Architecture Definition Document v2

Diffusion : OpenClassrooms – Mentor : Cesare de Padua - Evaluateur :

Version	***Date***	***Auteur***	***Motif***
1	22/08/2025	Christophe Pierrès	Création du document
2			

Contenu ***Page***

1. Architecture Definition Document - Your Car Your Way..... 5

1.1. Objectifs du projet 5
1.1.1. Objectifs business 5
1.2. Principes d'architecture 5
1.2.1. Modularité et Microservices 5
1.2.2. Scalabilité et Performance 5
1.2.3. Sécurité et Conformité 6
1.2.4. Interopérabilité et Intégration 6
1.2.5. Résilience et Haute Disponibilité 6
1.2.6. Maintenabilité et Évolutivité 6
1.3. Architecture existante 6
1.4. Architectures 7
1.4.1. Vue Métier 7
1.4.1.1. Acteurs du système 7
1.4.1.2. Cas d'utilisation – Vue générale 8
1.4.1.3. Vue détaillée des interactions 9
1.4.1.4. Fonctionnalités détaillées par domaine 9
Gestion Utilisateur 9
Use case : Processus d'inscription utilisateur 10
Gestion Véhicules 11
Use case : Recherche de véhicules 11
Réservation 12
Use case : Réservation et paiement 12
Support Client 13
Use case : Système de support client 13
Diagramme d'activité : support client 14
Diagramme de séquence : Session de tchat complète (avec visio) 15
Intégrations avec les SI des agences via les API 16
1.4.2. Vue de données 16
1.4.2.1. Modèle de domaine - Microservices 16
Auth Service - Modèle de données 17
User Service - Modèle de donnée 18
Vehicle Service - Modèle de données avec Service ACRISS Externe 19
Reservation Service - Modèle de données 20
Support Service - Modèle de données (POC Chat) 21
1.4.2.2. Schémas de base de données par microservice 22
Diagramme d'entités-relations : vue générale des databases pour chaque microservice 23
1.4.3. Vue Technologique 25

1.4.3.1. Architecture Microservices (vue « mono-région ») 25
1.4.3.2. Diagramme de déploiement multi-régional (simplifié)..... 26
Vue détaillée du diagramme de déploiement complet (vue éclatée en 3 parties !) 27
1.4.3.3. Vision des protocoles d'échange 30
1.4.3.4. Stack technologique..... 31
Frontend 31
Backend..... 31
Base de données..... 31
1.4.3.5. Infrastructure 32
1.4.3.6. DevOps 32
1.4.4. Intégrations externes..... 32
1.4.4.1. Service ACRISS Externe 32
1.4.4.2. Stripe Payment..... 32
1.4.4.3. Services de communication 33
1.5. Justification de l'approche architecturale..... 33
1.5.1. Réponses aux objectifs 33
1.5.2. Respect des principes d'architecture 33
1.5.3. Suivi des normes et bonnes pratiques reconnues 34
1.5.4. Plus d'informations sur l'architecture et les composants 35
1.5.4.1. Amazon Route 53 39
1.5.4.2. Amazon CloudFront CDN 39
1.5.4.3. Équilibreurs de charge régionaux (HAProxy et NGINX)..... 39
1.5.4.4. Intérêts de Spring Cloud Gateway dans l'architecture..... 40
1.5.4.5. Kubernetes..... 41
1.5.4.6. Rôle de kafka 43
1.5.4.7. Importance de Kafka Mirror Maker 43
Événements critiques : 43
Exemples :..... 43
Sélection des événements critiques :..... 45
Processus de réplication : 45
Exemple concret de synchronisation d'un événement critique : 45
Avantages de cette approche :..... 46
1.5.4.8. Redis et gestion ACRISS..... 47
Gestion ACRISS..... 48
Intégration de Redis et de la gestion ACRISS dans Notre architecture48
Exemple de Scénario d'Intégration 49
Conclusion..... 50
1.6. Architectures de transition 50
1.6.1. Phase 1: Core Services (Mois 1-3) 50
1.6.2. Phase 2: Booking Engine (Mois 4-6) 50
1.6.3. Phase 3: Support & Notifications (Mois 7-8) 51
1.6.4. Phase 4: Multi-Regions (Mois 9-12)..... 51

1.6.5. Phase 5: Migration Legacy (Mois 12+) 51

1. Architecture Definition Document - Your Car Your Way

1.1. Objectifs du projet

1.1.1. Objectifs business

Your Car Your Way vise à moderniser et unifier son système d'information pour :

1. **Centraliser les opérations** : Remplacer les applications distinctes par pays par une plateforme unique multi-régionale
2. **Harmoniser l'expérience utilisateur/clients** : Offrir une interface cohérente tout en respectant les spécificités locales
3. **Faciliter la gestion** : Simplifier les réservations pour les clients et optimiser les opérations pour les agences
4. **Assurer la conformité** : Respecter les réglementations locales (RGPD, CCPA, post-Brexit)
5. **Permettre la croissance** : Préparer l'infrastructure pour une expansion géographique future
6. **Intégrer les systèmes existants** : Fournir des API pour les applications d'agence

1.2. Principes d'architecture

Les principes suivants guident la conception de l'architecture :

1.2.1. Modularité et Microservices

- Architecture en microservices pour faciliter la maintenance et l'évolution
- Séparation claire des responsabilités par domaine métier (Domain Driven Design)
- Database-per-service pour assurer l'indépendance des services

1.2.2. Scalabilité et Performance

- Capacité à supporter une charge croissante d'utilisateurs internationaux (+ de 10 000 simultanés)
- Cache distribué (Redis) pour optimiser les performances
- Déploiement multi-régional pour minimiser la latence
- Kafka pour event driven design

1.2.3. Sécurité et Conformité

- Protection des données personnelles selon les réglementations locales
- Chiffrement end-to-end des transactions sensibles
- Audit et traçabilité de toutes les opérations critiques (immutabilité des logs signés avec kafka, event sourcing)

1.2.4. Interopérabilité et Intégration

- API RESTful pour l'intégration avec les systèmes d'agence
- Standards ouverts (ACRISS) pour la classification des véhicules
- Intégration avec des services tiers de qualité (Stripe, services officiels)

1.2.5. Résilience et Haute Disponibilité

- Tolérance aux pannes avec circuit breakers et fallback (avec Spring Circuit breaker)
- Haute disponibilité (99.9%) avec réplication multi-région
- Monitoring et alerting proactif

1.2.6. Maintenabilité et Évolutivité

- Qualité de code contrôlée par la mise en place de Quality gates Sonar, documenté et testé (couverture > 80%)
- Déploiement continu avec CI/CD
- Architecture évolutive permettant l'ajout de nouvelles fonctionnalités

1.3. Architecture existante

Le projet ne s'appuie pas sur un produit existant qui serait repris puis mis à jour.

Il s'agit d'un remplacement complet des applications disparates actuellement utilisées dans chaque pays pour la gestion des clients.

Cependant, le système doit intégrer avec :

- **Applications d'agence existantes** : Via API RESTful pour maintenir les opérations terrain
- **Systèmes de paiement locaux** : Migration vers Stripe pour uniformiser
- **Données historiques** : Migration des réservations et profils clients existants

1.4. Architectures

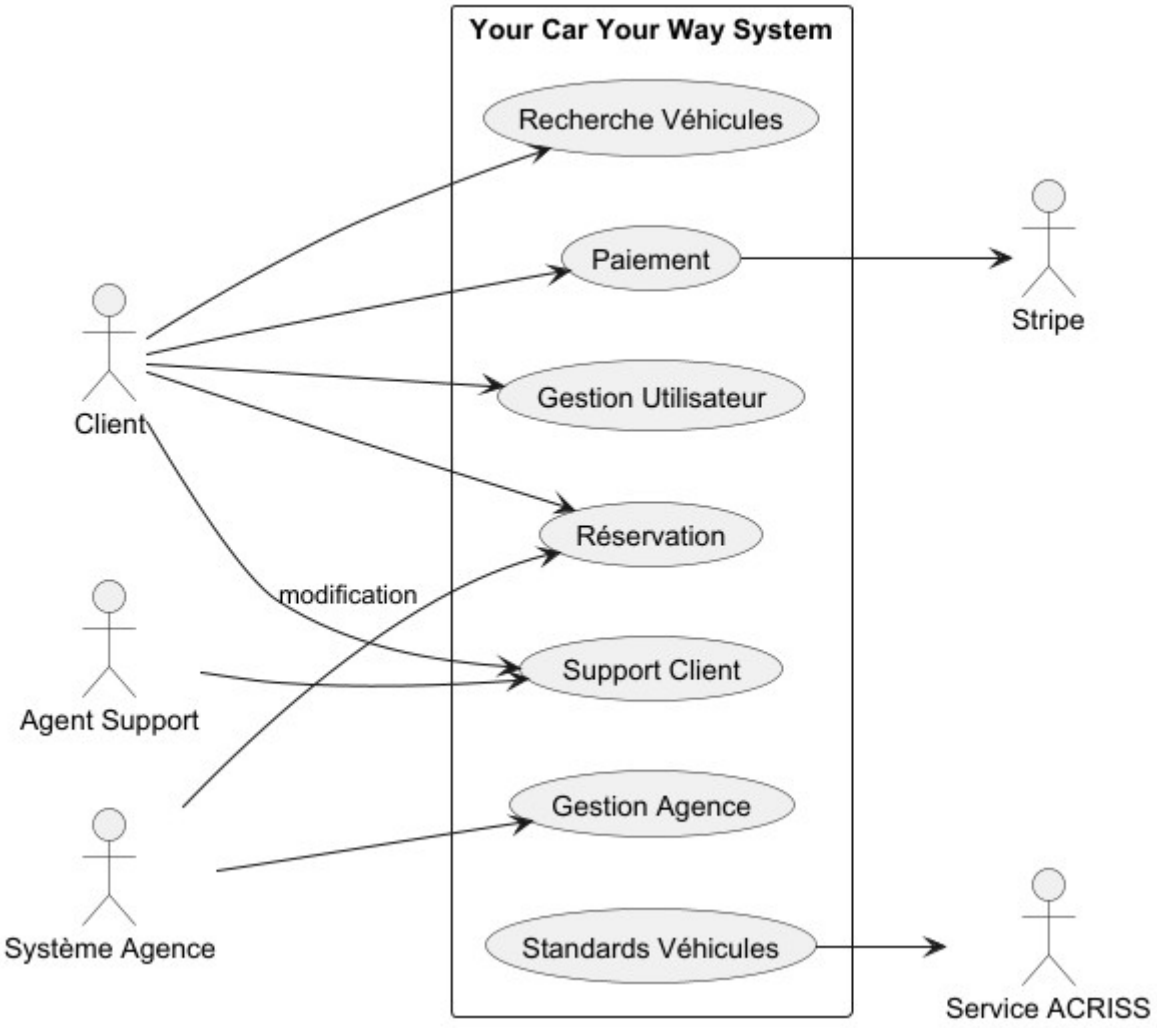
1.4.1. Vue Métier

La vue métier est modélisée par plusieurs diagrammes de cas d'utilisation, d'activités et de séquences UML.

