Optimisation du routage de données

10/03/2023
MIAGE Master 2

Tuteur :
Adrien SALES

Etudiants :
Antoine GINDRE
Kenneth WINCHESTER

Table des matières

1. Remerciements 3
2. Introduction 3
3. Contexte 4
   a. Consommer les données de l'API 4
   b. Alimentation d'un fichier 4
   c. Intégration Kafka 4
   d. Stockage et visualisation de la donnée 4
4. Organisation 5
5. Technologies utilisées 5
6. Le projet 10
7. Résultats attendus 18
8. Améliorations futures 18
9. Conclusion 18
10. Références 19
11. Annexes 20

1. Remerciements

Nous souhaitons adresser nos remerciements les plus sincères à toutes les personnes qui nous ont apportés leur aide pour mener à bien ce projet et à l'élaboration du rapport.

Nous tenons à remercier Adrien Sales qui, en tant que tuteur encadrant, s'est toujours montré à l'écoute tout au long de la réalisation de ce projet. Ainsi nous le remercions pour l'aide et tout le temps qu'il a souhaité nous consacrer.

Nos remerciements s'étendent également à Madame Michèle BARRÉ qui nous a offert son support technique et expertise lors des différentes réunions.

2. Introduction

L'Office des Postes et des Télécommunications de Nouvelle-Calédonie (OPT-NC) est un établissement public à caractère industriel et commercial (EPIC).

Organisée autour de 4 métiers (le postal, les télécommunications, les services financiers et le réseau de distribution), l'OPT-NC est présente sur l'ensemble de la Nouvelle-Calédonie via ses agences postales et techniques.

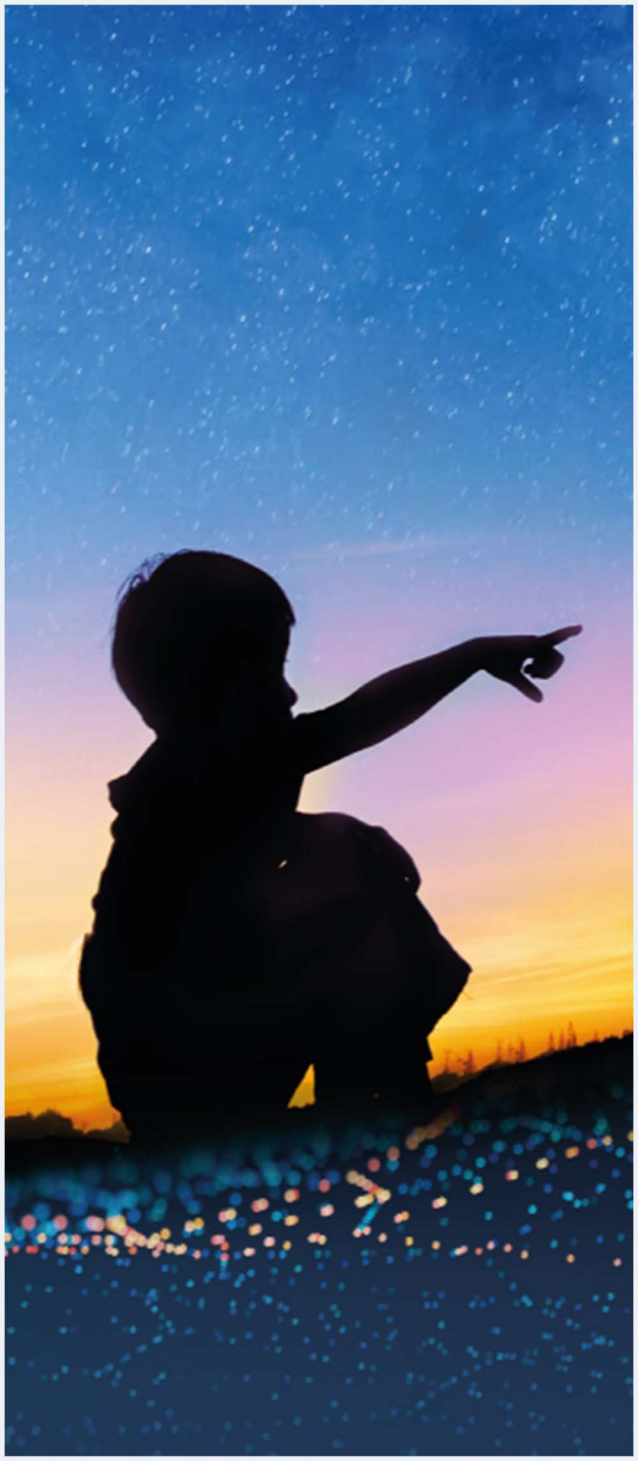
Par l'intermédiaire de ses investissements, l'office joue un rôle important dans l'aménagement, le développement et la construction du pays.

Résolument tournée vers l'avenir, l'office se réinvente pour que ses métiers puissent bénéficier de technologies et services qualité.

La digitalisation de ses services est un axe d'amélioration majeur de l'office afin de pouvoir offrir un meilleur service à ses clients.

La DSI (Direction des Systèmes d'Information) est de fait un acteur important dans la démarche d'innovation de l'office.

Le bureau Génie Logiciel Inter Applicatif (GLIA), constitué de M. Adrien SALES et son équipe, occupe un rôle important en proposant des solutions permettant de copier, d'intégrer et d'enrichir les données entre les différents systèmes de l'OPT-NC, mais aussi depuis et vers l'extérieur de l'office.



3. Contexte

Les API (Application Programming Interface) mise en place et gérées par le GLIA sont le pivot central pour un partage efficace des données. Les données des temps d'attente de l'ensemble des agences de l'OPT-NC ne font pas exception.

L'OPT utilise Kafka pour partager et transformer un volume de données toujours plus important et toujours plus vivantes entre ses métiers. La DSI souhaite maintenant mettre en place différentes manières de consommer ces données à moindre effort, puis de les valoriser via des solutions dédiées à l'exploration et la visualisation de données. Et ainsi permettre de développer les technologies liées à l'IA et au ML afin de disposer de chaînes efficaces de « Realtime Analytics ».

Les données visées par ce projet seront celles générées par une API public, mise à disposition par l'OPT-NC, permettant de récupérer les temps d'attente des différentes agences en Nouvelle-Calédonie.

L'objectif de ce projet est donc de mettre en place une route Camel permettant de récupérer, toutes les 5 minutes, les temps d'attente en agence. Puis, pousser les données reçues dans un topic Kafka via un Producer et enfin consommer les données grâce à une solution de data visualisation tel que OpenSearch.

Missions

a. *Consommer les données de l'API*

Mise en place d'une route Camel permettant de récupérer, toutes les 5 minutes, les temps d'attente en agence.

b. *Alimentation d'un fichier*

Via la route Camel précédente, pousser les données automatiquement dans un fichier json.

c. *Intégration Kafka*

Grâce à un Poll régulier de 5 minutes et toujours en utilisant la route Camel, pousser les données dans un topic Kafka dédié avec l'aide d'un Producer.

d. *Stockage et visualisation de la donnée*

Consommer les données grâce à une solution de data visualisation tel que OpenSearch.

