



Université Mohammed V de Rabat
École Mohammadia d'Ingénieurs

Module : Traitement du Streaming

SYSTÈME DE SUPERVISION DE TRANSPORT SCOLAIRE DISTRIBUÉ AVEC KAFKA, SPRING BOOT ET MICROSERVICES

Réalisé par :
SMAIKI ANAS

Sous la direction de :
**Mr. KABBAJ MOHAMED
ISSAM**

Année Universitaire 2025-2026

Table des matières

1	Introduction	2
1.1	Contexte et Problématique	2
1.2	Objectifs du Projet	2
2	Infrastructure Kafka et Déploiement	3
2.1	Le Rôle de Zookeeper et du Broker	3
2.2	Configuration Docker Compose (YAML)	3
2.3	Commandes de Lancement	4
3	Architecture Décentralisée et Separation of Concerns	5
3.1	Le Concept du "Thin Client" (BusProducer)	5
3.2	Le Cerveau Centralisé (API Spring Boot)	5
3.3	Gestion des Topics Kafka	5
3.3.1	1. Topic <code>positions-bus</code>	5
3.3.2	2. Topic <code>penalites-bus</code>	6
4	Implémentation et Intégration Frontend-Backend	7
4.1	Liaison Frontend-Backend et Ajout Temps Réel	7
4.2	Calcul de Distance et Routing (Nearest Neighbor)	8
4.3	Gestion Automatisée des Pénalités	8
4.4	Succès du Ramassage	9
5	Conclusion	10

1. Introduction

1.1 Contexte et Problématique

La supervision des flottes de transport scolaire nécessite une réactivité en temps réel et une tolérance aux pannes élevée. Les architectures monolithiques classiques montrent leurs limites face à la gestion de flux de données continus (positions GPS) et d'événements critiques (retards, pénalités).

1.2 Objectifs du Projet

Ce projet vise à concevoir une architecture **décentralisée** et **événementielle (Event-Driven)** capable de :

- Ingérer des flux de données GPS en temps réel via **Apache Kafka**.
- Déléguer l'intelligence de calcul (routing) à des microservices stateless.
- Assurer l'intégrité des données d'infraction (pénalités) même en cas de panne partielle.
- Fournir une interface de visualisation réactive intégrant des ajouts de données dynamiques.

2. Infrastructure Kafka et Déploiement

Le cœur de notre système de messagerie repose sur Apache Kafka. Pour garantir la portabilité et la facilité de déploiement, nous avons conteneurisé l'infrastructure avec Docker.

2.1 Le Rôle de Zookeeper et du Broker

L'architecture Kafka déployée comprend deux composants essentiels :

1. **Zookeeper** : Il agit comme le coordinateur du cluster. Il gère l'élection du contrôleur, la configuration des topics et le suivi de l'état de santé du Broker. Sans lui, le cluster Kafka ne peut fonctionner.
2. **Kafka Broker** : C'est le serveur de stockage qui reçoit, stocke et distribue les messages. Dans notre configuration mono-nœud (pour le développement), il gère seul les partitions.

2.2 Configuration Docker Compose (YAML)

Voici l'extrait du fichier `docker-compose.yml` utilisé pour orchestrer le lancement des services. Nous exposons le Broker sur le port 9092 pour permettre la connexion depuis nos applications Java (Spring Boot et Producers).

```
1 version: '3'
2 services:
3   zookeeper:
4     image: confluentinc/cp-zookeeper:7.4.0
5     container_name: zookeeper
6     environment:
7       ZOOKEEPER_CLIENT_PORT: 2181
8       ZOOKEEPER_TICK_TIME: 2000
9
10  broker:
11    image: confluentinc/cp-kafka:7.4.0
12    container_name: broker-1
13    ports:
14      - "9092:9092"
15    depends_on:
16      - zookeeper
17    environment:
18      KAFKA_BROKER_ID: 1
19      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
20      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT
21      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
```

```
22 KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
```