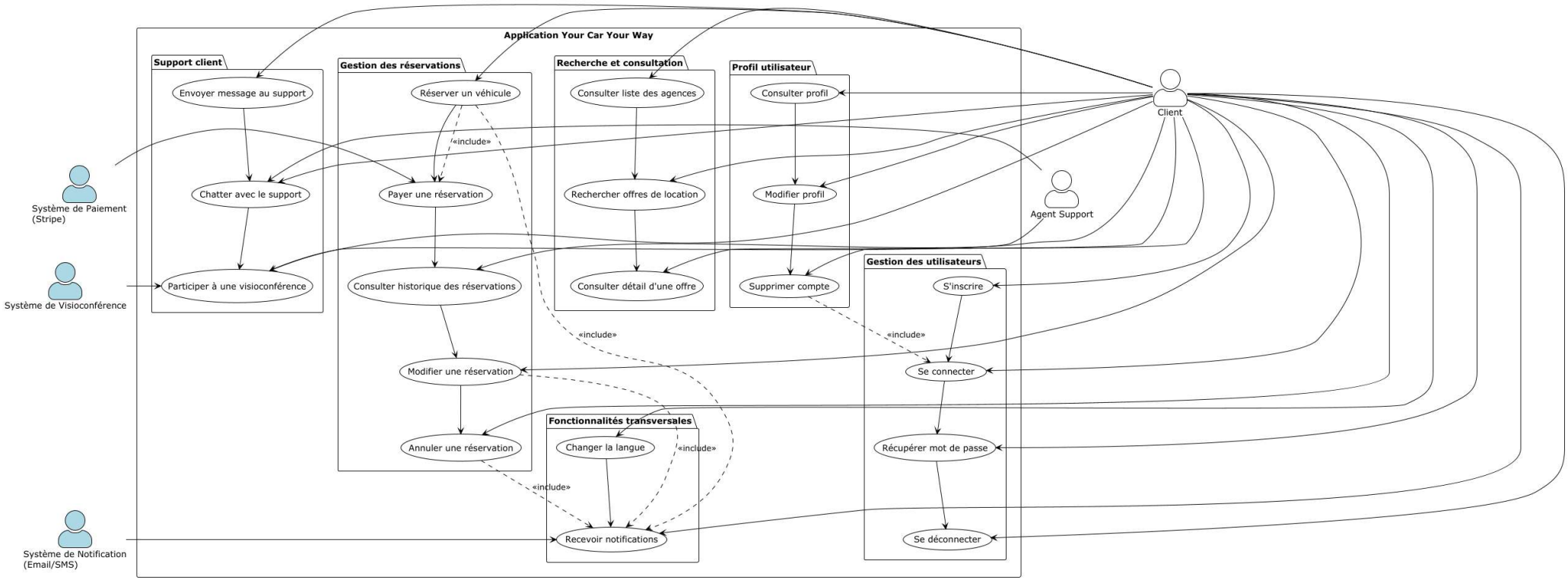
1.4.1.1. Acteurs du système

- **Client** : Utilisateur final effectuant des réservations
- **Agent Support** : Personnel du service client
- **Système d'Agence** : Applications utilisées en agence
- **Services Externes** : ACRISS, Stripe, services de géolocalisation

1.4.1.2. Cas d'utilisation – Vue générale



1.4.1.3. Vue détaillée des interactions



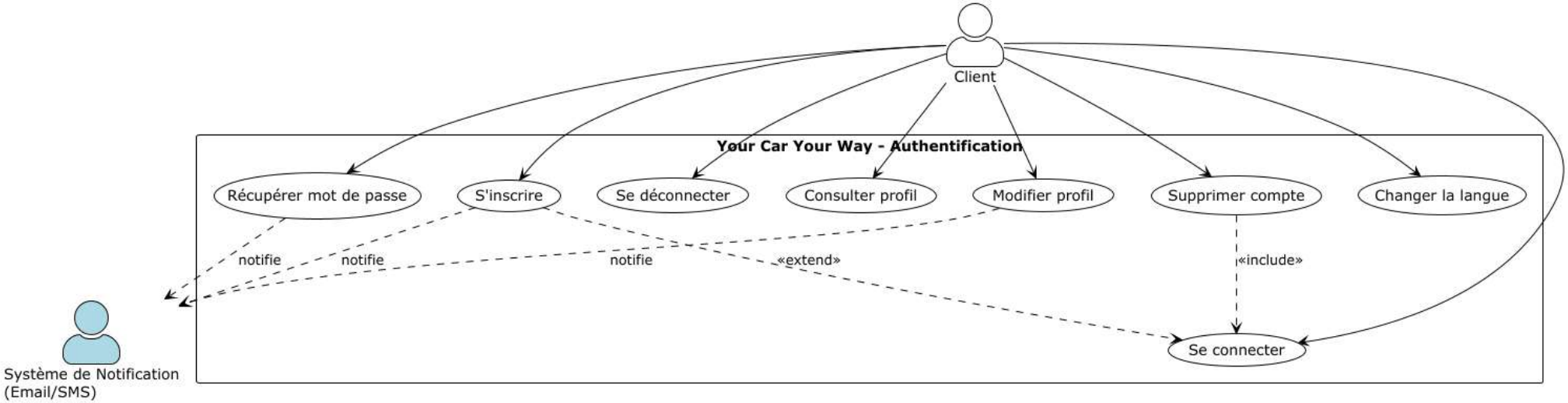
1.4.1.4. Fonctionnalités détaillées par domaine

<u>Gestion Utilisateur</u>

- Inscription, connexion, gestion profil
- Authentification multi-facteurs
- Préférences linguistiques et régionales

Use case : Processus d'inscription utilisateur

- `use-cases-user_register.puml` :

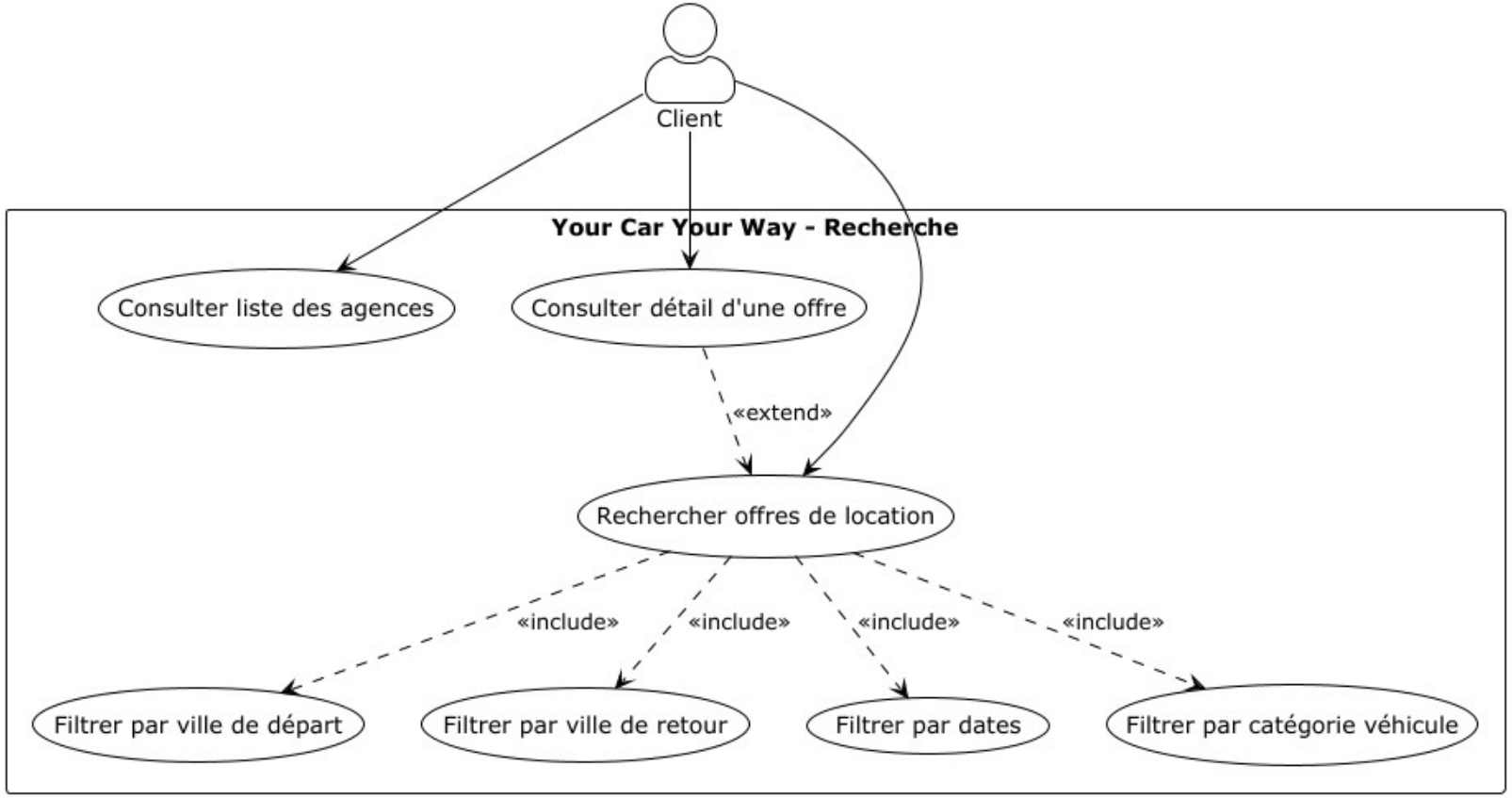


Gestion Véhicules

- Catalogue véhicules avec standards ACRISS
- Disponibilités en temps réel

Use case : Recherche de véhicules

- `use-cases-rental_car_search.puml` :