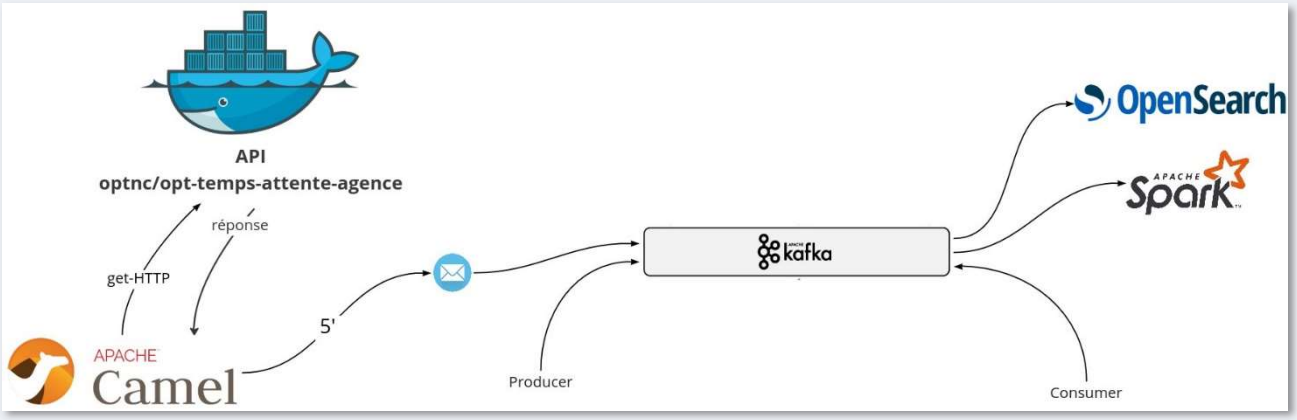


Figure 1 : description de l'architecture cible

4. Organisation

Afin d'atteindre les objectifs fixés dans la partie précédente, le groupe c'est organisé de la manière suivante :

- Une réunion de cadrage du projet a eu lieu avec Adrien SALES : celle-ci nous a permis de définir le périmètre du projet, les attentes et de définir les contraintes technologiques.
- La mise en place de l'outil de gestion de version Git avec un dépôt GitHub : cela a permis de suivre l'avancement du projet au niveau des différentes étapes et mission ainsi que de faciliter les échanges et les retours.
- Des réunions d'avancement du projet ont eu lieu à chaque avancement majeur, notable ou nécessitant une expérience technique plus pointue. Ces réunions se sont déroulées en faisant une démonstration du travail réalisé depuis le dernier rassemblement, suivi d'un échange avec le tuteur du projet. Ceci afin d'orienter la suite du travail dans des conditions optimales (méthode LEAN).

5. Technologies utilisées

La réalisation de ce projet a nécessité une montée en compétence sur plusieurs technologies, qui sont présentées ci-dessous :

5.1 L'API temps d'attentes en agence de l'OPT-NC

L'OPT-NC met à disposition du public un service permettant de connaître les temps d'attente en cours dans les agences OPT. Ce service est conteneurisé dans une image docker sous forme d'API.

Lorsqu'elle est interrogée, elle renvoie une réponse au format JSON contenant un certain nombre d'informations.


```
[
  {
    "idAgence": 0,
    "designation": "string",
    "realMaxWaitingTimeMs": 0,
    "coordonneeX": 0,
    "coordonneeY": 0,
    "coordonneeXPrecise": 0,
    "coordonneeYPrecise": 0,
    "position": {
      "lat": 0,
      "lon": 0
    },
    "commune": "string",
    "type": "string",
    "codeESirius": "string",
    "codePostal": "string",
    "lieuDitOuTribu": "string",
    "localite": "string",
    "idRefloc": "string",
    "codePostalRefloc": "string",
    "localiteRefloc": "string"
  }
]
```


Figure 2 : schéma de données reçues en retour d'un appel vers l'API des temps d'attente

5.2 Camel-JBang

JBang est un outil open-source qui permet de créer et d'exécuter rapidement des scripts Java avec une expérience de développement fluide. Il offre une alternative aux environnements de développement Java lourds tels que les IDE, en permettant aux développeurs de travailler sur des scripts Java en mode ligne de commande avec un flux de travail efficace.

JBang permet de travailler rapidement et facilement avec Java sans configuration de projet ni installation de dépendances et permettant ainsi aux développeurs de se concentrer sur la tâche à accomplir plutôt que sur les détails de la configuration.

JBang peut-être complété par le module Camel JBang.

Apache Camel est un Framework open-source d'intégration d'entreprise basé sur les messages. Il fournit une solution complète et extensible pour l'intégration de différents systèmes hétérogènes en utilisant une variété de protocoles et de technologies.
Camel est conçu pour aider à résoudre les problèmes courants de l'intégration de systèmes, tels que la transformation de données, le routage de messages, la mise en file d'attente, la détection de pannes, l'orchestration de services et bien plus encore.

Camel prend en charge de nombreux protocoles et technologies, notamment HTTP(S), FTP(S), JMS, REST, SOAP, MQTT, AMQP, Apache Kafka, Camel Netty, CXF, et plus encore. De plus, il offre un support pour de nombreux formats de données courants tels que XML, JSON, CSV, YAML, et d'autres formats personnalisés.

L'architecture de Camel est basée sur un système de routage de messages qui permet de connecter différents points de terminaison, tels que des bases de données, des files d'attente, des services Web, etc. L'utilisateur peut configurer les routes en utilisant une variété de langages, notamment XML, Java DSL[1], Scala DSL, Groovy DSL et Kotlin DSL.

La terminologie et l'architecture de base de Camel :

- Le **message** contient les données qui sont transférées vers une route. Chaque message a un identifiant unique et est construit à partir d'un corps, d'en-têtes et de pièces jointes.
- L'**échange (exchange)** est le conteneur d'un message et est créé lorsqu'un consommateur reçoit un message au cours du processus de routage. L'échange permet différents types d'interaction entre les systèmes - il peut définir un message à sens unique ou un message de demande-réponse.
- Le **Endpoint** est un canal par lequel un système peut recevoir ou envoyer un message. Il peut faire référence à une URI de service web, à une URI de file d'attente, à un fichier, à une adresse e-mail, etc.

Le **composant Camel (Camel component)** agit en tant qu'usine[2] de Endpoint. En termes simples, les composants de Camel offrent une interface à différentes technologies en utilisant la même approche et la même syntaxe. Camel prend déjà en charge de nombreux composants dans ses DSL pour presque toutes les technologies possibles, mais il offre également la possibilité d'écrire des composants personnalisés.

[1] DSL (Domain Specific Language) : langage de programmation spécifiquement conçu pour résoudre des problèmes dans un domaine d'application particulier.

[2] Usine (factory) : dans un contexte informatique c'est objet ou un ensemble d'objets qui est responsable de la création d'autres objets.

5.3 Apache Kafka

Kafka est une plateforme open-source de streaming de données distribuée, conçue pour gérer efficacement des volumes importants de données en temps réel. Elle est souvent utilisée pour la création de pipelines de données, l'analyse en temps réel, le traitement de flux, la surveillance et le traitement des logs.

Kafka a été initialement développé par LinkedIn et est maintenant géré par la fondation Apache. Il est écrit en Java et offre des API pour plusieurs langages de programmation tels que Java, Scala, Python et C++.

Kafka est une plateforme de streaming de données distribuée dont les principales fonctionnalités sont les suivantes :

- Haute performance : Kafka est conçu pour fournir une latence très faible et une haute capacité de traitement des messages, même avec de gros volumes de données.
- Scalabilité : Kafka est conçu pour être facilement extensible afin de pouvoir gérer des flux de données de grande taille.
- Tolérance aux pannes : Kafka est capable de gérer les pannes et de maintenir la disponibilité des données même en cas de défaillance d'un ou plusieurs nœuds.
- Flexibilité : Kafka est utilisé dans une variété de cas d'utilisation allant de l'ingestion de données, à la surveillance en temps réel, en passant par le traitement des données, le stockage de logs, etc.
- Traitement des flux : Kafka permet la transformation et le traitement des données en temps réel grâce à son architecture basée sur des flux.

Kafka utilise un modèle de publication/abonnement pour la diffusion des messages entre les producteurs et les consommateurs. Les messages sont stockés dans des topics, qui sont des canaux de communication entre les différents composants du système. Les producteurs écrivent des messages dans un topic et les consommateurs les lisent à partir de ce topic. Kafka peut stocker de grands volumes de données et permet de stocker les messages pendant une période déterminée, en fonction de la configuration.

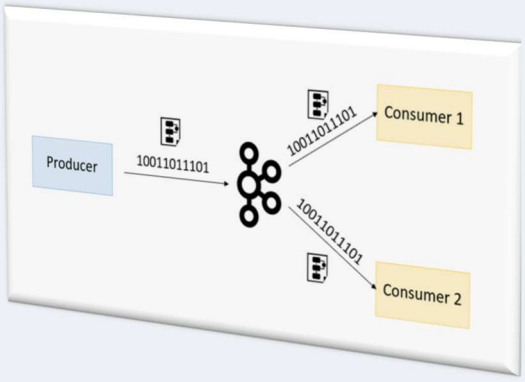


Figure 3 : Kafka

5.4 Docker

Docker est une plateforme qui permet de créer, exécuter et gérer des applications dans des conteneurs logiciels. Il est basé sur la technologie de virtualisation de système d'exploitation (OS-level virtualization), qui permet d'exécuter plusieurs instances isolées d'un système d'exploitation sur une même machine physique.