Listing 2.1 – Configuration Infrastructure Kafka

2.3 Commandes de Lancement

Le cycle de vie de l'infrastructure est géré via les commandes suivantes :

- Démarrage en arrière-plan : `docker-compose up -d`
- Vérification des logs : `docker logs -f broker-1`
- Création manuelle de topic (si nécessaire) :

```
1 docker exec -it broker-1 kafka-topics --create \  
2   --bootstrap-server localhost:9092 \  
3   --replication-factor 1 --partitions 1 --topic positions-bus  
4
```

3. Architecture Décentralisée et Separation of Concerns

L'un des points forts de ce projet est la séparation stricte des responsabilités (*Separation of Concerns*). Nous avons évité le piège du "Bus Intelligent" pour privilégier une architecture Client-Serveur robuste.

3.1 Le Concept du "Thin Client" (BusProducer)

Le programme simulant le bus (**BusProducer**) a été conçu comme un client léger ("Thin Client").

- **Ce qu'il ne fait pas** : Il ne connaît pas la géométrie de la ville, ne calcule pas de distances et ne décide pas de l'ordre de passage.
- **Ce qu'il fait** : Il s'occupe uniquement de la télémétrie (envoi GPS) et de l'exécution des ordres reçus.

3.2 Le Cerveau Centralisé (API Spring Boot)

Toute la logique mathématique et algorithmique est centralisée dans l'API Spring Boot.

- **DistanceController** : Expose des endpoints REST pour calculer la distance euclidienne ou Haversine entre deux points.
- **Optimisation** : C'est le serveur qui exécute l'algorithme du "Plus Proche Voisin" (*Nearest Neighbor*) sur demande du bus.

FIGURE 3.1 – Diagramme de séquence : Le Bus demande sa route au Serveur, puis exécute le trajet.

3.3 Gestion des Topics Kafka

Nous avons structuré les échanges de données autour de deux topics distincts pour séparer les flux de haute fréquence des événements métiers critiques.

3.3.1 1. Topic positions-bus

- **Type** : Flux continu (Streaming).
- **Producer** : **BusProducer**.

- **Consumer** : ParentApp (Frontend) et EcoleMonitoring.
- **Rôle** : Mise à jour de la carte en temps réel (Lat/Lon, Status).

3.3.2 2. Topic penalites-bus

- **Type** : Événementiel (Event-Driven).
- **Producer** : EcoleMonitoring.
- **Consumer** : PenaliteService.
- **Rôle** : Assure la persistance des amendes. Si le service de base de données est éteint, Kafka retient ces messages pour les traiter plus tard (Résilience).

4. Implémentation et Intégration Frontend-Backend

Ce chapitre détaille comment les différents composants interagissent pour former un système cohérent.

4.1 Liaison Frontend-Backend et Ajout Temps Réel

L'interface utilisateur (Leaflet + JS) n'est pas statique. Elle interagit dynamiquement avec le backend.

Le Scénario d'Ajout Dynamique :

1. L'administrateur clique sur la carte et saisit un nom ("Famille X").
2. Une requête POST `/api/parents` est envoyée au Backend.
3. La base de données MySQL est mise à jour instantanément.
4. **Impact sur le Bus :** Lors de son prochain arrêt, le Bus interroge l'API. L'algorithme de routing inclut immédiatement cette nouvelle famille dans le calcul du plus proche voisin, sans redémarrer le système.