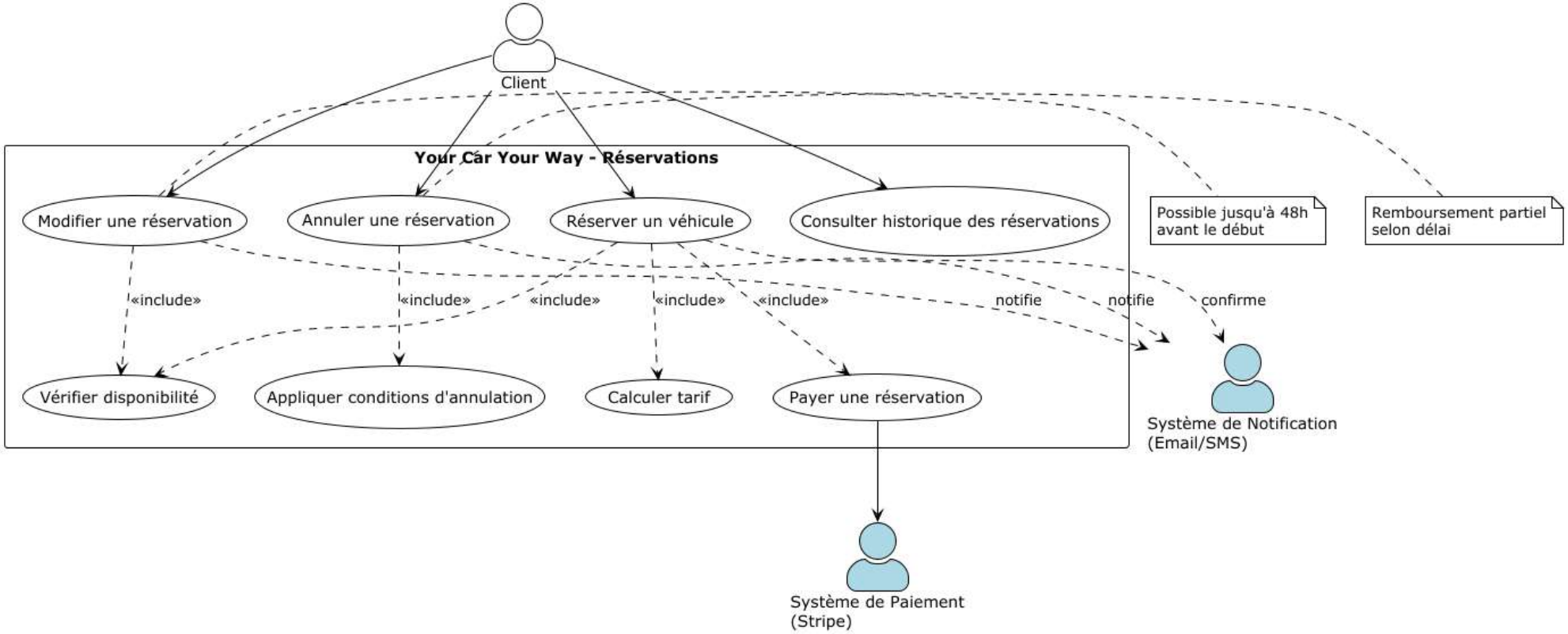


Réservation

- Processus de réservation complet
- Modification et annulation
- Historique et suivi

Use case : Réservation et paiement

- use-cases-reservation_and_payment.puml :



Support Client

- Chat en ligne et visioconférence
- Système de tickets
- Base de connaissances

Use case : Système de support client

- use-cases-support_system.puml :

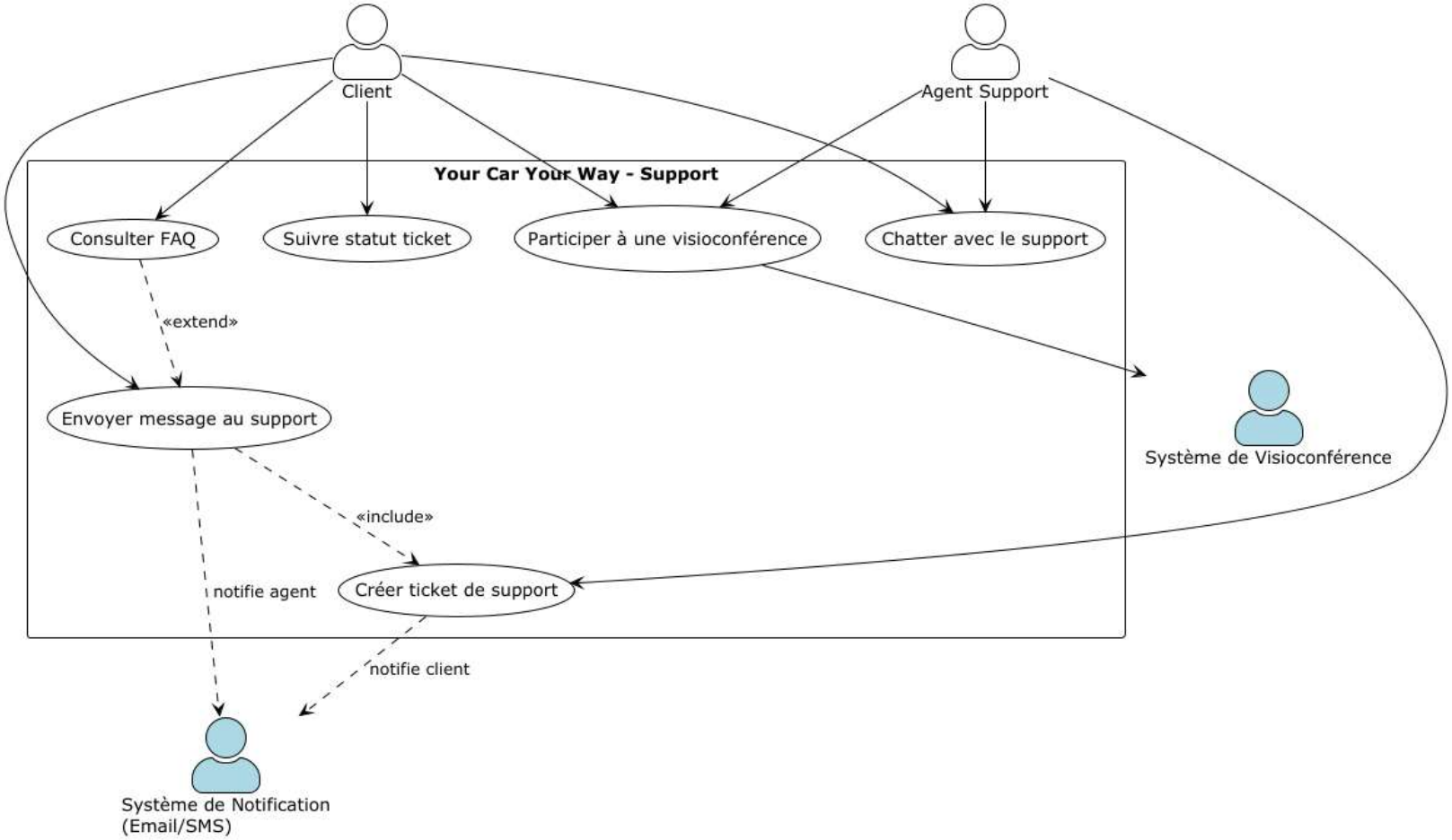


Diagramme d'activité : support client

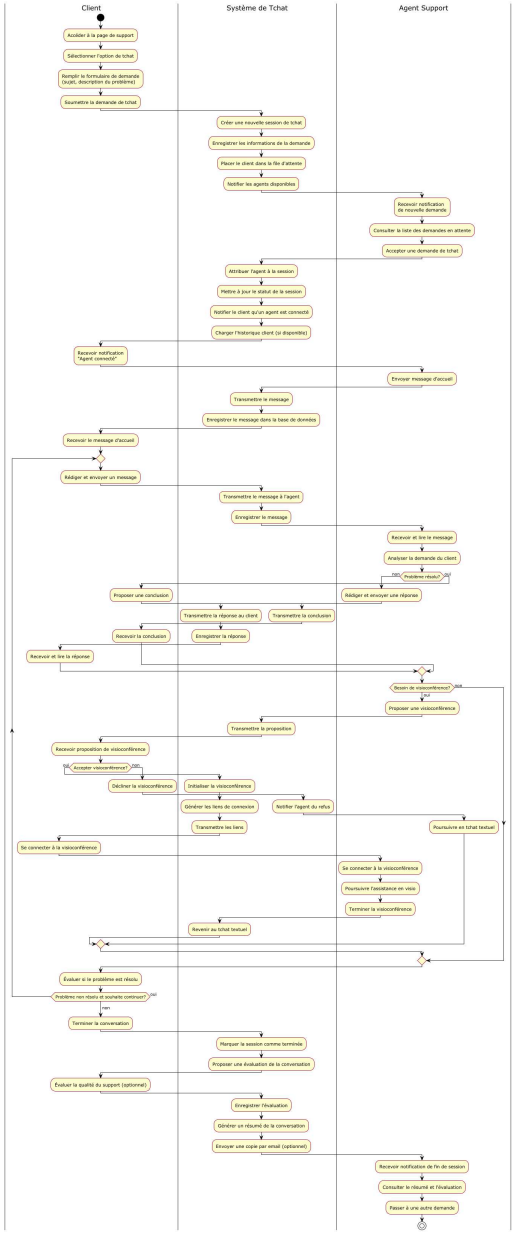
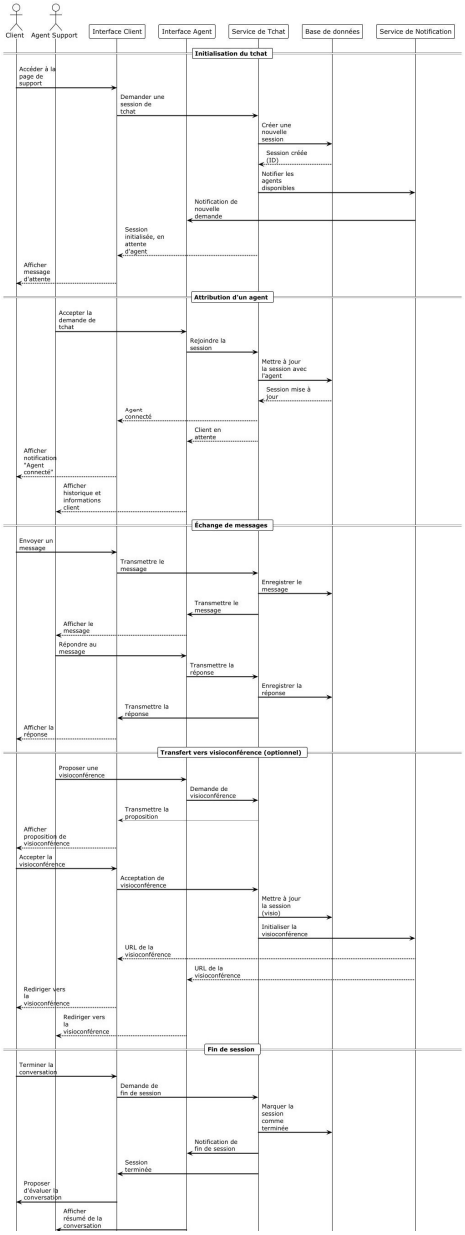
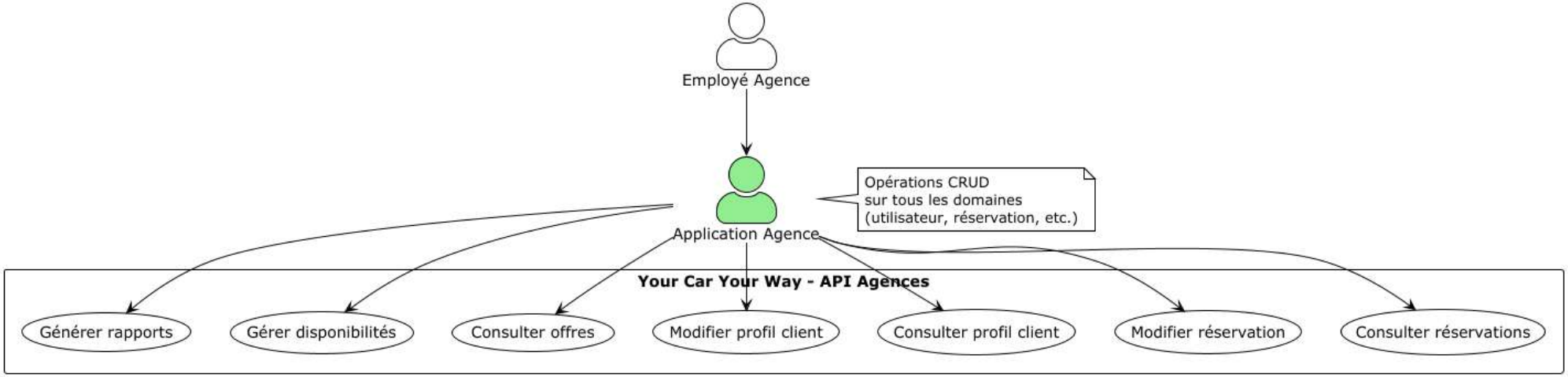