Les conteneurs Docker encapsulent les applications et leurs dépendances, ce qui les rend portables, reproductibles et isolées du reste de l'environnement système. Les applications peuvent être exécutées de manière cohérente et prévisible, quel que soit l'environnement de déploiement, que ce soit un ordinateur de développement, un serveur de production ou un environnement cloud (on parle alors d'application cloud natives).

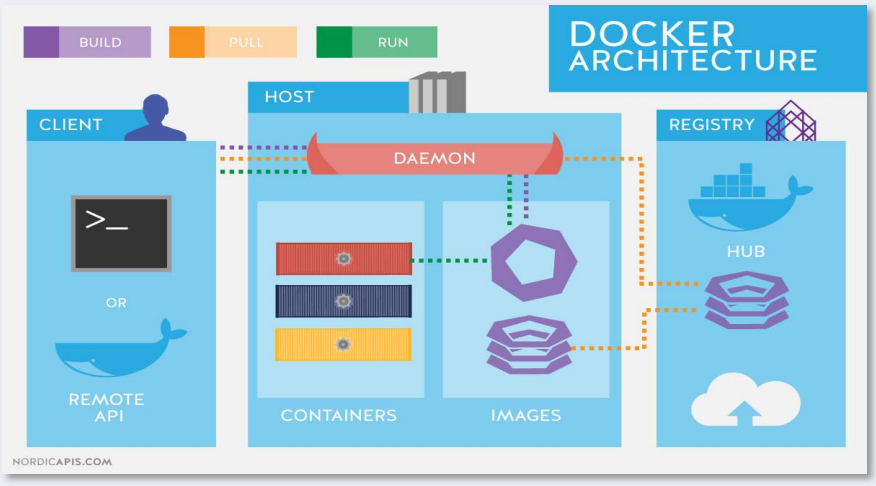
Voici quelques concepts clés de la conteneurisation :

- Image : un modèle de base pour créer des conteneurs. Il contient tous les fichiers et les configurations nécessaires pour exécuter une application dans un environnement isolé. Les images sont créées à partir de fichiers de configuration appelés "Dockerfiles", qui décrivent les étapes nécessaires pour construire l'image.
- Conteneur : une instance en cours d'exécution d'une image Docker. Les conteneurs sont créés à partir d'une image Docker et peuvent être démarrés, arrêtés, supprimés et gérés à partir de la ligne de commande ou à l'aide d'outils de gestion de conteneurs tels que Docker-Compose et Kubernetes.
- Registre : un référentiel centralisé où les images Docker sont stockées et partagées. Docker Hub est un registre public où les développeurs peuvent trouver et partager des images Docker. Les entreprises peuvent également utiliser des registres privés pour stocker leurs propres images Docker.
- Dockerfile : un fichier de configuration qui décrit les étapes nécessaires pour construire une image Docker. Il contient des instructions pour installer les dépendances, copier les fichiers de l'application, configurer l'environnement, etc.

Docker offre de nombreux avantages pour les développeurs et les opérateurs :

- Isolation : les applications sont isolées du reste du système d'exploitation, ce qui les rend plus fiables et plus sécurisées.
- Portabilité : les conteneurs sont portables et peuvent être exécutés sur n'importe quelle machine ou environnement de déploiement, ce qui simplifie la mise en production.
- Rapidité : Docker permet de déployer des applications plus rapidement en utilisant des processus automatisés.
- Flexibilité : Docker offre une grande flexibilité pour lancer et gérer des applications, en permettant des déploiements monolithiques ou micro-services.

Docker est maintenant une plateforme populaire et puissante pour créer, exécuter et gérer des applications dans des conteneurs. Il offre une isolation, une portabilité et une rapidité accrues, ce qui simplifie la gestion des environnements d'exécution d'applications et accélère les cycles de développement.



Docker-compose est une surcouche de Docker qui permet de définir et de gérer des applications Docker multi-conteneurs. A l'aide de cet outil, plusieurs conteneurs peuvent être exécutés en parallèle dans le même réseau docker et communiquer entre eux en utilisant leur nom de conteneur. Ceci est rendu possible grâce à un script YAML qui spécifie les conteneurs, les volumes, les réseaux et les variables d'environnement nécessaires à l'exécution. Ce script simplifie de manière notable la configuration nécessaire pour imbriquer tous les conteneurs.

5.4 OpenSearch

OpenSearch est un moteur de recherche et d'analyse de données distribué open source, basé sur Elasticsearch. OpenSearch a été développé et lancé en avril 2021 par AWS qui souhaitait créer une alternative open source à Elasticsearch, et ce après que la licence de ce dernier soit passée de l'open source à une licence propriétaire.

OpenSearch a été conçu pour offrir les mêmes fonctionnalités que Elasticsearch, avec des améliorations de performance, de sécurité et de facilité d'utilisation. OpenSearch prend en charge la recherche de texte intégral, la recherche géospatiale, la recherche de séquences temporelles, la recherche de graphe, la recherche de métriques et l'analyse de logs.

Voici quelques-unes des fonctionnalités clés d'OpenSearch :

- Recherche de texte intégral : OpenSearch utilise l'algorithme de recherche de texte intégral Lucene pour offrir une recherche rapide et précise de tout type de contenu.
- Recherche géospatiale : OpenSearch prend en charge la recherche et l'analyse de données géospatiales, permettant de trouver des lieux, des points d'intérêt et des itinéraires.
- Recherche de séquences temporelles : OpenSearch offre des fonctionnalités avancées pour la recherche et l'analyse de données temporelles, telles que les agrégations temporelles, les intervalles de temps et les histogrammes.
- Recherche de graphe : OpenSearch permet de trouver des relations entre les données en utilisant des algorithmes de recherche de graphe avancés.
- Recherche de métriques : OpenSearch prend en charge la recherche et l'analyse de données de métriques, telles que les mesures de performances, les événements de système et les indicateurs de santé.
- Analyse de logs : OpenSearch peut être utilisé pour analyser les logs système et les logs d'application, permettant de trouver des erreurs, des problèmes de performance et des tendances.

OpenSearch est distribué sous la licence Apache 2.0, ce qui signifie qu'il est libre d'utilisation, de modification et de distribution, et qu'il peut être intégré à des projets open source et propriétaires. OpenSearch est également conçu pour être facilement extensible et configurable, permettant aux développeurs de personnaliser le moteur de recherche pour répondre à leurs besoins spécifiques.

OpenSearch est souvent utilisé dans des applications web, des systèmes de traitement de données volumineuses, des systèmes d'analyse de logs et des applications de surveillance et de sécurité. Il est compatible avec de nombreux langages de programmation, des bibliothèques et des outils tiers, tels que Python, Java, Ruby, Logstash, Beats, Kafka, et bien d'autres encore.

5.5 Zookeeper

Apache ZooKeeper est un service de coordination distribuée open-source qui permet aux applications distribuées de coordonner et de gérer les tâches entre différents nœuds dans un cluster ou un réseau. Il est souvent utilisé pour les systèmes distribués tels que les bases de données distribuées, les systèmes de traitement de flux de données, les systèmes de messagerie et les systèmes de verrouillage de ressources.

Kafka utilise Zookeeper pour stocker de nombreuses informations en tant que métadonnées sur les partitions et les brokers. Et aussi pour élire un broker en tant que contrôleur Kafka.

Cette brique est donc souvent utilisée en complément d'Apache Kafka afin de gérer la configuration et la synchronisation des nœuds du cluster. Pour ce faire il stocke les informations relatives à l'état du cluster, les différentes configurations et les informations relatives aux topics et partitions implémentées.

6. Le projet

En tout premier lieu, il est important de préciser que ce projet est orienté open source, micro-service et interopérable entre différents systèmes.

6.1 API des temps d'attente

La première étape du projet a été déployer le conteneur Docker de l'api publique optnc/opt-temps-attente-agences-api, qui est mise à disposition par l'OPT-NC et de se familiariser avec.

Pour ce faire, le premier outil qui a été utilisé est httpie[3] afin de nous permettre de facilement prendre connaissance de la structure des données disponibles.