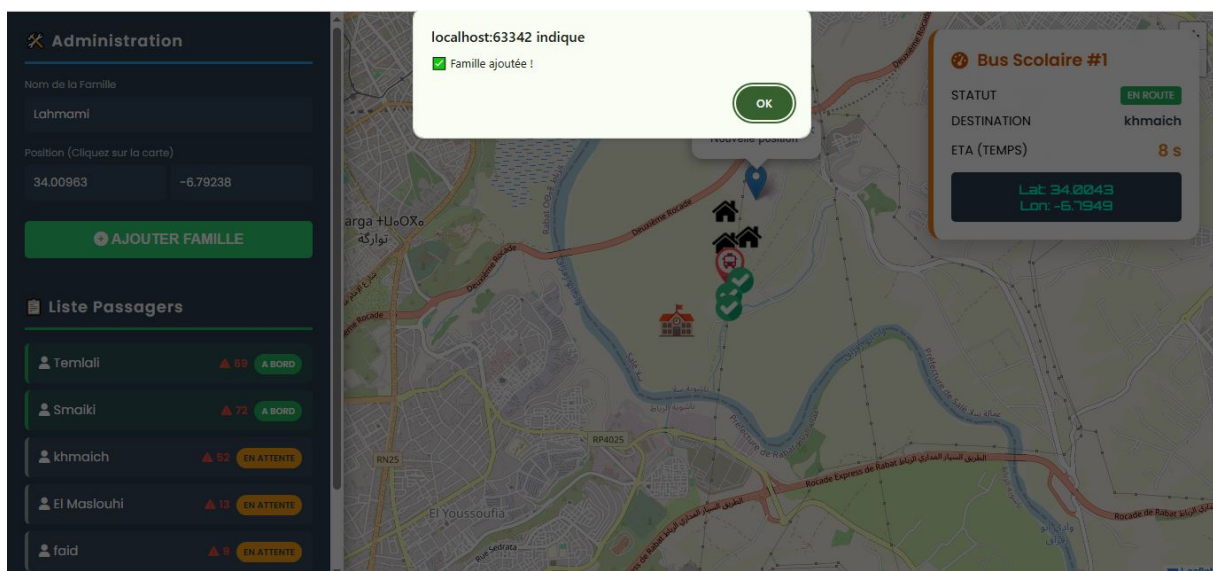


FIGURE 4.1 – Interface d'administration permettant l'injection de nouvelles données en cours de simulation.

4.2 Calcul de Distance et Routing (Nearest Neighbor)

Le choix de la prochaine destination est le fruit d'une collaboration API :

```

1 // 1. Le Bus envoie sa position actuelle et la liste des lves
  restants
2 ParentLocation nextStop = HttpUtils.demanderProchaineDestination(
  currentLat, currentLon, toVisit);
3
4 // 2. Le Bus reçoit l'objet JSON optimal et affiche la notification
5 System.out.println("CAP SUR : " + nextStop.nom);

```

Listing 4.1 – Logique de Routing côté BusProducer

Sur le Frontend, cette décision est captée pour afficher la **notification bleue** prédictive.