Diagramme de séquence : Session de tchat complète (avec visio)



Intégrations avec les SI des agences via les API



1.4.2. Vue de données

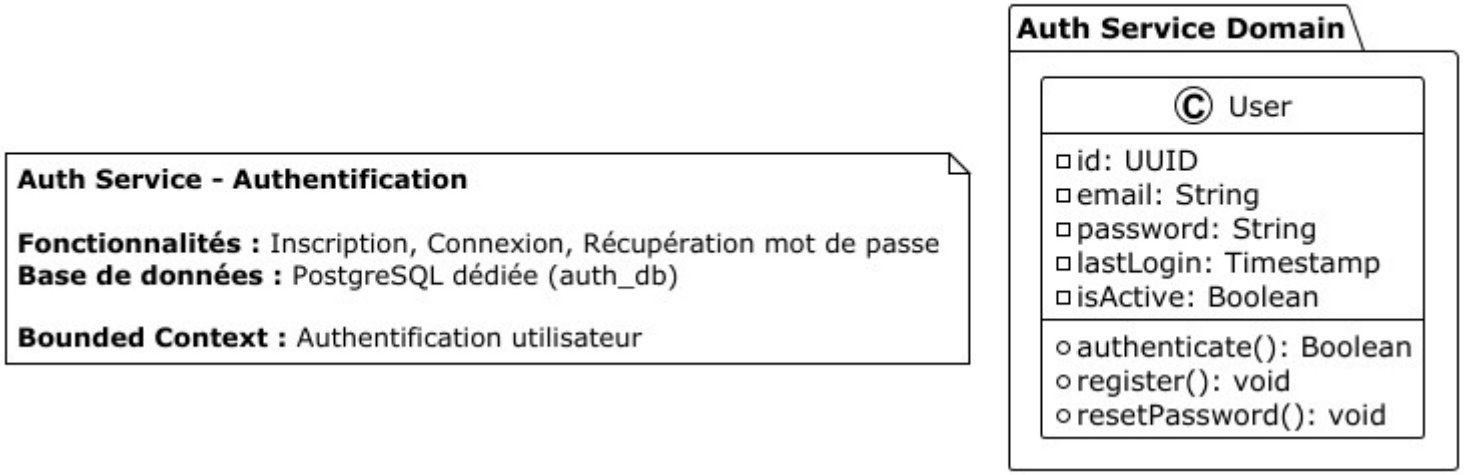
La vue des données est modélisée à travers des diagrammes de classes UML.

1.4.2.1. Modèle de domaine - Microservices

L'architecture Database-per-service implique une séparation claire des données par domaine :

Auth Service - Modèle de données

Auth Service - Modèle de Domaine



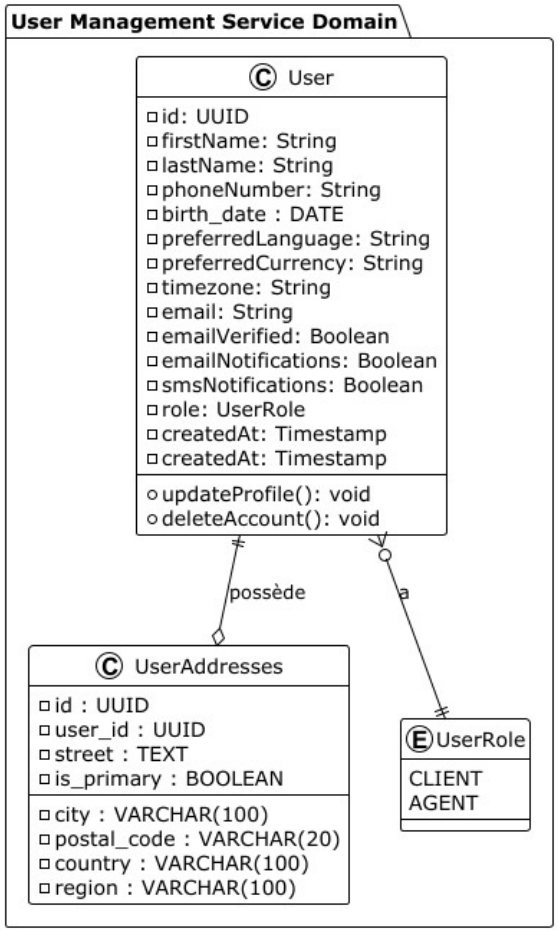
User Service - Modèle de donnée

User Management Service - Modèle de Domaine

User Service - Gestion des profils

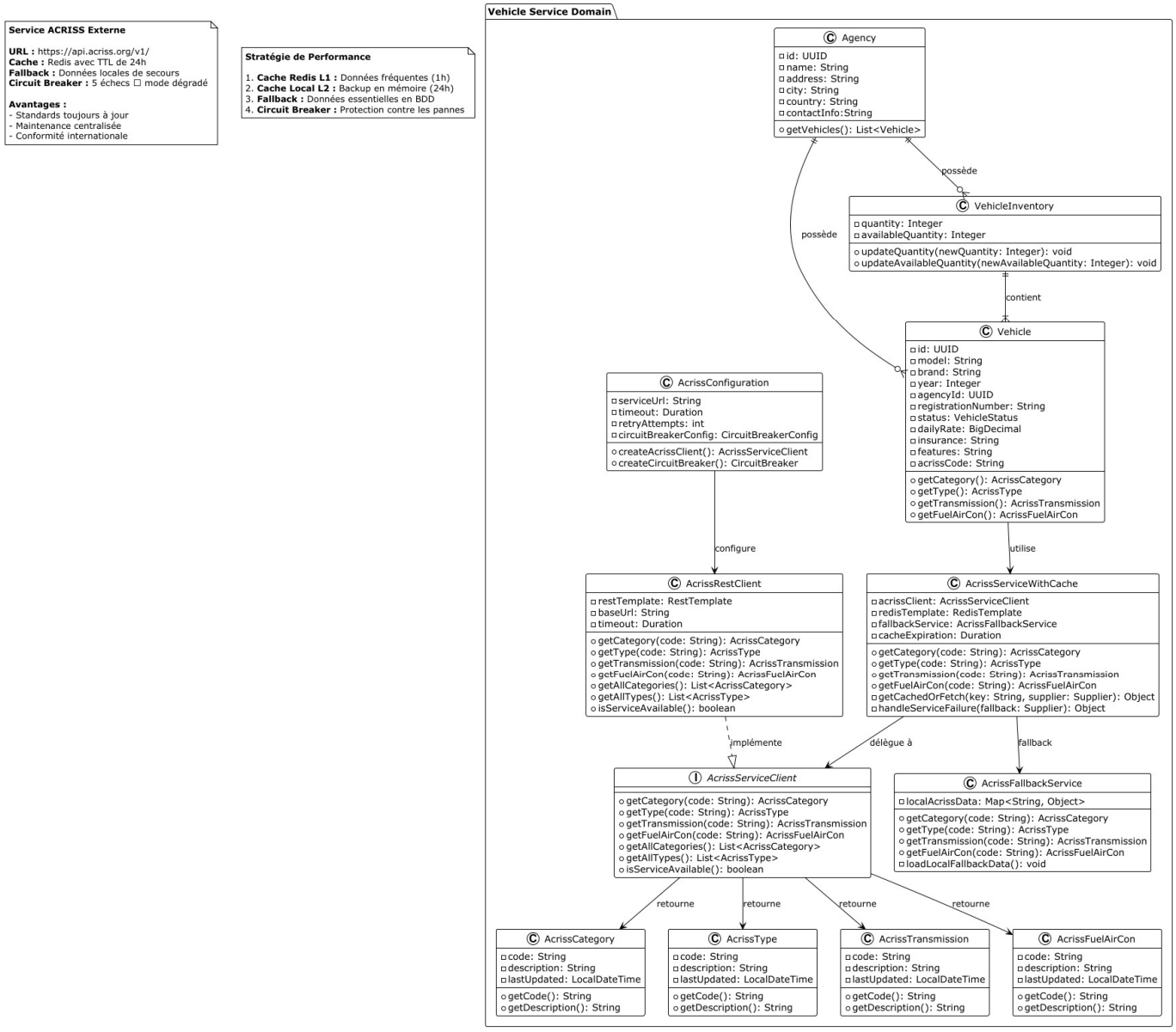
Fonctionnalités : Consulter, modifier et supprimer profil utilisateur
Données : Nom, prénom, adresse
Base de données : PostgreSQL dédiée (user_db)

