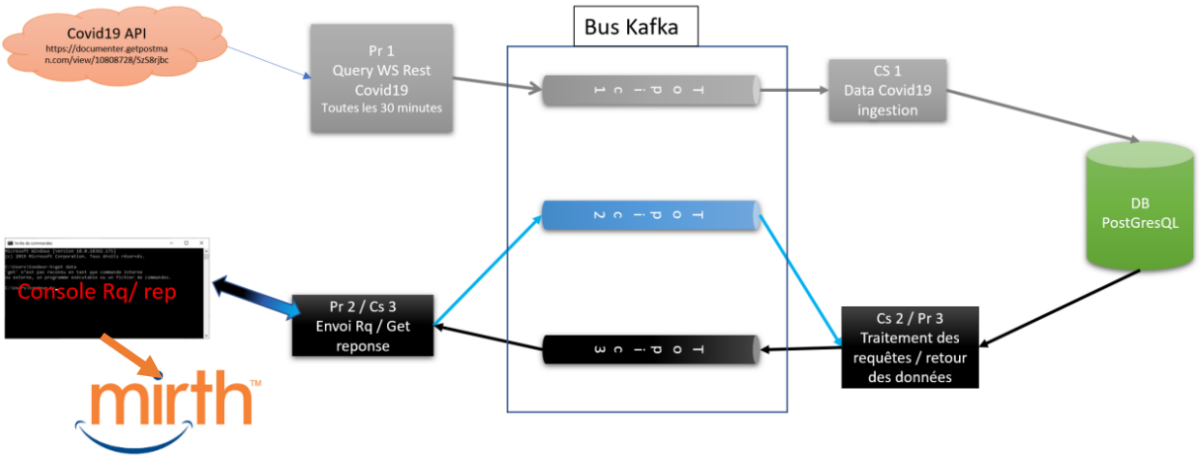
Carlier Amandine
Denorme William
Deroubaix François

FISA 4 Informatique

Intergiciel et programmation par composant



Monsieur Hervé TONDEUR

INSA

Université Polytechnique HAUTS-DE-FRANCE

Table des matières

Introduction

Dans le cadre du module intergiciel et programmation par composant, nous avons eu l'occasion, entre autres, d'apprendre ou de réapprendre le mode de fonctionnement des API KafkaClient. Lors des cours magistraux et travaux dirigés, nous avons vu quels étaient leur rôle, ainsi que leur utilité dans divers contextes applicatifs. Dans la continuité de ces cours, un projet nous a été proposé.

L'objectif de ce dernier est de réaliser une application en mode console texte ou web. Dans notre cas, nous avons choisi une application web, qui utilise le langage Java, les API KafkaClient et le Framework SpringBoot. Ces derniers nous permettent de lancer des requêtes Custom 1 au travers d'un producteur Kafka 2 vers le Topic Kafka 2, ce message est par la suite interprété par le Custom consommateur 2 connecté à ce topic et permet de requêter la base de données PostGreSQL selon le besoin exprimé par la requête. Le retour de ses informations se fait par le Custom producteur 3 vers le Topic Kafka et le consommateur 3 a pour but de consommer les messages de façon à le présenter en affichage sur la console.

Comme exprimé dans les objectifs de ce TP, nous utilisons une base de données PostGreSQL. Cette dernière est alimentée par les données provenant du site COVID19 API toutes les 30 minutes, en JSON. Un custom producteur 1 réalise cette requête API vers ce site et pousse le résultat vers le topic Kafka 1. Par la suite, un consommateur custom 1 va consommer le message et ingérer les données dans la base PostgreSQL dédiée.

