

Architecture Logicielle: Évolution

Equipe B: Guillaume Piccina, William d'Andrea, Nicolas Fernandez, Yann Brault

Sujet V6 : Pay as you pollute

Les services en détails:

- **Car tracker :**
 - Consomme dans le bus un topic de tracking
 - Transmettre à grande vitesse les infos de tracking vers la base de données (heure de pointes, bouchons)
 - Option les infos de zones en faisant une requête à Zones Pollution et les met en cache (Cache NestJS)
- **Tracking shutdown :**
 - Consomme dans le bus un topic de shutdown
 - Récupère en base de données *Tracking Infos* les data du user
 - Déclenche la facturation pour un user en envoyant les data
 - Flush la base de données *Tracking Infos* de ces data
- **Billing :**
 - Reçoit l'événement les infos de tracking depuis *Tracking Shutdown*
 - Calcule la facture finale du trajet
 - Stock la facture en dans la base de données *Bills*
 - Peut recevoir des requêtes de clients pour accéder à la base de données *Bills*
- **anti fraud service:**
 - Anciennement position checker
 - Consomme dans un topic de redémarrage
 - Communique avec le service de caméras de la ville
 - Contact Billing en cas de problème

Nouveaux besoins:

- Retour et aide à l'utilisateur en temps réel (le guider vers un parcours en zone moins chère)
- Détection des coupures réseaux avec détection de fraudes (par une carte des zones blanches connues) donc recalcule des montants après coup
- Pousser la résilience et le maintien des infos de passages
- Scénario de résilience maximale : pouvoir exploiter les données de passages / positionnement a posteriori

Service pour répondre aux nouveaux besoins (en discussion)

- **User configurator :**
 - Branché sur un topic kafka
 - Pull les infos de zones de pollutions : coordonnées des zones + prix associés aux zones
 - Calcule des zones géographiques "de transition" autour des frontières des zones de pollution
 - Doit faire correspondre une position gps avec des zones
 - Retourne au service, qui communique avec les users, une politique de tracking (fréquent si proche d'une frontière de zones)
- **Route advisor**
 - Doit accepter un trajet et retourner les zones composants ce trajet et donner un coût provisoire
 - Reçoit des requêtes périodique de l'utilisateur pour connaître la position et recalcul l'itinéraire
 - Renvoie à l'utilisateur, le trajet, la position, le prix estimé de la course et les zones
 - Remontée d'informations se faisant à travers client communication service donc via socket
 - Génère les trajets via le composant open route service
- **Camera Checker:**
 - Mock de vérification par vidéo surveillance
- **Tracking analytics:**
 - Composant en charge de récupérer toutes les données de tracking
 - Met les données dans une base du même nom dans le but de pouvoir les exploiter plus tard
 - Nouvelle DB nécessaire car celle de tracking flush les data quand un utilisateur finit sa course

- **User Position proxy**
 - Joue le rôle d'intermédiaire entre notre architecture backend et notre vrai client
 - Il reçoit les positions des utilisateurs en temps réel et émet ces positions dans le bus de position
 - Proxy load balancé, aide pour le faire :
<https://socket.io/docs/v4/using-multiple-nodes/>
- **ClientCommunicationService**
 - Il ouvre un websocket avec le client afin d'envoyer en temps réel des informations de changement d'itinéraire ou de changement de fréquence d'envoi de position et de prix
 - Utilise la plaque d'immatriculation, ou peut être un hash de différentes infos utilisateurs comme identifiants pour garder la communication en vie
 - Utilise un système d'accusé de réception de la part du client pour les informations comme par exemple un trajet pour ne perdre de l'info
 - Système de cache qui flush quand confirmation de réception de l'utilisateur pour la persistance des données
- **User Client**
 - Application mobile du client (comportant le prix dépensé en temps réel, l'itinéraire, l'affichage de ses factures)
 - Application développée sous framework front-end (angular ou react)
 - Communique via REST pour l'envoi de position toutes les N secondes avec le Proxy
 - Communique avec un second service afin de demander les infos de prix et le prix d'un itinéraire ou suggestion d'itinéraire
 - Reçoit des informations du service de communication client, via un web socket (changement d'itinéraire, changement de fréquence d'envoi de position)
- **OpenRouteService (externe):**
 - Sert pour les itinéraires
 - en pratique container avec une image open route pour éviter de se faire ban par l'api officielle lors des test de charges