Bounded Context : Profils utilisateur



Vehicle Service - Modèle de données avec Service ACRISS Externe

Vehicle Service - Modèle de Domaine avec Service ACRISS Externe



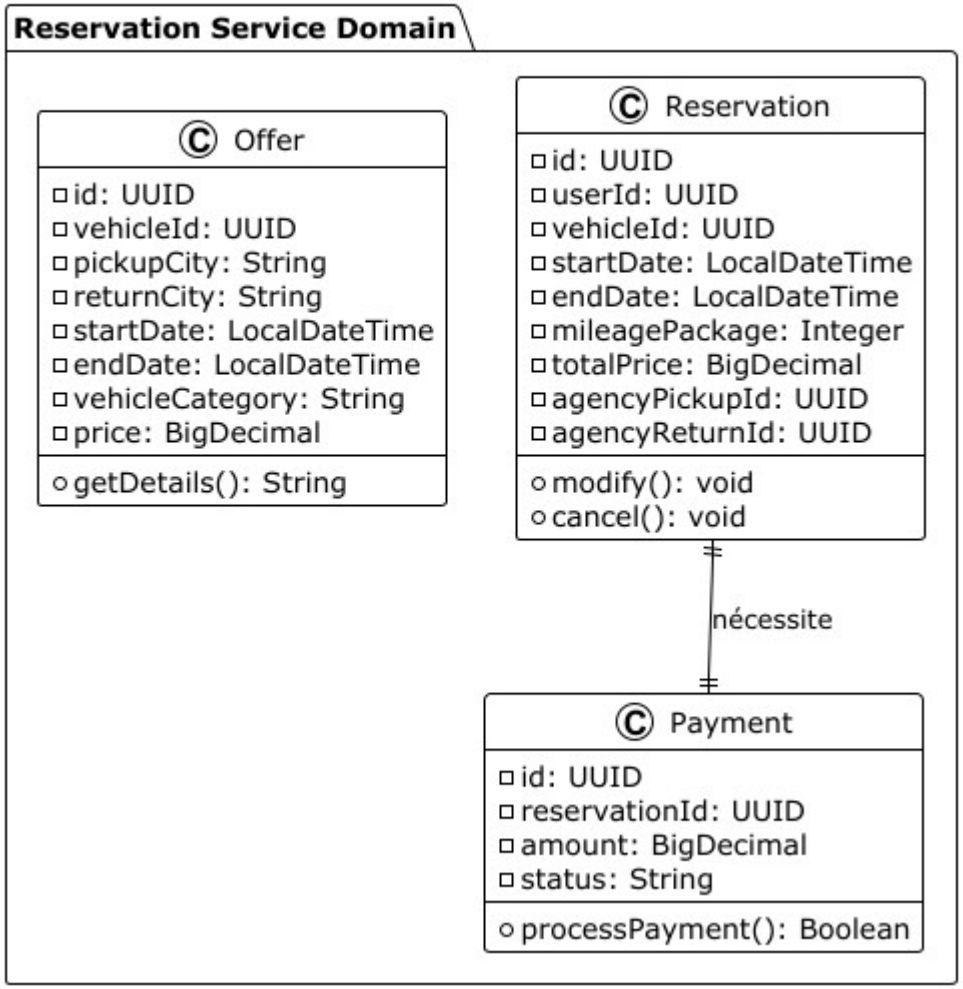
Reservation Service - Modèle de données

Reservation Service - Modèle de Domaine

Reservation Service - Gestion des réservations

Fonctionnalités : Offres, réservations, paiements externes (Stripe)
Règles : Modification jusqu'à 48h avant, remboursement 25% < 1 semaine
Base de données : PostgreSQL dédiée (reservation_db)

Bounded Context : Réservations et paiements



Support Service - Modèle de données (POC Chat)

Support Service - Modèle de Domaine (POC Chat Implémenté)

Support Service - Support client (POC Implémenté)

Fonctionnalités réalisées :
- Chat en temps réel via WebSocket
- API REST complète (CRUD)
- Gestion des utilisateurs (Client/Agent)
- Persistance PostgreSQL
- Sessions Redis
- Données de démonstration automatiques (Utilisateurs client et agent)

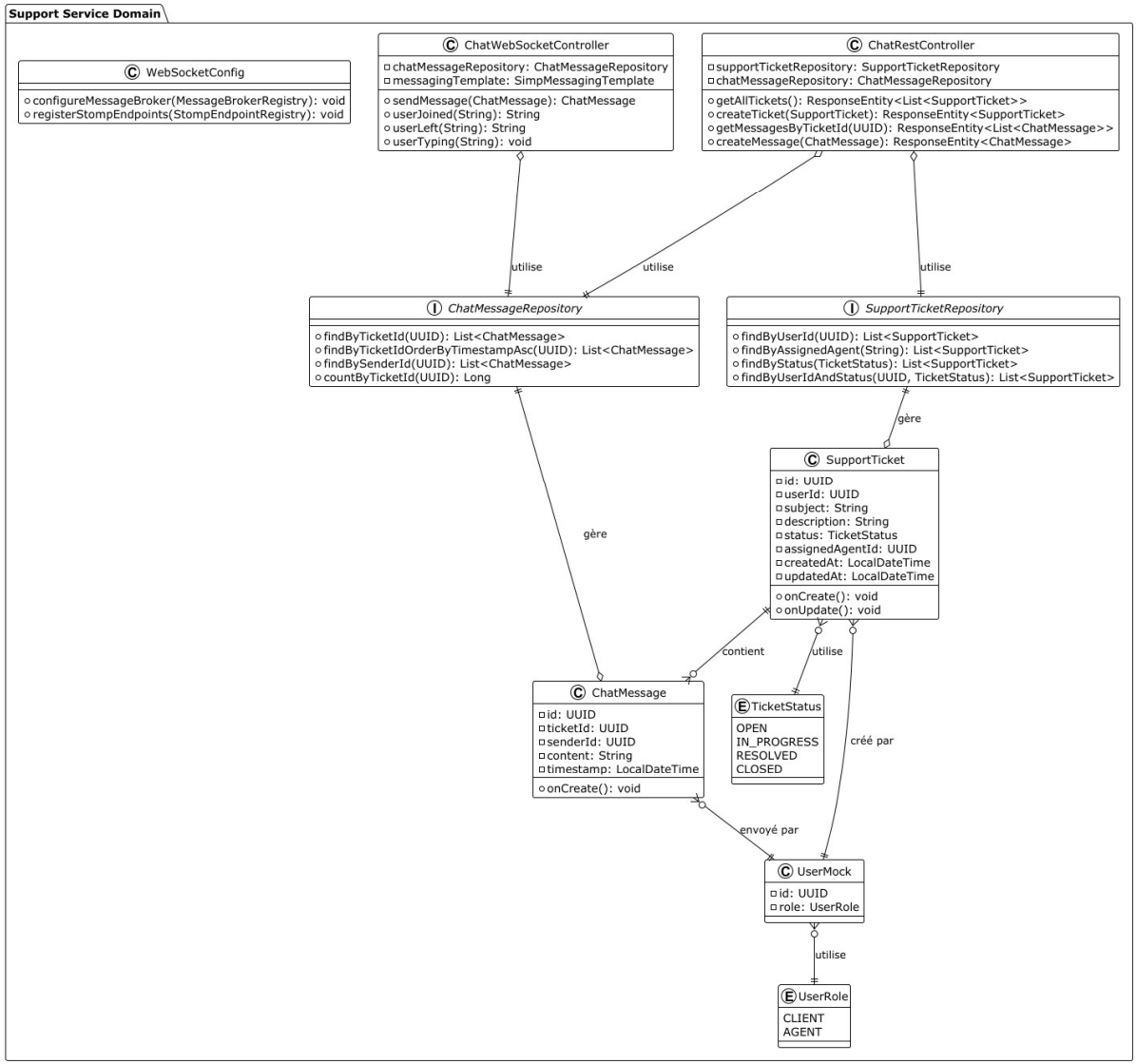
Technologies :
- Spring Boot 3.5.5
- WebSocket + STOMP
- JPA/Hibernate
- Lombok
- Docker Compose

Endpoints principaux :
- REST: /api/users, /api/chat/tickets, /api/chat/messages
- WebSocket: /ws, /app/chat.send, /topic/messages/{ticketId}

Bounded Context : Support client temps réel

Implémentation POC - Différences avec l'architecture finale :

- User mocké dans Support Service (simplifié pour POC)
- VideoSession non implémenté (focus sur chat texte)
- Communication inter-services non nécessaire
- Configuration WebSocket complète pour temps réel



Note : La ou les affectations d'agent(s) à un ticket de support sont tracés précisément par la brique audit trail (eventSourcing gérée par kafka).

1.4.2.2. Schémas de base de données par microservice

Chaque microservice possède sa propre base de données PostgreSQL :

- **auth_db** : Données d'authentification et sessions
- **user_db** : Profils utilisateurs et préférences
- **vehicle_db** : Catalogue véhicules et agences
- **reservation_db** : Réservations et options
- **support_db** : Tickets et conversations
- **notification_db** : Messages et historique notifications

Avantages de Database-per-service :

- Isolation des données par domaine
- Évolutivité indépendante
- Technologies de stockage optimisées par service
- Résilience (panne isolée)

Diagramme d'entités-relations : vue générale des databases pour chaque microservice

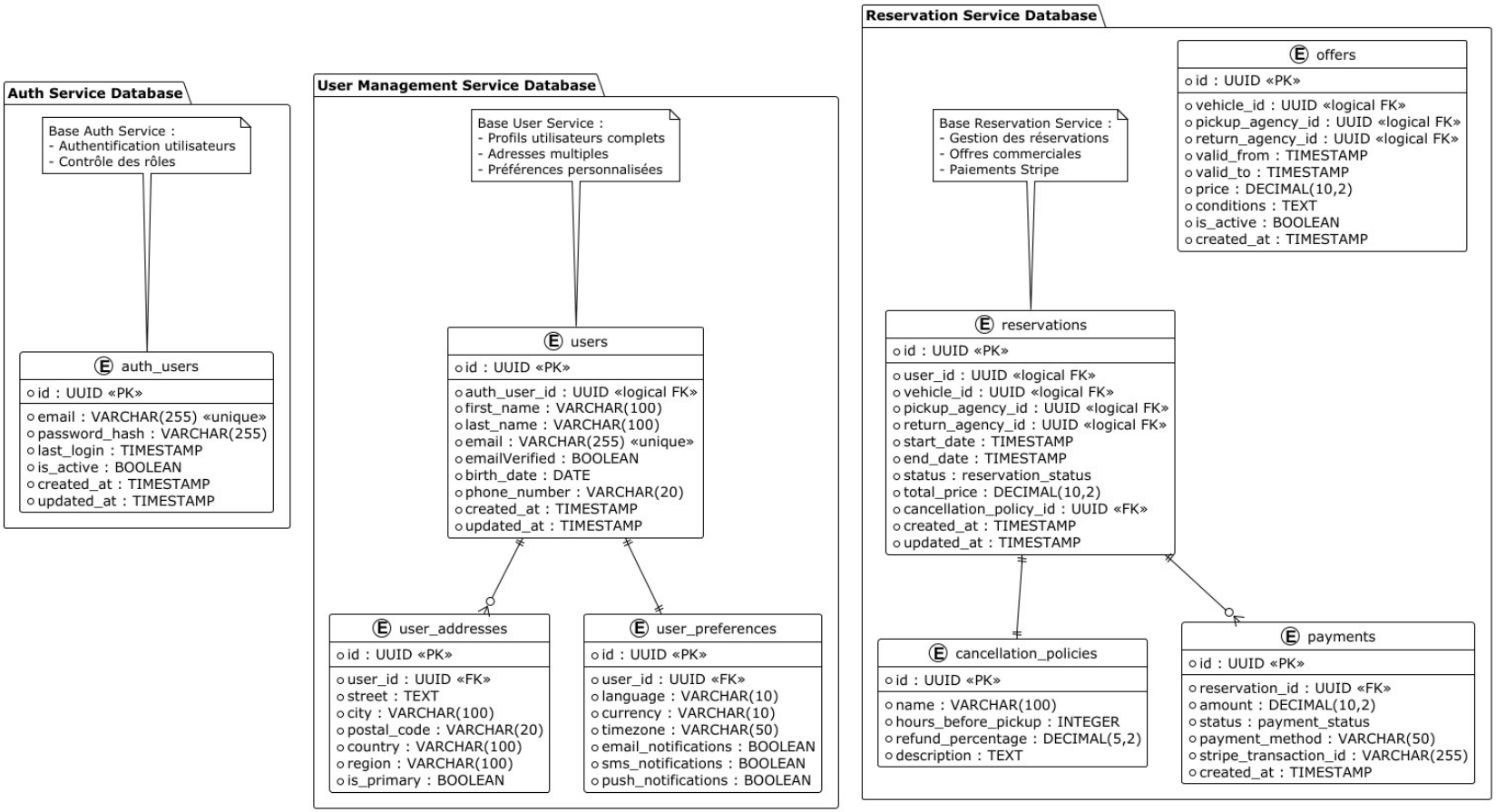
Relations logiques entre services :