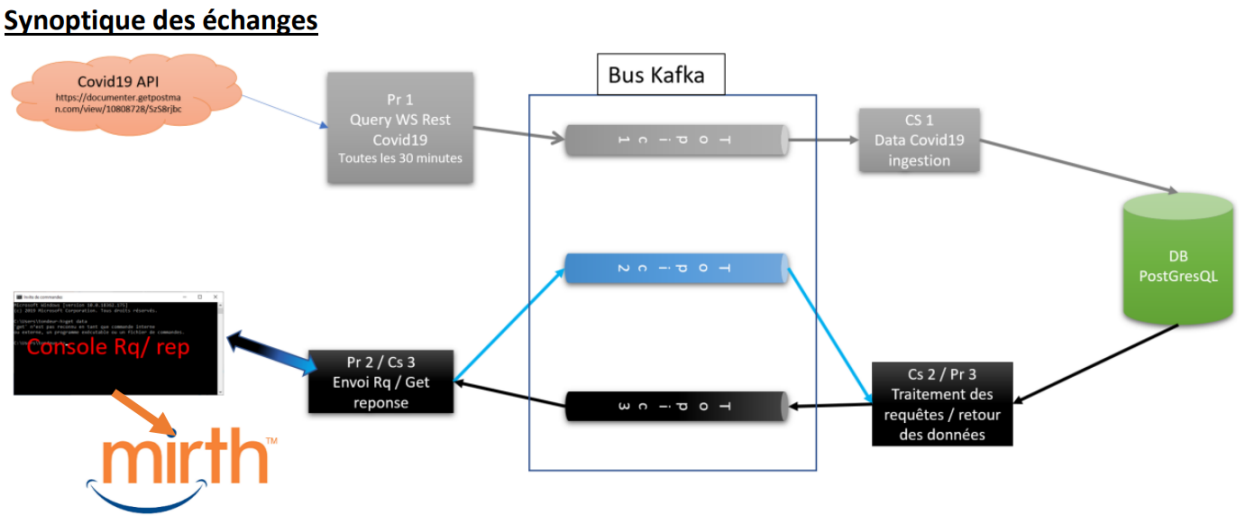
Pour la mise en œuvre de cette application, nous nous avons de choisi de s'associer à trois étudiants, Amandine CARLIER (n°21700078), William DENORME (n°21903046) et François DEROUBAIX (n°22105578). De cette façon, nous avons pu répartir la charge de travail de l'ensemble du projet.

Pour répondre au besoin de ce module et expliquer nos multiples implémentations, nous exposerons dans ce rapport un ensemble de points dont l'architecture mise en place, ainsi que les réponses aux différentes questions.

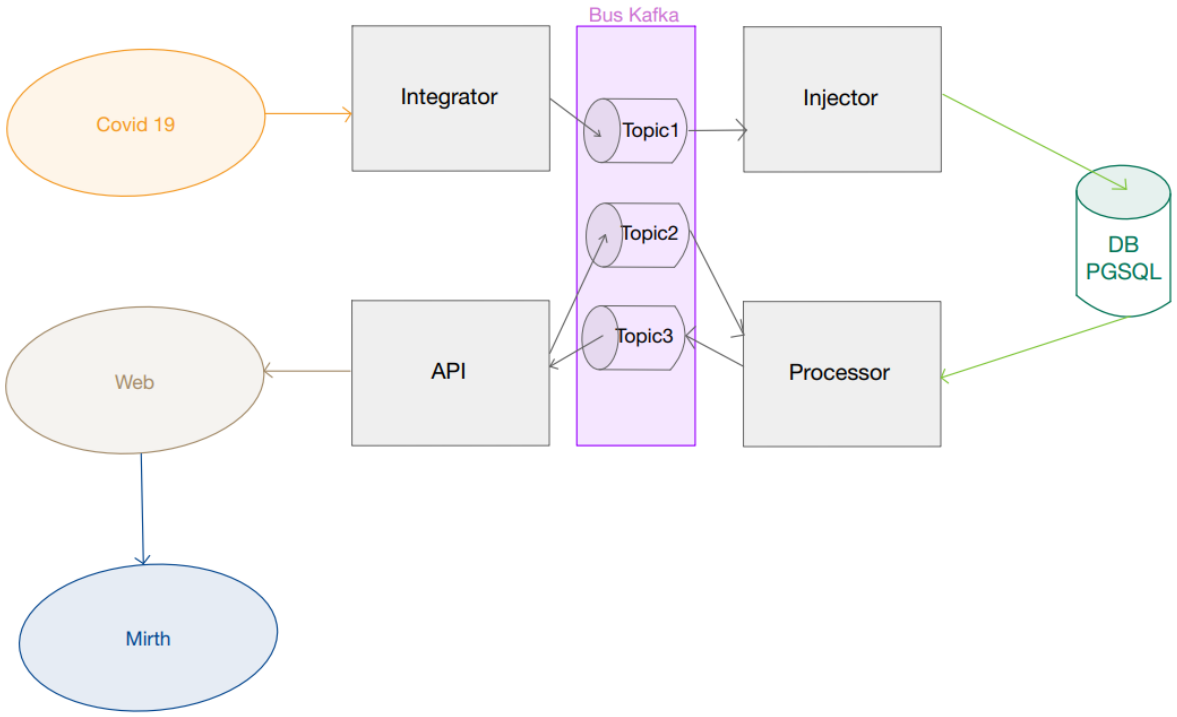
Remarque : Pour rappel, voici le lien git de notre projet :
https://github.com/Wanadian/kafka_project