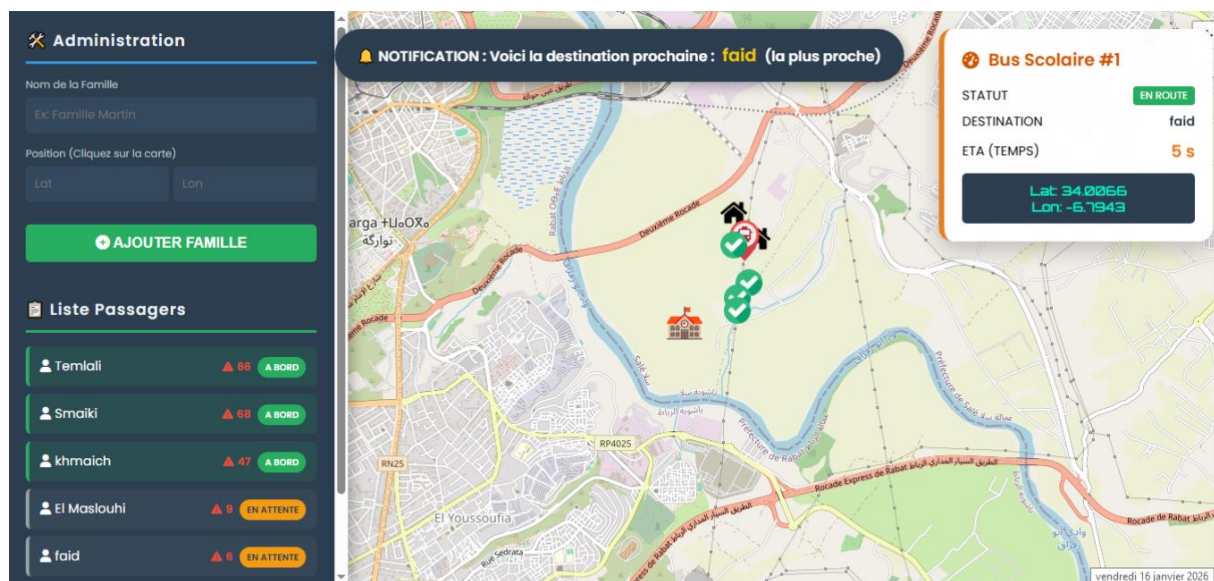


FIGURE 4.2 – Notification prédictive affichant la décision prise par l'algorithme serveur.

4.3 Gestion Automatisée des Pénalités

La chaîne de traitement des pénalités illustre parfaitement l'architecture distribuée :

1. **Constat** : Le Bus s'arrête. Il envoie `status: STOPPED` et `time: 8s` sur `positions-bus`.
2. **Détection** : Le consommateur `EcoleMonitoring` lit le message. Il applique la règle métier (Si temps > 5s).
3. **Alerte** : Si infraction, il produit un message JSON dans le topic `penalites-bus`.
4. **Sanction** : Le consommateur `PenaliteService` lit ce topic spécifique et déclenche l'écriture en BDD.
5. **Visualisation** : Le Frontend détecte le flag de pénalité et affiche l'alerte rouge.

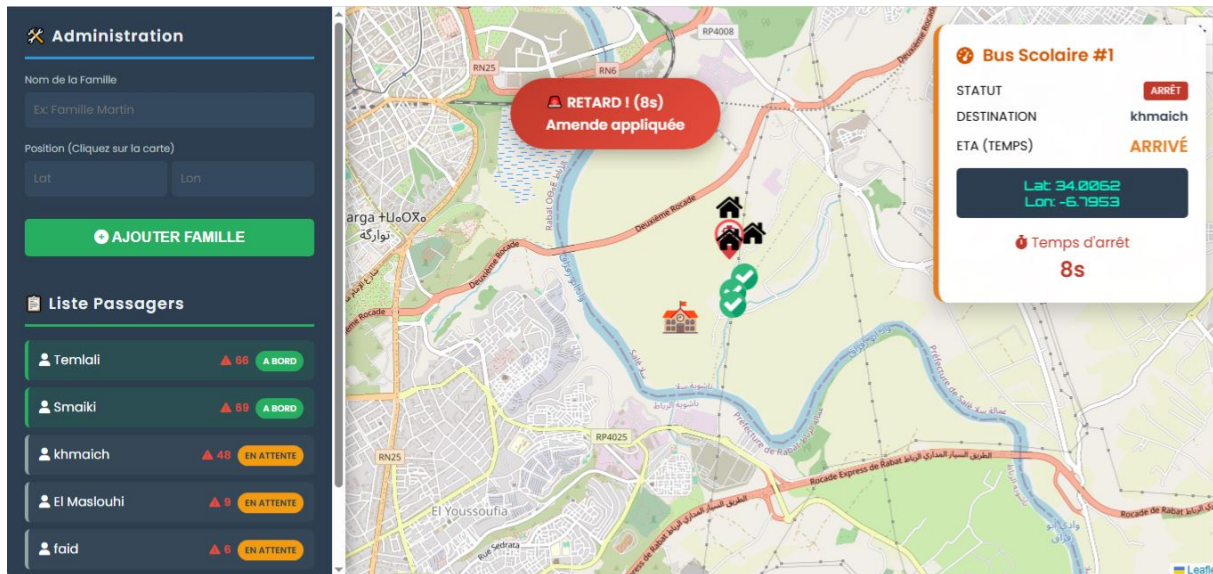


FIGURE 4.3 – Visualisation d'une infraction traitée par la chaîne Kafka complète.