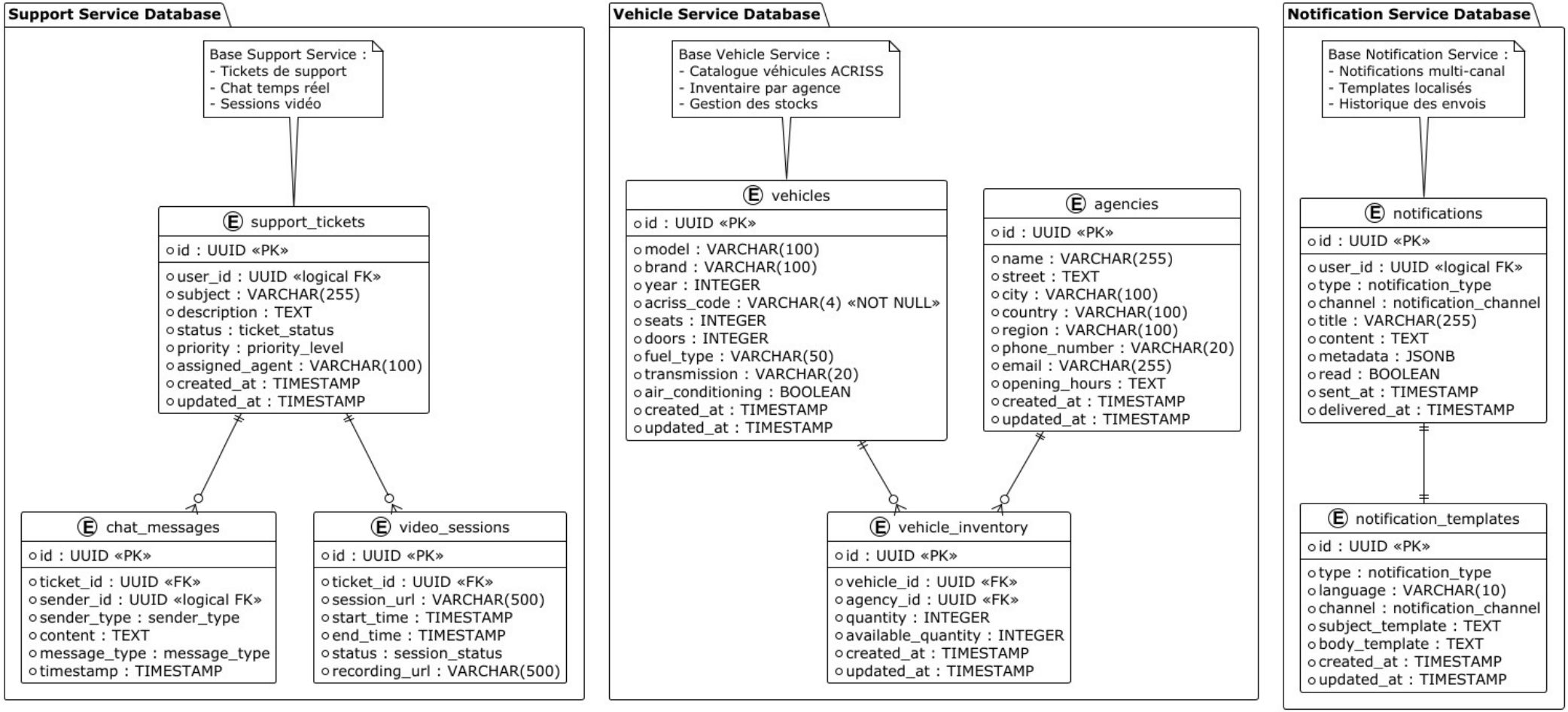
- auth_users.id = users.auth_user_id
- users.id = reservations.user_id
- vehicles.id = reservations.vehicle_id
- agencies.id = reservations.pickup_agency_id
- users.id = support_tickets.user_id
- users.id = notifications.user_id

Ces relations sont maintenues par les événements Kafka entre services

Types énumérés globaux :

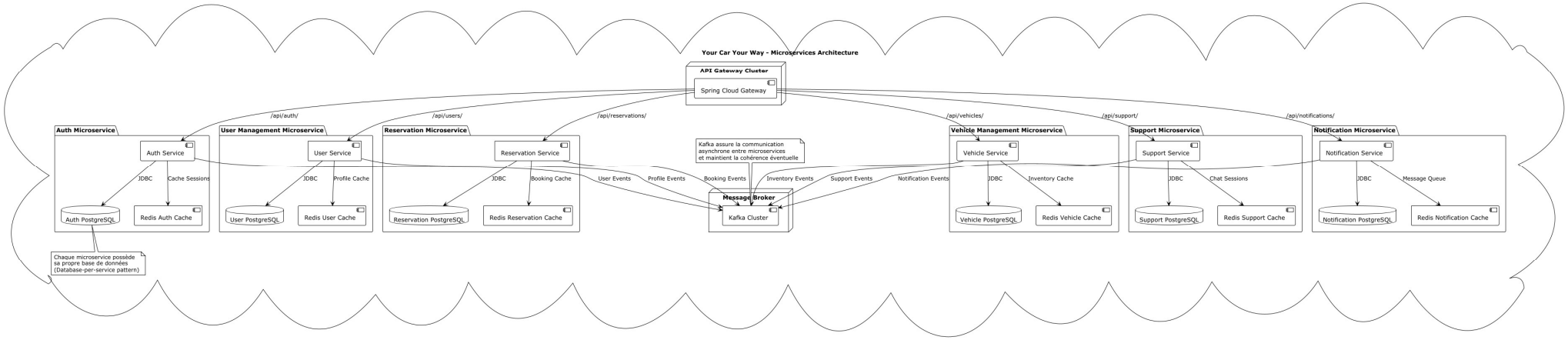
reservation_status: PENDING, CONFIRMED, IN_PROGRESS, COMPLETED, CANCELLED, MODIFIED
payment_status: PENDING, COMPLETED, FAILED, REFUNDED
ticket_status: OPEN, IN_PROGRESS, RESOLVED, CLOSED
priority_level: LOW, MEDIUM, HIGH, URGENT
sender_type: USER, AGENT, SYSTEM
message_type: TEXT, IMAGE, FILE, SYSTEM_MESSAGE
session_status: SCHEDULED, ACTIVE, ENDED, CANCELLED
notification_type: RESERVATION_CONFIRMED, RESERVATION_REMINDER, RESERVATION_CANCELLED, SUPPORT_MESSAGE, PROMOTIONAL, SYSTEM_ALERT
notification_channel: EMAIL, SMS, PUSH, IN_APP




Support Service Database
Base Support Service :
- Tickets de support
- Chat temps réel
- Sessions vidéo
support_tickets
id : UUID «PK»
user_id : UUID «logical FK»
subject : VARCHAR(255)
description : TEXT
status : ticket_status
priority : priority_level
assigned_agent : VARCHAR(100)
created_at : TIMESTAMP
updated_at : TIMESTAMP
chat_messages
id : UUID «PK»
ticket_id : UUID «FK»
sender_id : UUID «logical FK»
sender_type : sender_type
content : TEXT
message_type : message_type
timestamp : TIMESTAMP
video_sessions
id : UUID «PK»
ticket_id : UUID «FK»
session_url : VARCHAR(500)
start_time : TIMESTAMP
end_time : TIMESTAMP
status : session_status
recording_url : VARCHAR(500)
Vehicle Service Database
Base Vehicle Service :
- Catalogue véhicules ACRISS
- Inventaire par agence
- Gestion des stocks
vehicles
id : UUID «PK»
model : VARCHAR(100)
brand : VARCHAR(100)
year : INTEGER
acriss_code : VARCHAR(4) «NOT NULL»
seats : INTEGER
doors : INTEGER
fuel_type : VARCHAR(50)
transmission : VARCHAR(20)
air_conditioning : BOOLEAN
created_at : TIMESTAMP
updated_at : TIMESTAMP
agencies
id : UUID «PK»
name : VARCHAR(255)
street : TEXT
city : VARCHAR(100)
country : VARCHAR(100)
region : VARCHAR(100)
phone_number : VARCHAR(20)
email : VARCHAR(255)
opening_hours : TEXT
created_at : TIMESTAMP
updated_at : TIMESTAMP
vehicle_inventory
id : UUID «PK»
vehicle_id : UUID «FK»
agency_id : UUID «FK»
quantity : INTEGER
available_quantity : INTEGER
created_at : TIMESTAMP
updated_at : TIMESTAMP
Notification Service Database
Base Notification Service :
- Notifications multi-canal
- Templates localisés
- Historique des envois
notifications
id : UUID «PK»
user_id : UUID «logical FK»
type : notification_type
channel : notification_channel
title : VARCHAR(255)
content : TEXT
metadata : JSONB
read : BOOLEAN
sent_at : TIMESTAMP
delivered_at : TIMESTAMP
notification_templates
id : UUID «PK»
type : notification_type
language : VARCHAR(10)
channel : notification_channel
subject_template : TEXT
body_template : TEXT
created_at : TIMESTAMP
updated_at : TIMESTAMP