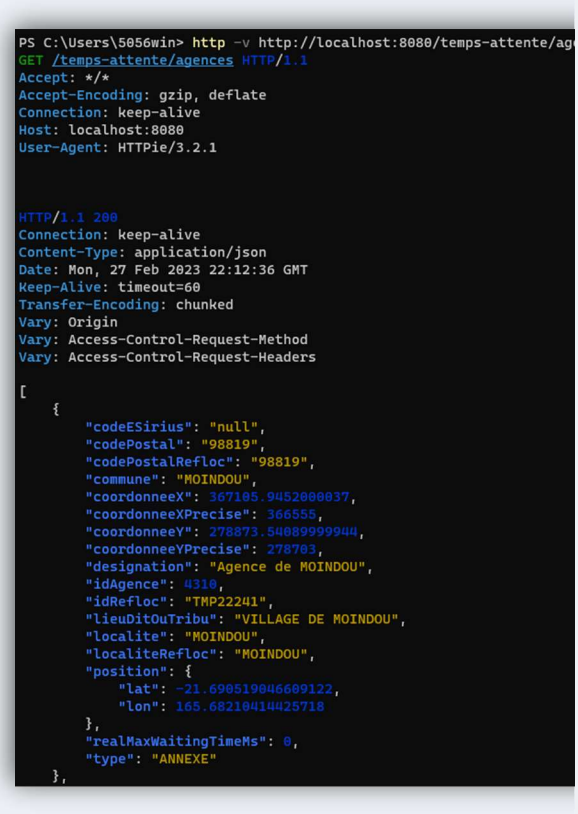


Figure 4 : Données API

[3] Outil moderne de requêtes HTTP permettant de tester des API de manière simple et moderne - https://chat.openai.com/chat

La seconde étape a été de mettre en place une solution permettant de pouvoir rapidement tester la disponibilité de l'api. Cela a été réalisé par l'intermédiaire d'un script PowerShell qui, via un Heath test, vérifie que le code retour de l'api est bien 200, puis valide que les données sont bien accessibles.

```powershell
### Construction des URL de tests
$actuatorUri = "http://localhost:8080/actuator"
$healthUri = $actuatorUri +"/health"
$infoUri = $actuatorUri +"/info"

### HealtTestURI
Try {
    $healthTestUri =  Invoke-WebRequest -Uri $healthUri -UseBasicParsing
    $healthStatusCode = $healthTestUri.StatusCode

    If ($healthStatusCode -eq "200"){
    Write-Host "API HealthTest OK :" $healthStatusCode
    }Else {
        Write-Host "API HealthTest KO :" $healthStatusCode
    }

}Catch{
    $healthStatusCode = $_.Exception.Response.StatusCode.value__
}

### InfoTestURI
Try {
    $infoTestUri =  Invoke-WebRequest -Uri $infoUri -UseBasicParsing
    $infoStatusCode = $healthTestUri.StatusCode

    If ($infoStatusCode -eq "200"){
        Write-Host "API InfoTest OK :" $infoStatusCode
    }Else {
        Write-Host "API InfoTest KO :" $infoStatusCode
    }

}Catch{
    $infoStatusCode = $_.Exception.Response.StatusCode.value__
}


### Test de récupération du csv de toutes les agences
If (($healthTestUri.StatusCode -eq "200") -and  ($healthTestUri.StatusCode -eq "200")) {
    $webdata = Invoke-WebRequest -Uri http://localhost:8080/temps-attente/agences -UseBasicParsing

    write-host "Récupération des données des agences en cours..."
    $releases = ConvertFrom-Json $webdata.Content
    write-host "Récupération des données des agences terminée."

    ### Export des données dans un fichier CSV
    if ($releases) {
       $releases | Export-Csv .\agences.csv -delimiter ';' -NoTypeInformation
        $file = Get-ChildItem agences.csv
        Write-Host "Le fichier" $file.Name "a été créé dans le répertoire courant."
    }

    Write-Host "L'API contient" $releases.Count "agences."

    ### Test pour une agence
    $4161 = Invoke-WebRequest -Uri http://localhost:8080/temps-attente/agence/4161 -UseBasicParsing
    Write-host "Détail de l'agence 4161" -ForegroundColor Green
    $4161
}
```

Figure 5 : Script PowerShell

Une fois le bon fonctionnement des requêtes vers l'API confirmés, la partie routage avec Camel peut être explorée.

6.2 Première route Camel

La mise en place d'une route Camel fonctionnelle est l'élément clé de ce projet. Sans une bonne implémentation de cette route le projet ne peut plus avancer.

Par conséquent, cette mission a nécessité un travail important d'autoformation sur les technologies JBang et Camel.

Deux routes de format différent ont été développées :

- Une route JBang en java

```java
///usr/bin/env jbang "$0" "$@" ; exit $?
// Use org.apache.camel:camel-bom:3.19.0@pom in order to avoid repeating the version in each line
//DEPS org.apache.camel:camel-core:3.19.0
//DEPS org.apache.camel:camel-main:3.19.0
//DEPS org.apache.camel:camel-stream:3.19.0
//DEPS org.apache.camel:camel-http:3.19.0
//DEPS org.apache.camel:camel-jackson:3.19.0

import org.apache.camel.*;
import org.apache.camel.builder.*;
import org.apache.camel.main.*;
import org.apache.camel.model.dataformat.JsonLibrary;

import static java.lang.System.*;

public class TempsAttenteAgenceCamel {

    public static void main(String... args) throws Exception {
        out.println("Hello World");
        out.println("Running camel route...");

        Main main = new Main();

        // Configuration de la route
        main.configure().addRoutesBuilder(new RouteBuilder() {
            public void configure() throws Exception {
                from("timer://foo?period=5000")
                .to("http://localhost:8080/temps-attente/agences")
                .unmarshal().json(JsonLibrary.Jackson)
                .process(exchange -> {
                    // On print la réponse
                    out.println(exchange.getIn().getBody());
                });

            }
        });
        main.run();
    }
}
```

Figure 6 : JBang Java

- Une route Camel-JBang en xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- camel-k: language=xml -->

<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://camel.apache.org/schema/spring"
        xsi:schemaLocation="
            http://camel.apache.org/schema/spring
            https://camel.apache.org/schema/spring/camel-spring.xsd">

    <!-- Write your routes here, for example: -->
    <route id="http-get-route">

        <!-- Ordonnance de l'exécution de la route toutes les 5 minutes -->
        <from uri="timer:http-get-timer??period={{time:300000}}"/>


        <!-- URL source de l'API avec la méthode GET -->
        <to uri="http://localhost:8080/temps-attente/agences?httpMethod=GET"/>

        <!-- Déserialisation des données au format JSON -->
        <unmarshal>
            <json library="Jackson" prettyPrint="true" />
        </unmarshal>

        <!-- Affichage du JSON dans le terminal -->
        <log message="Received data: ${body}"/>

        <!-- JSON -->
        <!-- Conversion des données au format JSON -->
        <marshal>
            <json library="Jackson" prettyPrint="true" />
        </marshal>
        <to uri="file:./data?fileExist=Append&amp;fileName=temps-attente.json"/>

        <!-- Affichage du JSON dans le terminal -->
        <log message="Received data: ${body}"/>