Architecture choisie

De façon à comprendre ce qui été demandé, un synoptique des échanges a été mis à disposition :



Au vu de la clarté de ce schéma, nous sommes partis naturellement sur une architecture comportant cinq packages. Voici ce que donne notre architecture technique avec les éléments de départ :



Nos cinqs blocs de codes sont donc les suivants :

- *Integrator* : il permet à un producteur de récupérer des données provenant du site COVID19 API toutes les 30 minutes et de les renvoyer vers le premier topic kafka (ici, nous requêtons les données);

- *Injector* : il permet à un consommateur custom 1 de consommer les messages du premier topic et d'ingérer les données dans la base PostgreSQL dédiée (ici, nous consommons les données) ;

- *Processor* : il permet au custom producteur 3 d'envoyer des informations dans le troisième topic kafka et au consommateur custom 2 de récupérer les informations du custom producteur 2 via le second topic kafka (ici, nous traitons les données) ;

- *API* : il permet au custom producteur 2 d'envoyer des informations dans le deuxième topic kafka et au consommateur custom 3 de recevoir des informations du topic kafka 3 (ici, nous avons le point d'entrée de nos requêtes) ;

- *Web* : il correspond à l'interface graphique de notre projet, celui-ci va interroger notre API.

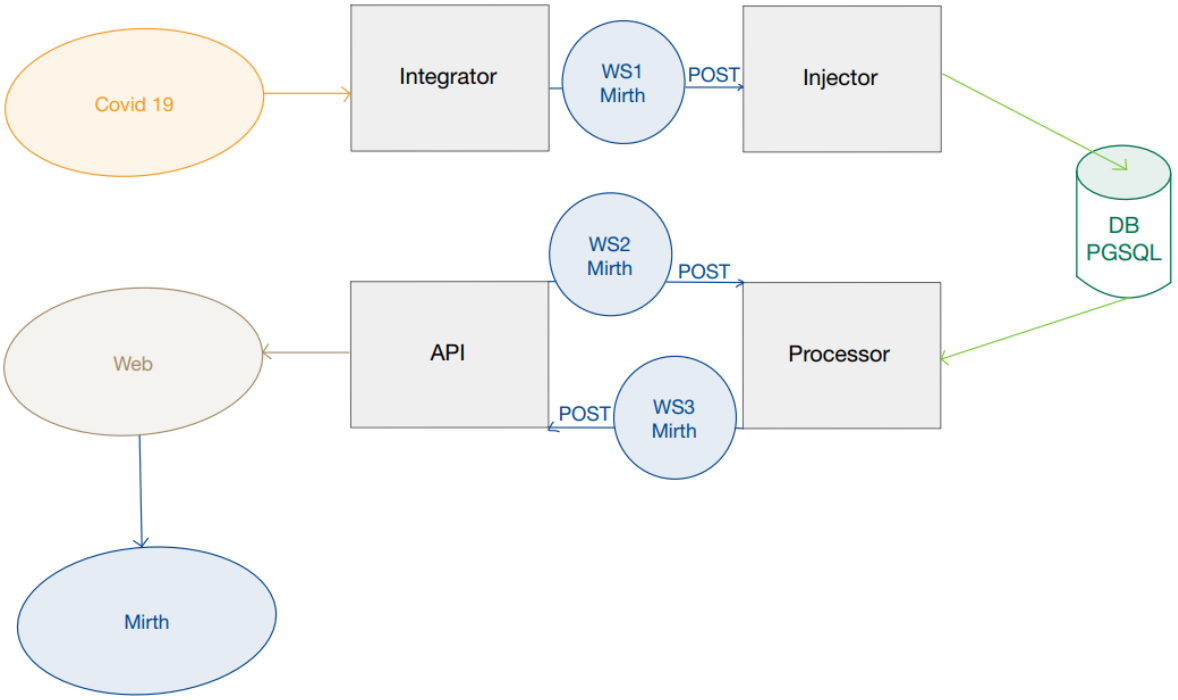
Maintenant que nous avons expliqué l'architecture que nous avons mis en place pour répondre à la demande du projet, nous allons pouvoir nous concentrer sur les différentes questions posées.

Question 1

Cette architecture est sympathique et répond au problème, mais n'aurait-il pas été aussi possible dans l'hypothèse où je ne connaîtrais pas les bénéfices du bus Kafka d'utiliser uniquement un EAI comme Mirth connect en tant que MiddleWare. Proposer une architecture qui répond au même besoin, mais uniquement avec l'EAI Mirth connect, attention ici je ne veux pas d'échange par fichiers sur le file système, mais uniquement l'utilisation de socket ou de web service REST

Si nous n'avions pas utilisé cette architecture, nous aurions pu directement utiliser l'EAI Mirth connect. En effet, avec mirth connect permet à la fois d'utiliser des sockets,autrement dit des connexions réseau qui permettent de transmettre des données en temps réel entre deux applications (en utilisant les protocoles de communication couramment utilisés tels que TCP et UDP). Mais aussi un web service REST, en effet Mirth Connect peut envoyer et recevoir des données via des requêtes HTTP.