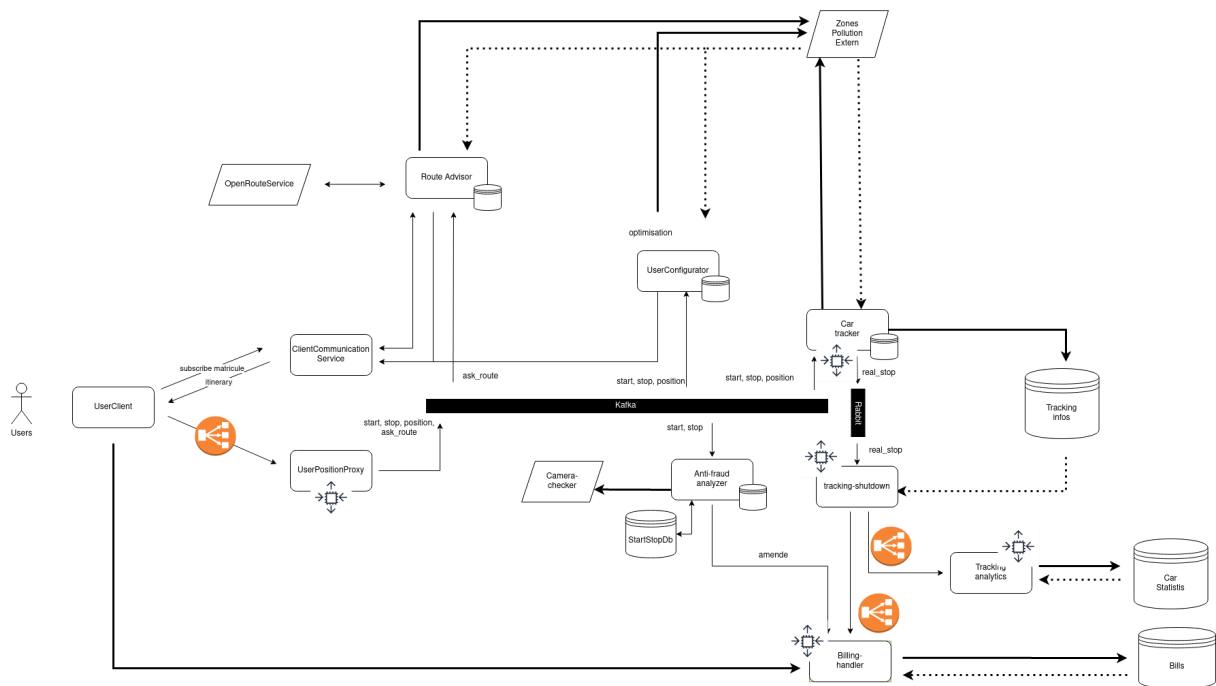
Scénario du flot de données V2

- L'utilisateur, quand il arrive dans sa voiture, appuie sur le bouton "se connecter" via UserClient, ce qui ouvre un socket avec ClientCommunicationService.
- ClientCommunicationService enregistre ce lien
- L'utilisateur entre sa destination dans son application
- Une requête est envoyée à ClientCommunicationService qui est redirigée à RouteAdvisor afin de générer un nouvel itinéraire
- RouteAdvisor envoie l'itinéraire (avec le montant potentiel à payer) à ClientCommunicationService qui envoie, via le socket, cet itinéraire au client
- Le client démarre son trajet, chaque N secondes, une requête (avec la position, timestamp, plaque d'immatriculation) est envoyée à UserPositionProxy qui convertit cette information en message et la transfère dans le bus.
- CarTracker reçoit le message, il le stocke en DB
- PositionChecker lit 1 message sur 10 dans le bus, regarde que la voiture se situe bien à l'endroit défini, si ce n'est pas le cas, génération d'une amende à travers le micro-service billing
- UserConfigurator lit chaque message, regarde la distance entre la voiture et les frontières de zones, renvoie une information au Proxy (qu'il envoie au client) sur la fréquence d'envoi de position
- L'utilisateur envoie périodiquement des requêtes à route advisor pour savoir s'il suit bien son trajet
- Quand un utilisateur a fini son trajet, il émet un "car-shutdown" qui va permettre au micro-service CarShutdown de mettre fin à la course, ce qui va entraîner une sauvegarde du trajet (pour avoir des statistiques) grâce à Tracking Analytics et une génération de la facture via Billing

Choix de techno :

- Bus : RabbitMQ, Kafka
- BDs : Mongo
- Archi micro service : NestJS
- Cache: Redis, NestJS

Schéma de l'architecture :



Auto-évaluation :

Cette seconde phase a été un peu plus compliquée que la première. Sur le plan technique les nouveaux besoins demandés ont été assez compliqués, ce qui a eu pour conséquence de nous pousser à bien réfléchir avant de faire des choix et de prendre des décisions, nous pensons notamment au changement de la technologie gérant le flux événementiel du projet ou encore comment gérer la remontée d'informations vers un utilisateur en particulier. En termes d'organisation, quelques divergences sont apparues lors de nos débats sur les idées mais des accords communs à tout le groupe étaient trouvés à chaque décision. Le groupe a conservé une bonne cohésion et une bonne synergie jusqu'à la fin. Toutes les tâches ont été réparties de manière à peu près équivalente, sauf la gestion du monitoring et tout ce qui touche à la charge du système (test et scale), cette partie a été entièrement faite par William comme lors de la première partie du projet.

- William 100 pts
- Guillaume 100 pts
- Nicolas 100 pts
- Yann 100 pts