    </route>
</routes>
```

Figure 7 : Camel-JBang XML

Après présentation de ces deux options au tuteur, le choix de ce dernier s'est orienté vers le format XML par souci d'interopérabilité, de simplicité d'implémentation, de maintenance et d'utilisation.

La route xml réalise les actions suivantes :

1. Ordonnancement de l'exécution de la route toutes les 5 minutes via la balise <from> et le composant Camel Timer.
2. Connection à l'API REST temps-attente avec la méthode GET via la première balise <to>
3. Conversion des données reçu en un objet Java (désérialisation), pour qu'il soit utilisable par les processus Camel, via la librairie Jackson
4. Conversion des données traitées au format JSON pour envoie vers la sortie via la balise <marshal>
5. Ajout des données dans un fichier local JSON via la balise <to> et le composant Camel File.

L'exécution de cette route se fait via la commande *« camel run <nomFichier>.xml »,* qui doit être exécutée depuis l'emplacement du fichier xml.

La prochaine étape du projet est d'envoyer les données vers un conteneur Kafka, ce qui a nécessité de travailler sur le déploiement d'environnement multi conteneur.

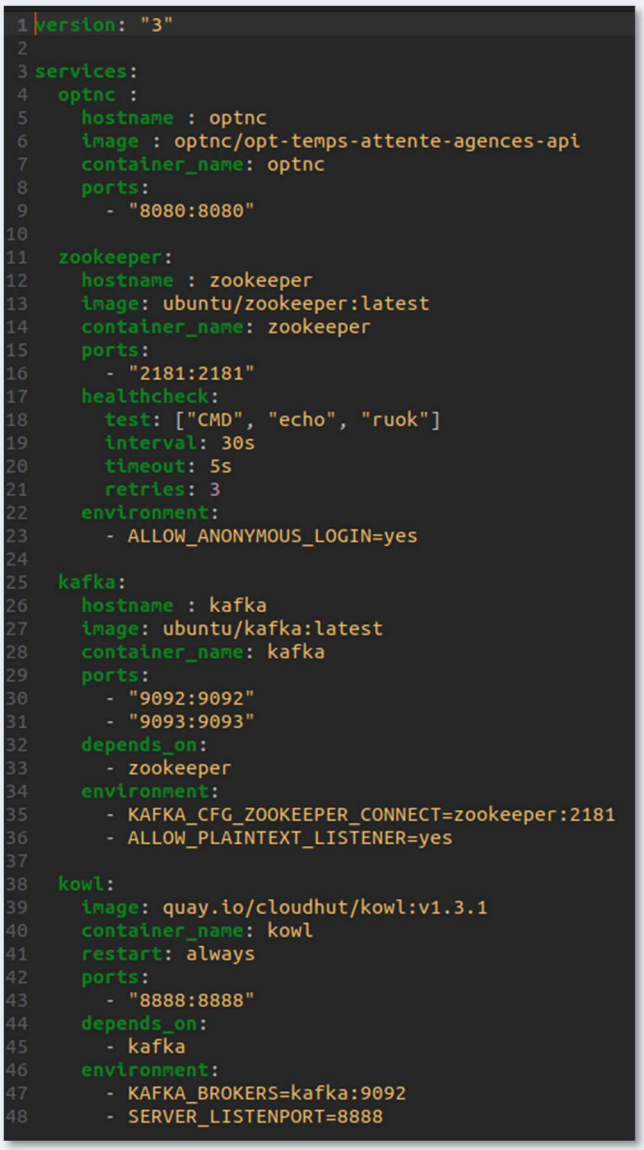
6.3 Mise en place de l'environnement Docker

Afin de faciliter le déploiement et la communication entre les différents conteneurs Docker nécessaire à la mise en place de la route Camel finale, un travail important a dû être mené sur l'outil Docker.

Cela a permis de réaliser un script Yaml, permettant de gérer la configuration des différents éléments et ainsi de gérer les éventuels conflits de port utilisé (ex : API temps-attente et Kowl qui par défaut utilise le même port).

Le script mis en place effectue les opérations suivantes

1. Récupération des dernières images des conteneurs utilisés
2. Configuration systèmes de chaque conteneur
3. Lancement de chaque conteneur avec un nom et un port spécifique
4. Ordonnancement du lancement des conteneurs pour s'assurer du bon fonctionnement (zookeeper – puis kafka – puis kowl)



```yaml
version: "3"

services:
  optnc :
    hostname : optnc
    image : optnc/opt-temps-attente-agences-api
    container_name: optnc
    ports:
      - "8080:8080"

  zookeeper:
    hostname : zookeeper
    image: ubuntu/zookeeper:latest
    container_name: zookeeper
    ports:
      - "2181:2181"
    healthcheck:
      test: ["CMD", "echo", "ruok"]
      interval: 30s
      timeout: 5s
      retries: 3
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes

  kafka:
    hostname : kafka
    image: ubuntu/kafka:latest
    container_name: kafka
    ports:
      - "9092:9092"
      - "9093:9093"
    depends_on:
      - zookeeper
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes

  kowl:
    image: quay.io/cloudhut/kowl:v1.3.1
    container_name: kowl
    restart: always
    ports:
      - "8888:8888"
    depends_on:
      - kafka
    environment:
      - KAFKA_BROKERS=kafka:9092
      - SERVER_LISTENPORT=8888
```

Figure 8 : Script Yaml

Une fois les différents conteneur opérationnels, l'apprentissage des bases de Kafka a pu commencer.

6.4 Kafka et nouvelle route Camel

L'architecture Kafka implémentée à l'OPT-NC intègre le module ZooKeeper, visible dans le fichier Yaml précédemment présenté.

En complément nous avons déployé l'outil open source Kowl. Ce dernier permet de faciliter le débogage, la surveillance et la gestion des clusters Kafka. Pour ce faire il est interconnecté avec les services Kafka et intègre une interface graphique intuitive permettant d'interagir avec les différents composant de Kafka comme les topics, les brokers, les consommateurs ou encore les producteurs.

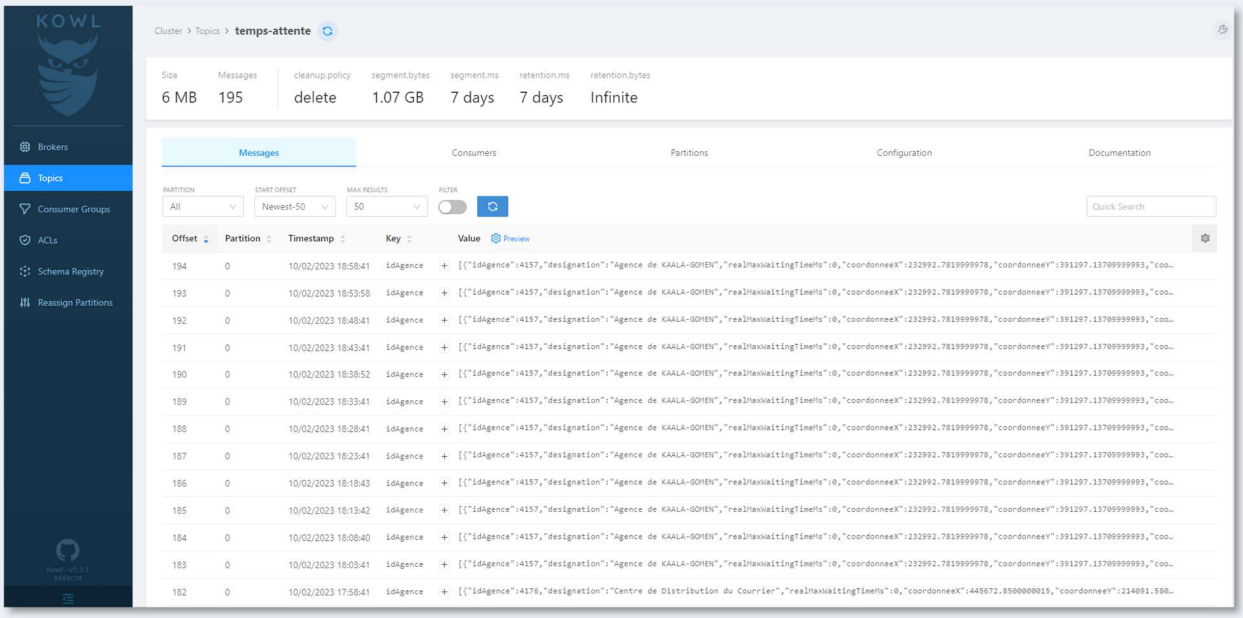


Figure 9 : interface Kowl

Pour que Kafka puisse intégrer des données, il doit avoir un serveur de messagerie de paramétré (broker[4]). Celui-ci intègre un emplacement (topic[5]) qui permettra d'y stocker les messages envoyés par les producteurs, chaque message étant identifié par une clé. Puis les messages seront mis à disposition des consommateurs.

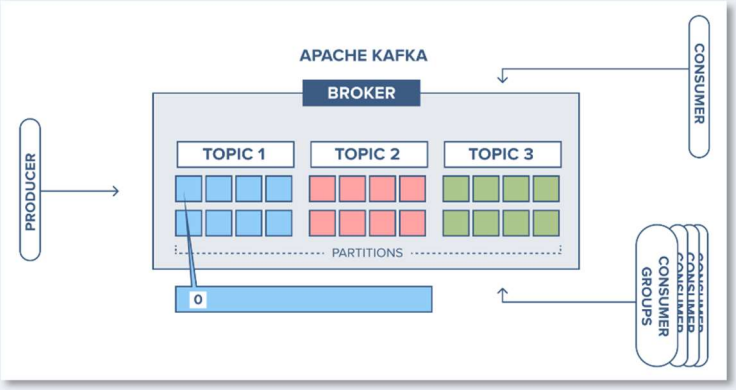


Figure 10 : flux de message Kafka

La mise en place de cette configuration est possible soit :

- Via les fonctions natives intégrées à Kafka
- Ou via l'utilitaire en ligne de commande Kafkacat, plus léger et flexible, qui publie et consomme des messages Kafka.

À la suite de nos recherches et monté en compétences, nous sommes parvenus à mettre en place un topic Kafka fonctionnel nommé *« temps_attentes ».*

Une fois le topic implémenté, nous avons alors entrepris de retravailler sur la route XML afin que le flux de données ne soit plus orienté en sortie vers un fichier JSON, mais vers le topic Kafka.

4 Broker : serveur Kafka qui gère les messages d'un ou plusieurs topics.

5 Topic : flux de messages organisés en catégories et stockés dans des partitions pour permettre une distribution efficace et une haute disponibilité.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- camel-k: language=xml -->

<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://camel.apache.org/schema/spring"
        xsi:schemaLocation="
            http://camel.apache.org/schema/spring
            https://camel.apache.org/schema/spring/camel-spring.xsd">

    <!-- Write your routes here, for example: -->
    <route id="http-get-route">

        <!-- Ordonnance de l'exécution de la route toutes les 5 minutes -->
        <from uri="timer:http-get-timer??period={{time:300000}}"/>

        <!-- URL source de l'API avec la méthode GET -->
        <to uri="http://localhost:8080/temps-attente/agences?httpMethod=GET"/>

        <!-- Déserialisation des données au format JSON -->
        <unmarshal>
            <json library="Jackson" prettyPrint="false" />
        </unmarshal>

        <!-- JSON -->
        <!-- Conversion des données au format JSON -->
        <marshal>
            <json library="Jackson" prettyPrint="false" />
        </marshal>

        <!-- URL de destination Kafka et paramétrage de la clé -->
        <to uri="kafka:temps-attente?brokers=localhost:9092&amp;key=idAgence"/>

    </route>
</routes>
```