Pour notre exemple, nous avons choisi de nous pencher sur l'utilisation de web services REST. Si nous faisons cela, nous allons obtenir l'architecture suivante :



De cette manière nous pouvons constater de nombreuses choses sur ce nouveaux schémas. En effet, nous avons à présent :

- *Integrator* : le Pr1 est maintenant directement connecté à un Web Service 1 Mirth grâce à la configuration proposé par l'EAI ;

- *WS1 Mirth* : s'occuper de faire un POST directement sur le bloc *Injector*, autrement dit sur le Cs1 ;

- *API* : le Pr2 est directement connecté à un Web Service 2 Mirth, de la même manière que pour le bloc *Integrator* ;

- *WS2 Mirth* : s'occuper de faire un POST directement sur le bloc *Processor*, autrement dit sur le Cs2 ;

- *Processor* : le Pr3 est directement connecté à un Web Service 3 Mirth, de la même manière que pour les blocs *Integrator* et *API* ;

- *WS3 Mirth* : s'occuper de faire un POST directement sur le bloc *Processor*, autrement dit sur le Cs3 ;

Question 2

Donnez votre avis sur les deux architectures celle proposé dans le TP et celle que vous proposerez, qu'elle est l'architecture la plus simple à mettre en place et à maintenir, comment et pourquoi ?

Comme nous venons de le voir dans la partie précédente, il y a différentes façons de pouvoir effectuer des communications entre plusieurs blocs de codes, ou web services. Ainsi, dans le cadre de ce travail, nous avons vu les API KafkaClient (que nous avons implémenté) mais aussi Mirth connect (dont nous avons parlé).

Si nous devions choisir entre ces deux méthodes de travail, nous pouvons affirmer que celle utilisant Kafka est bien plus pratique et maintenable dans le temps. En effet avec celle-ci, nous n'avons qu'une seule unité, alors qu'avec Mirth connect, nous en avons trois distinctes.

En résumé, dans un système Kafka, les producteurs publient des messages dans des topics, les brokers stockent ces messages et les distribuent aux consommateurs qui sont abonnés à ces topics. Les brokers travaillent ensemble pour gérer la distribution et la réplication des messages pour garantir la disponibilité et la tolérance aux pannes. Ainsi, avec Kafka nous pouvons facilement créer des brokers contenant des topics que nous pouvons orchestrer comme nous le souhaitons (nous n'avons pas besoin de mettre en place plusieurs Kafka).

De plus, le problème de Mirth connect et que si un de nos web service tombe en panne, l'ensemble de notre structure en subit les conséquences, il n'y a pas de topic qui peut prendre le relais comme c'est le cas avec Kafka. Par conséquent, l'un des points essentiels qui est d'assurer la pérennité d'une application n'est pas respecté. De cette façon, l'architecture implémentée pour ce travail, c'est-à-dire celle qui utilise Kafka est bien plus intéressante.

Question 3

Cette architecture, comme développée actuellement, ne propose pas de sécurité dans les échanges des messages sous Kafka, je vous demanderai donc de me présenter synthétiquement les différentes possibilités de sécurisation des échanges dans un bus Kafka dans un premier temps. Dans un second temps, vous choisirez un procédé possible parmi les différentes possibilités et exposerez clairement le principe de mise en place de celle-ci (plus de synthèse, mais approfondir le principe).

Pour sécuriser les échanges dans un bus Kafka, il existe plusieurs moyens. Parmi ceux-ci nous pouvons citer :

- *L'authentification des utilisateurs* : directement avec Kafka via SASL (*Simple Authentication and Security Layer*) et SSL (*Secure Socket Layer*). SASL peut être utilisé pour authentifier les utilisateurs via des mécanismes tels que PLAIN, SCRAM-SHA-256 et SCRAM-SHA-512. SSL peut être utilisé pour sécuriser les communications en chiffrant les données transmises entre les clients et les serveurs Kafka.

- *L'autorisation d'accès* : directement avec Kafka via les rôles ACL (*Access Control Lists*). Les ACL peuvent être utilisées pour contrôler l'accès aux topics et aux opérations (par exemple, lire, écrire, décrire) pour les utilisateurs et les groupes d'utilisateurs spécifiques.