1.4.3. Vue Technologique

1.4.3.1. Architecture Microservices (vue « mono-région »)



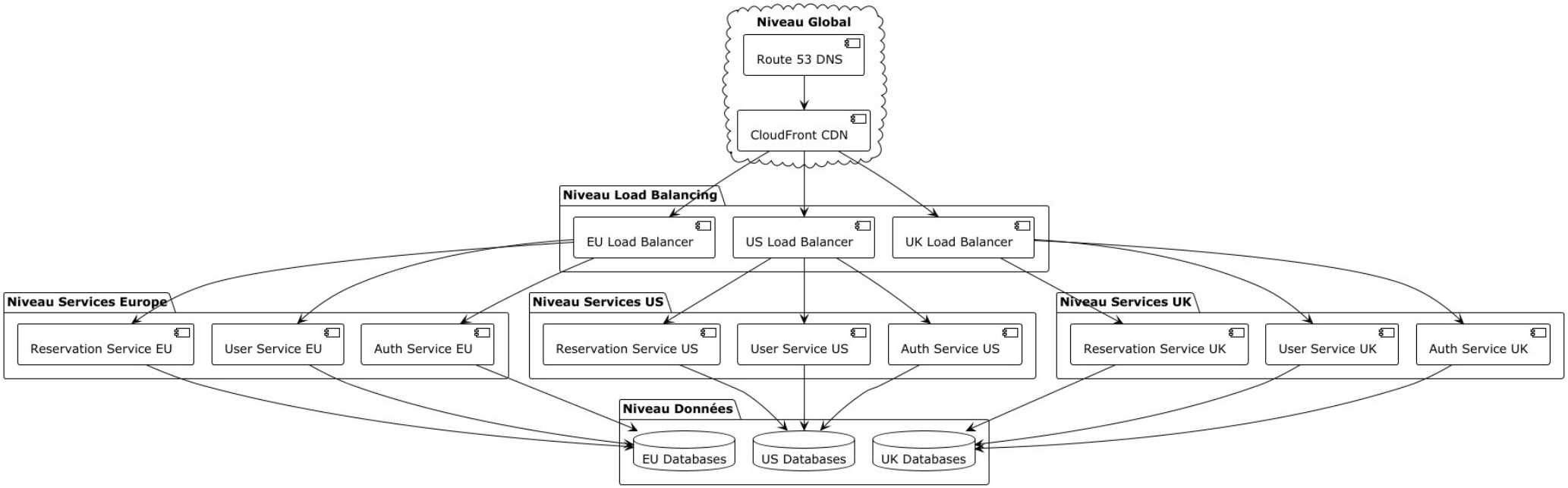
archi-microservices.puml

Plus d'informations sur :

- Intérêts de Spring Cloud Gateway dans l'architecture à la page 40.
- Rôle de kafka à la page 43.

1.4.3.2. Diagramme de déploiement multi-régional (simplifié)

Ce diagramme est volontairement simplifié (pour une approche progressive et pédagogique). Certains composants ne sont pas montrés, comme le Spring Cloud Gateway et le message broker kafka.

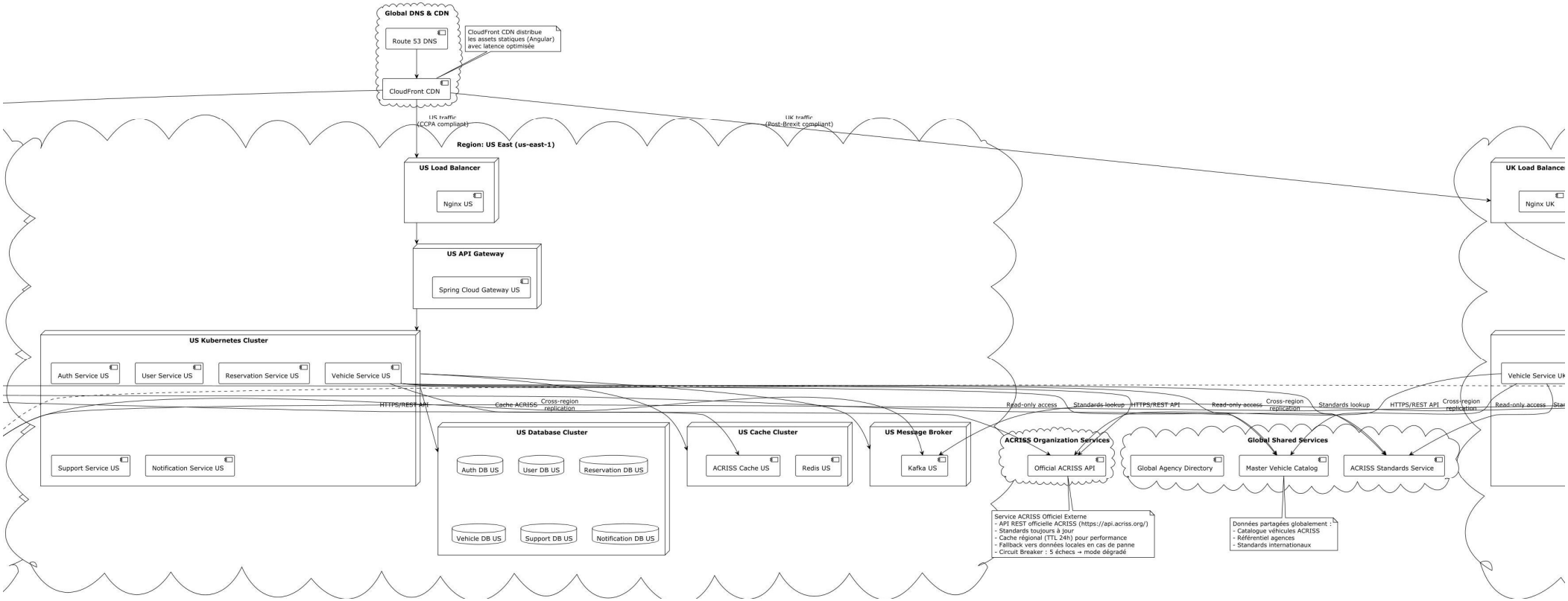


Plus d'informations sur :

- **Amazon Route 53** à la page 39.
- **Amazon CloudFront CDN** à la page 39
- **Équilibreurs de charge régionaux (HAProxy et NGINX)** à la page 39.

Vue détaillée du diagramme de déploiement complet (vue éclatée en 3 parties !)

Pas assez de largeur pour montrer le détail d'un coup dans son entièreté. Vue tronquée partie centrale (avec région US) :

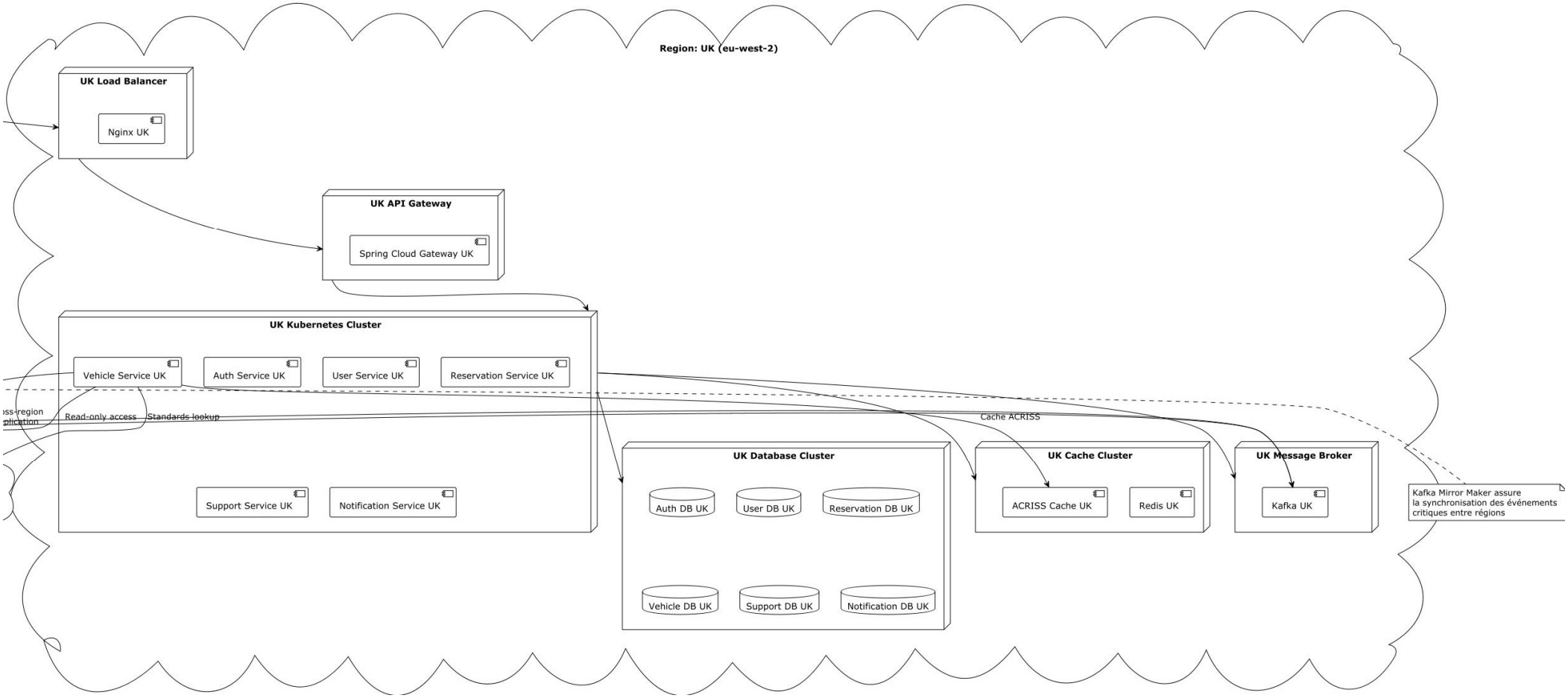


Plus d'informations :

- Kubernetes à la page 5.