Figure 11 : nouvelle route Camel

Au niveau des étapes de sérialisation/désérialisation des données l'option *prettyPrint* a été paramétrée à *false* afin que les données présentées à Kafka soient formatté sur une seule ligne. Cela est un prérequis nécessaire pour que Kafka puisse traitées les données.

De plus la balise *<to uri=… />* a été paramétrée pour se connecter à l'API de Kafka et la clé *idAgence* a été paramétrée.

Enfin pour que cette route soit fonctionnelle, il est nécessaire d'intégrer la dépendance Camel-Kafka. JBang apporte l'avantage de simplifier l'importation de cette dépendance en exécutant la route comme suit :

camel run route.xml --deps camel.dependencies=org.apache.camel:camel-kafka,org

6.5 Mise en place d'un premier consommateur

L'implémentation d'un consommateur (consumer) a été réalisé par l'intermédiaire d'un nouveau fichier Camel de route xml. Ce dernier se connecte à Kafka sur le port 9092 et doit spécifier un nom de groupe de consommateurs.

L'exécution de cette route s'effectue via une commande identique à celle présenté dans le point précédent et met à jour un fichier JSON en sortie à chaque fois qu'un nouveau flux de données arrive dans le topic Kafka.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- camel-k: language=xml -->

<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://camel.apache.org/schema/spring"
        xsi:schemaLocation="
            http://camel.apache.org/schema/spring
            https://camel.apache.org/schema/spring/camel-spring.xsd">

    <!-- Write your routes here, for example: -->
    <route id="kafka-get-route">

        <!-- URL source Kafka et indication du groupid -->
        <from uri="kafka:temps-attente?brokers=localhost:9092&amp;groupId=temps"/>

        <!-- Déserialisation des données au format JSON -->
        <unmarshal>
            <json library="Jackson" prettyPrint="true" />
        </unmarshal>

        <!-- Affichage du JSON dans le terminal -->
        <log message="Received data: ${body}" loggingLevel="DEBUG"/>

        <!-- Conversion des données au format CSV -->
        <marshal>
            <json library="Jackson" prettyPrint="true" />
        </marshal>

        <!-- Route de destination du fichier avec ajout des données à la fin du csv-->
        <to uri="file:./data.csv?fileExist=Append&amp;fileName=temps-attente.json"/>

        <!-- Affichage du CSV dans le terminal -->
        <log message="Received data: ${body}" loggingLevel="DEBUG"/>

    </route>
</routes>
```

Figure 12 : consumer Camel-JBang

6.6 OpenSearch

L'implémentation du service OpenSearch a nécessité la mise en place d'un nouveau fichier YAML

```yaml
version: '3'

services:

  opensearch: # This is also the hostname of the container within the Docker network (i.e. https://opensearch/)
    image: opensearchproject/opensearch:latest # Specifying the latest available image - modify if you want a specific version
    container_name: opensearch
    environment:
      - cluster.name=opensearch-cluster # Name the cluster
      - node.name=opensearch # Name the node that will run in this container
      - discovery.seed_hosts=opensearch # Nodes to look for when discovering the cluster
      - cluster.initial_cluster_manager_nodes=opensearch # Nodes eligibile to serve as cluster manager
      - bootstrap.memory_lock=true # Disable JVM heap memory swapping
      - "OPENSEARCH_JAVA_OPTS=-Xms512m -Xmx512m" # Set min and max JVM heap sizes to at least 50% of system RAM
    ulimits:
      memlock:
        soft: -1 # Set memlock to unlimited (no soft or hard limit)
        hard: -1
      nofile:
        soft: 65536 # Maximum number of open files for the opensearch user - set to at least 65536
        hard: 65536
    user: "1000:1001"
    #volumes:
      #- ./volumes/opensearch_data:/usr/share/opensearch/data:rw # Creates volume called opensearch-data and mounts it to the container
    ports:
      - 9200:9200 # REST API
      - 9600:9600 # Performance Analyzer
    networks:
      - opensearch-net # All of the containers will join the same Docker bridge network
### Si l'erreur  "[1]: max virtual memory areas vm.max_map_count [65530] is too low, increase to at least [262144]" s'affiche, exécuter la commande ci-dessous
### sudo sysctl -w vm.max_map_count=262144

  opensearch-dashboards:
    image: opensearchproject/opensearch-dashboards:latest # Make sure the version of opensearch-dashboards matches the version of opensearch installed on other nodes
    container_name: opensearch-dashboards
    ports:
      - 5601:5601 # Map host port 5601 to container port 5601
    expose:
      - "5601" # Expose port 5601 for web access to OpenSearch Dashboards
    environment:
      OPENSEARCH_HOSTS: '["https://opensearch:9200"]' # Define the OpenSearch nodes that OpenSearch Dashboards will query
    networks:
      - opensearch-net