- *Le chiffrement des données* : directement avec Kafka, en mouvement et en repos. Le chiffrement des données en mouvement peut être configuré en utilisant SSL pour sécuriser les communications entre les clients et les serveurs Kafka. Le chiffrement des données en repos peut être configuré en utilisant des disques chiffrés pour stocker les données sur les serveurs Kafka.

- *L'utilisation de proxy* : Il est possible de mettre en place un proxy d'accès pour faire office de point d'entrée pour les clients et de mettre en place des règles de sécurité pour les connexions entrantes, ou encore de mettre en place un VPN pour encadrer les communications entre les différents éléments du système.

Malgré ces différents moyens de sécuriser les échanges dans un bus Kafka, il est important de noter que la sécurisation d'un système peut être complexe. En effet, il est nécessaire de considérer tous les aspects de sécurité (authentification, autorisation, chiffrement, etc.) et de mettre en place des contrôles de sécurité adéquats pour protéger les données et les ressources de votre système.

Si nous voulons mettre en place une authentification au niveau des utilisateurs dans Kafka, nous pouvons utiliser le protocole SASL, protocole générique qui permet de définir plusieurs mécanismes d'authentification, tels que Kerberos, OAuth, LDAP, etc. Pour cela, voici les étapes générales pour mettre en place un système d'autorisation des utilisateurs avec SASL dans Kafka :

- Configuration du serveur Kafka : Pour utiliser SASL, nous devons activer le support SASL dans la configuration du serveur Kafka. Cela peut être fait en ajoutant les paramètres suivants à la configuration du serveur :

  security.inter.broker.protocol=SASL_PLAINTEXT
  sasl.mechanism.inter.broker.protocol=<MECHANISM>

Remarque : *<MECHANISM>* est le mécanisme SASL que nous souhaitons utiliser (par exemple, Kerberos, OAuth, etc.).

- Configuration du client : nous devons aussi configurer le client pour qu'il utilise SASL. Cela peut être fait en ajoutant les paramètres suivants à la configuration du client :

  security.protocol=SASL_PLAINTEXT sasl.mechanism=<MECHANISM>

- Configuration du mécanisme SASL : nous devons configurer le mécanisme SASL que nous avons choisi. Les détails de la configuration dépendent du mécanisme que vous utilisez. Par exemple, si nous utilisons Kerberos, nous devons configurer le client et le serveur pour utiliser une instance de Kerberos pour l'authentification.
- Configuration des ACL : Une fois le support SASL activé et configuré, nous pouvons définir des rôles ACL pour contrôler les autorisations d'accès aux topics. Les ACL peuvent être définies en utilisant des groupes d'utilisateurs et des rôles pour contrôler les autorisations d'accès aux topics.

En bref, pour mettre en place un système d'autorisation des utilisateurs avec SASL dans Kafka, nous devons activer et configurer le support SASL dans le serveur et le client, configurer le mécanisme SASL choisi, et définir les ACL pour contrôler les autorisations d'accès aux topics.

Conclusion

Pour conclure sur ce projet, nous pouvons dire qu'il a été très enrichissant car il nous a permis de comprendre les mécanismes de Kafka dans une utilisation concrète via plusieurs acteurs. De plus, ce projet nous a donné l'occasion de manipuler plus en profondeur l'outil Docker que nous avions peu utilisé jusqu'à présent. Ainsi, nous avons aujourd'hui une vision plus pratique du comportement que peut avoir un outil tel que Kafka sur un environnement spécifique.

En plus de son côté pratique, ce TP nous a permis de nous rendre compte que chaque outil à sa propre utilisation. En effet, nous avons vu que même si nous pouvions utiliser Mirth Connect pour remplacer le rôle de Kafka, cela n'aurait pas été très approprié pour de nombreuses raisons. En revanche, Mirth connect reste une très bonne alternative, notamment pour exporter des données, comme ce fut demandé dans ce projet.

Pour ce qui est de Kafka, nous avons pu être témoin de sa facilité de prise en main et d'exécution afin de faire communiquer nos divers blocs de codes, alors que comme nous le disions, cela aurait été compliqué avec un EIP. En effet, nous avons pu ordonnancer nos topics comme nous l'avons voulu de façon à répondre à la demande.

Bien entendu, des améliorations sont envisageables sur ce projet. Parmi celles-ci, nous pouvons citer la mise en place de sécurisation au niveau du bus Kafka. Nous l'avons étudié, il existe de nombreuses manières pour faire cela, comme la mise en place d'autorisations, de chiffrement ou encore de proxy.