4.4 Succès du Ramassage

Dans le cas nominal, le processus est fluide et l'interface confirme le bon déroulement.

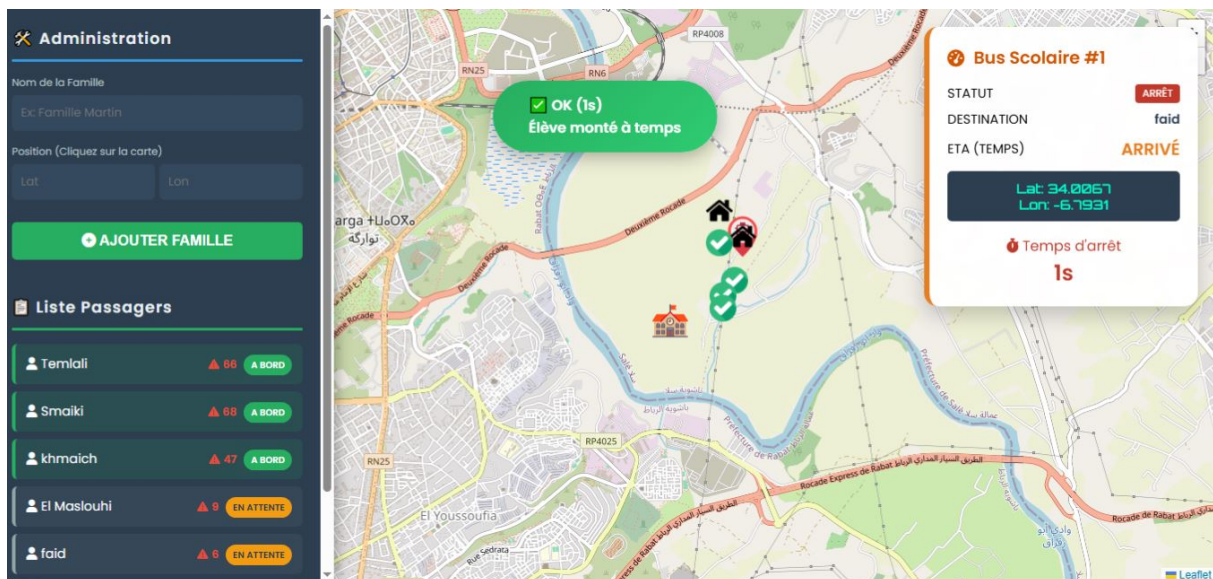


FIGURE 4.4 – Validation visuelle d'un arrêt conforme aux règles.

5. Conclusion

Ce projet a permis de démontrer la viabilité d’une architecture distribuée pour la gestion de flottes en temps réel. En remplaçant une logique monolithique par une orchestration de microservices autour d’**Apache Kafka**, nous avons obtenu un système :

- **Robuste** : La séparation des topics (Positions vs Pénalités) garantit que la saturation de la télémétrie n’impacte pas la gestion des incidents.
- **Évolutif** : L’ajout de nouvelles règles métier (ex : optimisation de carburant) se ferait uniquement côté serveur API, sans modifier le code embarqué des bus.
- **Interactif** : L’intégration bidirectionnelle entre le Frontend Leaflet et le Backend permet une supervision proactive (notifications prédictives).

Les compétences acquises couvrent l’ensemble du cycle DevOps : de la configuration infrastructurelle (**Docker/Zookeeper**) au développement applicatif (**Spring Boot/Java**) et à l’intégration Front-End.