networks:
  opensearch-net:
    driver: bridge

volumes:
  esdata:
    driver: local
    driver_opts:
      o: uid=1000,gid=1000
```

Figure 13 : Docker-Compose OpenSearch

Ce fichier de configuration permet de lancer les conteneurs OpenSearch et OpenSearch Dashboard par l'intermédiaire de la commande :

docker compose -f <nomFichier>.yml up

7. Résultats attendus

Le résultat final correspond aux attentes initiales du projet. L'exécution de la route Camel-JBang au format XML permet d'acheminer efficacement et simplement des données d'une source vers la destination voulue que ce soit en mode Producer ou en mode Consumer.

De plus cette méthode de routage de données sera reprise et mise en pratique par le bureau GLIA sur d'autre projet à l'avenir, en s'appuyant sur le dépôt GitHub du projet (cf. chapitre 9) que nous avons régulièrement alimenter et qui intègre tous les scripts, modes opératoires produits (cf. chapitre 10) et problèmes rencontrés durant le projet.

8. Améliorations futures

Il est important de rester vigilant sur la suppression progressive de la brique Zookeeper par la communauté de Kafka. En tant que système externe, Zookeeper est une contrainte à l'évolutivité de Kafka.

Apache Camel est un outil très flexible et extensible, cependant, comme tout outil, il présente certaines limites. Bien qu'il soit conçu pour être rapide, il peut y avoir des baisses de performance lorsque le volume de données à traiter est trop important.

9. Conclusion

L'objectif principal du projet était de créer une route Camel permettant d'automatiser la récupération des données de l'API des temps d'attentes de l'OPT-NC et l'alimentation d'un topic Kafka.

Le travail produit dans le cadre de ce projet a permis d'étudier une nouvelle méthode de consommation des données afin qu'elle puisse être réinvestie de manière opérationnelle et à moindre effort par le bureau GLIA de Monsieur Adrien SALES sur d'autres thématiques.

De plus cela nous a permis une monté en compétences importante sur l'outil Apache Camel et Docker. Et en complément de commencer à appréhender les environnements Kafka et OpenSearch.

Nous regrettons de n'avoir pas eu le temps matériel de finaliser l'étude de l'outil OpenSearch et des fonctionnalités de géolocalisations, d'IA ou de ML qu'il propose.

10. Références

Dépôt du projet : https://github.com/opt-nc/opt-temps-attente-agences-camel

- Apache Camel Kamelets Examples
- Bringing live data to life with Digital Art & Unity
- Camel Jbang
- Docker Tutorial for Beginners
- Docker-Hub
- From waiting time metrics to Generative art
- How to Use the Apache httpd Docker Official Image
- Jbang
- Learn how to use Kafkacat – the most versatile Kafka CLI client
- Organizational Performance with Open Data, 5S & digital art
- Pipe xlsx files into/from Kafka... From cli with (k)cat
- Qu'est-ce qu'OpenSearch ?
- Show me your DockerHub public images in your terminal
- Waiting time in OPT-NC agencies... on Rapidapi.com
- We have just released our JBang! Catalog
- Welcome to Apache ZooKeeper
- We're on DockerHub

11. Annexes

11.1 Script de test

```
### Construction des URL de tests
$actuatorUri = "http://localhost:8080/actuator"
$healthUri = $actuatorUri +"/health"
$infoUri = $actuatorUri +"/info"

### HealtTestURI
Try {
   $healthTestUri =  Invoke-WebRequest -Uri $healthUri -UseBasicParsing
   $healthStatusCode = $healthTestUri.StatusCode

   If ($healthStatusCode -eq "200"){
   Write-Host "API HealthTest OK :" $healthStatusCode
   }Else {
      Write-Host "API HealthTest KO :" $healthStatusCode
   }

}Catch{
   $healthStatusCode = $_.Exception.Response.StatusCode.value__
}

### InfoTestURI
Try {
   $infoTestUri =  Invoke-WebRequest -Uri $infoUri -UseBasicParsing
   $infoStatusCode = $healthTestUri.StatusCode

   If ($infoStatusCode -eq "200"){
      Write-Host "API InfoTest OK :" $infoStatusCode
   }Else {
      Write-Host "API InfoTest KO :" $infoStatusCode
   }

}Catch{
   $infoStatusCode = $_.Exception.Response.StatusCode.value__
}

### Test de récupération du csv de toutes les agences
If (($healthTestUri.StatusCode -eq "200") -and  ($healthTestUri.StatusCode -eq "200")) {
   $webdata = Invoke-WebRequest -Uri http://localhost:8080/temps-attente/agences -
UseBasicParsing

   write-host "Récupération des données des agences en cours..."
   $releases = ConvertFrom-Json $webdata.Content
   write-host "Récupération des données des agences terminée."

   ### Export des données dans un fichier CSV
   if ($releases) {
      $releases | Export-Csv .\agences.csv -delimiter ';' -NoTypeInformation
       $file = Get-ChildItem agences.csv
       Write-Host "Le fichier" $file.Name "a été créé dans le répertoire courant."
```

```
    }

    Write-Host "L'API contient" $releases.Count "agences."

    ### Test pour une agence
    $4161 = Invoke-WebRequest -Uri http://localhost:8080/temps-attente/agence/4161 -
UseBasicParsing
    Write-host "Détail de l'agence 4161" -ForegroundColor Green
    $4161
}
```

11.2 Docker-Compose API/Kafka/Zookeeper/Kowl

```
version: "3"

services:
  optnc :
    hostname : optnc
    image : optnc/opt-temps-attente-agences-api
    container_name: optnc
    ports:
      - "8080:8080"

  zookeeper:
    hostname : zookeeper
    image: docker.io/bitnami/zookeeper:latest
    container_name: zookeeper
    ports:
      - "2181:2181"
    healthcheck:
      test: ["CMD", "echo", "ruok"]
      interval: 30s
      timeout: 5s
      retries: 3
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes

  kafka:
    hostname : kafka
    image: docker.io/bitnami/kafka:latest
    container_name: kafka
    ports:
      - "9092:9092"
      - "9093:9093"
    depends_on:
      - zookeeper
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes

  kowl:
    image: quay.io/cloudhut/kowl:v1.3.1
    container_name: kowl
    restart: always
```

```
ports:
  - "8888:8888"
depends_on:
  - kafka
environment:
  - KAFKA_BROKERS=kafka:9092
  - SERVER_LISTENPORT=8888
```

11.3 Route Camel-JBang XML – Producer Kafka

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- camel-k: language=xml -->

<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://camel.apache.org/schema/spring"
        xsi:schemaLocation="
            http://camel.apache.org/schema/spring
            https://camel.apache.org/schema/spring/camel-spring.xsd">

    <!-- Write your routes here, for example: -->
    <route id="http-get-route">

        <!-- Ordonnance de l'exécution de la route toutes les 5 minutes -->
        <from uri="timer:http-get-timer??period={{time:300000}}"/>

        <!-- URL source de l'API avec la méthode GET -->
        <to uri="http://localhost:8080/temps-attente/agences?httpMethod=GET"/>

        <!-- Déserialisation des données au format JSON -->
        <unmarshal>
            <json library="Jackson" prettyPrint="false" />
        </unmarshal>

        <!-- JSON -->
        <!-- Conversion des données au format JSON -->
        <marshal>
            <json library="Jackson" prettyPrint="false" />
        </marshal>

        <!-- URL de destination Kafka et paramétrage de la clé -->
        <to uri="kafka:temps-attente?brokers=localhost:9092&amp;key=idAgence"/>

    </route>
</routes>
```

11.4 Route Camel-JBang XML – Consumer Kafka

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!-- camel-k: language=xml -->

<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://camel.apache.org/schema/spring"
        xsi:schemaLocation="
            http://camel.apache.org/schema/spring
            https://camel.apache.org/schema/spring/camel-spring.xsd">

    <!-- Write your routes here, for example: -->
    <route id="kafka-get-route">

        <!-- URL source Kafka et indication du groupid -->
        <from uri="kafka:temps-attente?brokers=localhost:9092&amp;groupId=test"/>

        <!-- Déserialisation des données au format JSON -->
        <unmarshal>
            <json library="Jackson" prettyPrint="true" />
        </unmarshal>

        <!-- Affichage du JSON dans le terminal -->
        <log message="Received data: ${body}" loggingLevel="DEBUG"/>

        <!-- Conversion des données au format CSV -->
        <marshal>
            <json library="Jackson" prettyPrint="true" />
        </marshal>

        <!-- Route de destination du fichier avec ajout des données à la fin du csv-->
        <to uri="file:./data.csv?fileExist=Append&amp;fileName=temps-attente.json"/>

        <!-- Affichage du CSV dans le terminal -->
        <log message="Received data: ${body}" loggingLevel="DEBUG"/>

    </route>
</routes>
```

11.5 Docker-Compose OpenSearch/OpenSearhc-Dashboard

```yaml
version: '3'

services:

  opensearch: # This is also the hostname of the container within the Docker network (i.e. https://opensearch/)
    image: opensearchproject/opensearch:latest # Specifying the latest available image - modify if you want a specific version
    container_name: opensearch
    environment:
      - cluster.name=opensearch-cluster # Name the cluster
      - node.name=opensearch # Name the node that will run in this container
```

```
      - discovery.seed_hosts=opensearch # Nodes to look for when discovering the
cluster
      - cluster.initial_cluster_manager_nodes=opensearch # Nodes eligibile to serve as
cluster manager
      - bootstrap.memory_lock=true # Disable JVM heap memory swapping
      - "OPENSEARCH_JAVA_OPTS=-Xms512m -Xmx512m" # Set min and max JVM
heap sizes to at least 50% of system RAM
    ulimits:
      memlock:
        soft: -1 # Set memlock to unlimited (no soft or hard limit)
        hard: -1
      nofile:
        soft: 65536 # Maximum number of open files for the opensearch user - set to at
least 65536
        hard: 65536
    user: "1000:1001"
    #volumes:
      #- ./volumes/opensearch_data:/usr/share/opensearch/data:rw # Creates volume
called opensearch-data and mounts it to the container
    ports:
      - 9200:9200 # REST API
      - 9600:9600 # Performance Analyzer
    #command: bash -c "cp /usr/share/logstash/vendor/bundle/jruby/2.3.0/gems/logstash-
output-opensearch-1.1.0-java/lib/logstash/outputs/opensearch/templates/ecs-
disabled/1x.json /usr/share/logstash/vendor/bundle/jruby/2.3.0/gems/logstash-output-
opensearch-1.1.0-java/lib/logstash/outputs/opensearch/templates/ecs-disabled/2x.json"
    networks:
      - opensearch-net # All of the containers will join the same Docker bridge network
 ### Si l'erreur  "[1]: max virtual memory areas vm.max_map_count [65530] is too low,
increase to at least [262144]" s'affiche, exécuter la commande ci-dessous
  ### sudo sysctl -w vm.max_map_count=262144

  opensearch-dashboards:
    image: opensearchproject/opensearch-dashboards:latest # Make sure the version of
opensearch-dashboards matches the version of opensearch installed on other nodes
    container_name: opensearch-dashboards
    ports:
      - 5601:5601 # Map host port 5601 to container port 5601
    expose:
      - "5601" # Expose port 5601 for web access to OpenSearch Dashboards
    environment:
      OPENSEARCH_HOSTS: '["https://opensearch:9200"]' # Define the OpenSearch
nodes that OpenSearch Dashboards will query
    networks:
      - opensearch-net

networks:
  opensearch-net:
    driver: bridge

volumes:
  esdata:
    driver: local
    driver_opts:
```

```
o: uid=1000,gid=1000
```

11.6 ReadMe.md GitHub

```readme
# Prérequis

* docker
* camel-jbang
* kafka (via docker-compose)
* kafkacat

## Installer camel-jbang

Note: Java n'a pas besoin d'être déjà présent car jbang va le télécharger automatiquement si java 8 ou un JDK plus récent n'est pas détecté.

### 1. Installer jbang

### <i>Linux</i> :
Ouvrir un terminal puis exécuter une des deux commandes suivantes :

* Avec SDKMan
``` bash
sdk install jbang
```
* Avec jbang lui même :
``` bash
curl -Ls https://sh.jbang.dev | bash -s - app setup
```
Une fois installer, il est recommandé d'exécuter la commande suivante afin de modifier les variables d'environnement pour inclure les scripts jbang :
```bash
jbang app setup
```

<i>Windows</i> :

* Avec jbang :
    * ouvrir powershell et exécuter la commande suivante :
    ```powershell
    iex "& { $(iwr -useb https://ps.jbang.dev) } app setup"
    ```
* Avec chocolatey
    * A partir d'une invite commande installer chocolatey :
    ```bash
    choco install jdk11
    ```
    * Une fois java installer :
    ```bash
    choco install jbang
```
```

```

### 2. Installer camel
* Ouvrir une invite de commande ou un terminal et exécuter la commande suivante :
```bash
jbang app install camel@apache/camel
```

3. Installer Kafkacat
* Ouvrir une invite de commande ou un terminal et exécuter la commande suivante :
```bash
apt install -y kafkacat
```

Utilisation de la route camel :
Ouvrir un terminal ou une invite de commande puis exécuter les commandes suivantes dans le dossier où le script docker-compose se trouve :

* Lancement des différents conteneur docker dont l'API des temps d'attentes en agence avec le fichier docker-compose a récupérer sur le dépôt Github
```bash
docker-compose up -d
```

* Pour afficher le status des conteneurs, exécuter la commande ci-dessous
```bash
docker ps
```

L'affichage devrait correspondre à celui ci-dessous :
```bash
CONTAINER
ID   IMAGE                                  COMMAND                  CREATED       STATUS
                     PORTS                                                  NAMES
bcd9edd7de54   quay.io/cloudhut/kowl:v1.3.1          "./kowl"                 4 days
ago   Up 6 seconds                       0.0.0.0:8888-
>8888/tcp                                  kowl
52bea7369ab6   bitnami/kafka:latest                  "/opt/bitnami/script…"   4 days
ago   Up 9 seconds                       0.0.0.0:9092-9093->9092-
9093/tcp                       kafka
84c0d56969ec   optnc/opt-temps-attente-agences-api   "/cnb/process/web"       4 days
ago   Up 11 seconds                      0.0.0.0:8080-
>8080/tcp                                  optnc
551fff613ee6   bitnami/zookeeper:latest              "/opt/bitnami/script…"   4 days
ago   Up 11 seconds (health: starting)   2888/tcp, 3888/tcp, 0.0.0.0:2181->2181/tcp,
8080/tcp   zookeeper
```
A noter que les conteneurs ont été nommés via le fichier yaml, cf. colonne "NAMES".

* Pour arrêter les conteneurs
```bash

```
    docker-compose stop
    ```

Kafka
* Création d'un Producer kafka :

Linux :
* Depuis un terminal
    ```bash
    kafkacat -P -b kafka:9092 -t temps-attente &
    ```
* Pour afficher les topics
    ```bash
    kafkacat -b localhost:9092 -L
    ```

Windows :
* Ouvrir un terminal et exécuter les commandes ci-dessous

    ### Depuis Docker
    ```bash
    docker exec -it kafka kafka-console-producer.sh --broker-list localhost:9092 --topic
temps-attente -P
    ```
    * Pour afficher les topics
    ```bash
    docker exec -it kafka  kafka-topics.sh --list --bootstrap-server localhost:9092
    ```
    * Pour supprimer un topic
    ```bash
    docker exec -it kafka kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic
"temps-attente"
    ```
Depuis le conteneur Kafka
* Exécuter les commandes ci-dessous pour accèder au terminal du docker Kafka
    ```bash
    docker exec -it kafka sh
    ```
    Ou
    ```bash
    docker exec -ti kafka bash
    ```
    Se positionner dans le répertoire bin de Kafka
    ```bash
    cd opt/bitnami/kafka/bin/
    ```
    Et exécuter la commande suivante
    ```bash
    kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --
partitions 1 --topic temps-attente
    ```
    * Pour afficher les topics

    ```bash
    kafka-topics.sh --list --bootstrap-server localhost:9092
    ```
    * Pour supprimer un topic
    ```bash
    kafka-topics.sh --bootstrap-server localhost:9092 --delete --topic temps-attente
    ```

Lancement de la route Camel :

IMPORTANT : la route camel contient une dépendance Kafka, il est nécessaire de l'exécuter comme suit :
```bash
camel run route-kafka.xml --dep camel.dependencies=org.apache.camel:camel-kafka
```

Si vous souhaitez que la sortie se fasse dans un fichier log, ajoutez l'option ci-dessous à la fin de la précédente commande :

```bash
> output.log
```

Déroulement et résultat attendu de la route :
Une fois lancé, la route Camel envoie une requête GET-HTTP vers l'API des temps d'attente de l'OPT.

Cette dernière lui répond et la route reçoit le message de retour au format JSON.
Les données sont ensuite poussées dans le topic Kafka 'temps-attente', créé précédemment.

Cette procédure est répétée toute les 5 minutes tant que la route n'est pas manuellement arrêtée.

A noter que la route Camel formatte les données de façons à ce qu’elles soient sur une seule ligne afin que Kafka puisse l'intégrer.
Par ailleurs le champ idAgence est utilisé comme clé au niveau du topic Kafka.

Les messages reçus sont visibles via le conteneur Kowl en utilisant l'URL suivante https://localhost:8888/topics/temps-attente