- Redis et gestion ACRISS à la page 47. (sert aussi à maintenir les sessions de chat entre instances de nodes kubernetes)

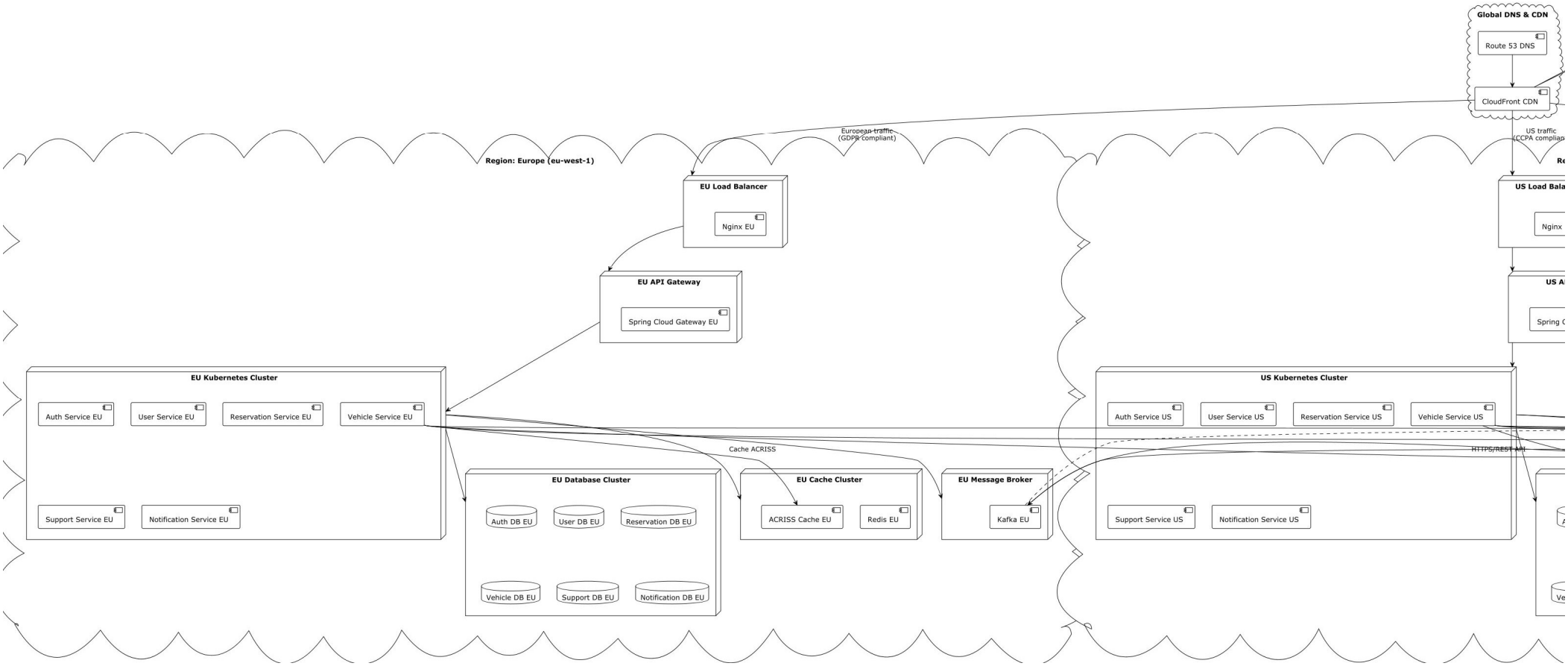
Vue tronquée partie gauche pour la fin du nuage UK :



Plus d’informations sur :

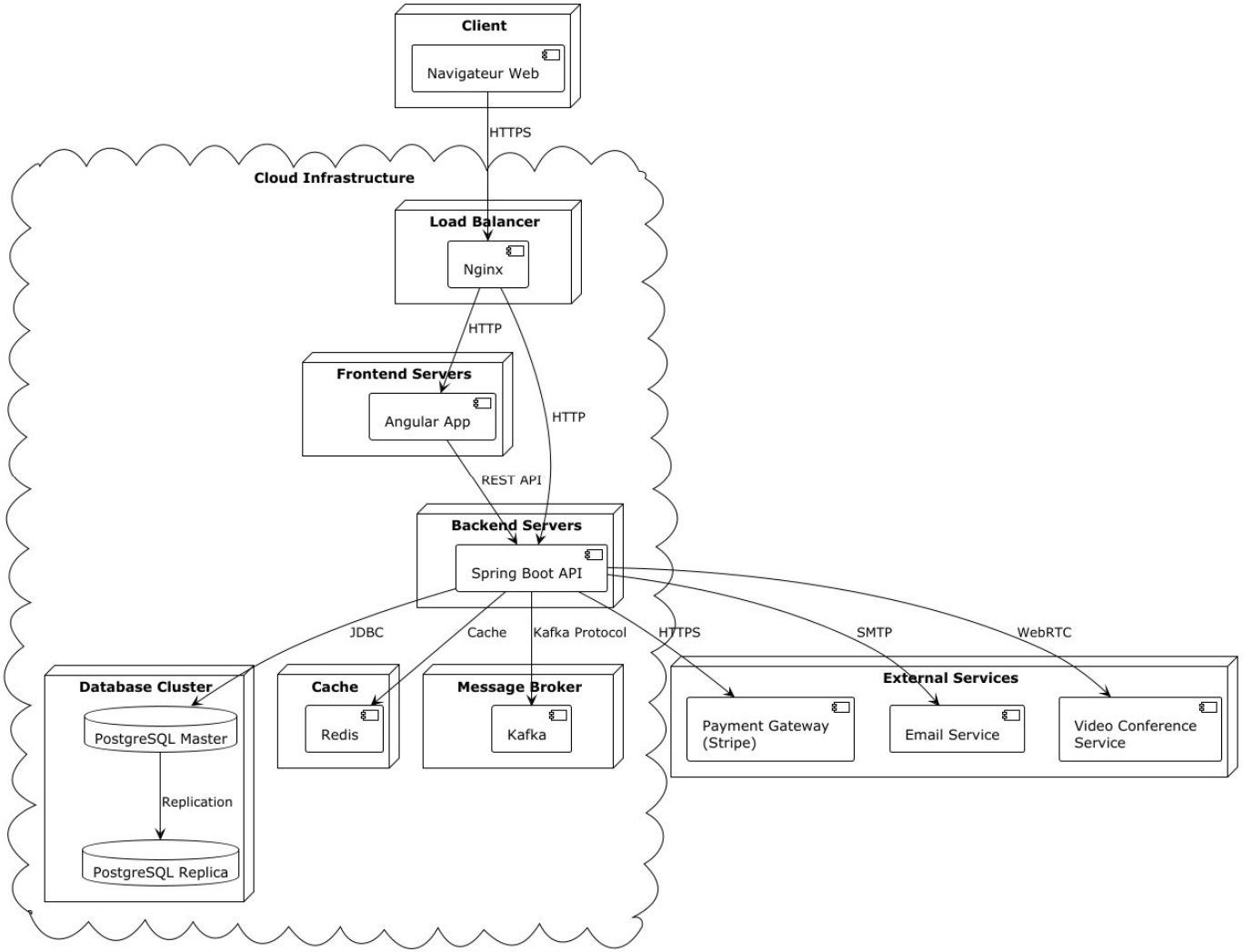
- Importance de Kafka Mirror Maker pour les événéments critiques à transférer entre régions à la page 43.

Vue tronquée partie droite (pour l'europe) :



1.4.3.3. Vision des protocoles d'échange

Ce diagramme de déploiement n'est pas la réalité du déploiement mais a pour seul but de communiquer sur les protocoles utilisés.



1.4.3.4. Stack technologique

Frontend

- **Framework** : Angular 20.2.1
- **UI Library** : Angular Material
- **Gestion d'états** : NgRx et Signal
- **Routing** : Angular Router avec lazy loading
- **Internationalisation** : Angular i18n
- **WebSocket** : Socket.io-client pour chat temps réel
- **Tests** : TestBed, Jest, Cypress (e2e)

Backend

- **Framework** : Spring Boot 3.5.5
- **Sécurité** : Spring Security + JWT
- **API Gateway** : Spring Cloud Gateway
- **Configuration** : Spring Cloud Config
- **WebSocket** : Spring WebSocket + STOMP
- **Monitoring** : Micrometer + Prometheus
- **Tests** : JUnit 5, Testcontainers

Base de données

- **SGBD Principal** : PostgreSQL 15+
- **Cache** : Redis 7+ (cluster mode)
- **Message Broker** : Apache Kafka 3+
- **Search** : Elasticsearch (catalogue véhicules)

1.4.3.5. Infrastructure

- **Conteneurisation** : Docker
- **Orchestration** : Kubernetes
- **Load Balancer** : Nginx + HAProxy
- **CDN** : CloudFront de AWS
- **DNS** : Route 53 de AWS - traduit les noms de domaine (ex: monapplication.com) en adresses IP, dirige la requête vers le bon serveur

1.4.3.6. DevOps

- **CI/CD** : Jenkins + GitLab CI
- **IaC** : Terraform + Ansible
- **Monitoring** : Prometheus + Grafana + Jaeger
- **Logs** : ELK Stack (Elasticsearch, Logstash, Kibana)

1.4.4. Intégrations externes

1.4.4.1. Service ACRISS Externe

- **URL** : https://api.acriss.org/v1/
- **Authentification** : API Key
- **Cache** : Redis (TTL 24h)
- **Fallback** : Données locales de secours
- **Circuit Breaker** : Spring Circuit breaker (5 échecs → mode dégradé)

1.4.4.2. Stripe Payment

- **API** : Stripe REST API v2023+
- **Webhooks** : Confirmation paiements

- **Sécurité** : PCI DSS compliant
- **Multi-région** : Entités Stripe par région

1.4.4.3. Services de communication

- **Email** : API SendGrid
- **SMS** : API Twilio
- **Visioconférence** : WebRTC + Jitsi Meet

1.5. Justification de l'approche architecturale

1.5.1. Réponses aux objectifs

Centralisation : L'architecture microservices avec API Gateway unique fournit un point d'entrée centralisé tout en maintenant la modularité.

Expérience utilisateur harmonisée : Le frontend Angular unique avec i18n assure une interface cohérente adaptée aux spécificités locales.

Scalabilité : Le déploiement multi-régional avec cache distribué et database-per-service permet de supporter 10,000+ utilisateurs simultanés.

Conformité : L'isolation des données par région et service respecte RGPD, CCPA et post-Brexit.

Intégration : Les API RESTful permettent l'intégration fluide avec les systèmes d'agence existants.

1.5.2. Respect des principes d'architecture

Modularité : Chaque microservice est autonome avec sa propre base de données, respectant le principe Database-per-service.

Scalabilité : Kubernetes permet l'auto-scaling horizontal de chaque service selon la charge.

Sécurité : JWT, HTTPS, chiffrement des données sensibles, audit trail complet.

Interopérabilité : Standards REST, OpenAPI, formats JSON, intégration ACRISS officielle.

Résilience : Circuit breakers, fallback, réplication multi-région, monitoring proactif.

Maintenabilité : Code propre, tests automatisés, CI/CD, documentation technique.

1.5.3. Suivi des normes et bonnes pratiques reconnues

Domain Driven Design (DDD) : Séparation claire par bounded context (User, Vehicle, Reservation, Support).

Microservices Patterns :
- Database-per-service
- API Gateway
- Event-driven architecture avec Kafka
- Circuit Breaker pattern

Security by Design :
- Authentification JWT + OAuth2
- Chiffrement end-to-end
- Audit et monitoring complet
- Conformité réglementaire intégrée

DevOps Best Practices :

- Infrastructure as Code

- CI/CD automatisé

- Monitoring observability-driven

- Blue-green deployments : stratégie de déploiement logiciel conçue pour réduire les temps d'arrêt et les risques lors de la mise en production de nouvelles versions d'une application. Une fois la nouvelle version déployée et testée sur l'environnement vert, le trafic est basculé du bleu (l'ancienne production) vers le vert.

Performance Optimization :

- Cache multi-niveau (L1: Redis – cache distribué, L2: Local)

- CDN pour assets statiques

- Database indexing optimisé

- Lazy loading et pagination

Patterns d'architecture :

- CQRS pour séparation lecture/écriture (Command Query Responsibility Segregation)

- Event Sourcing pour audit trail : au lieu de sauvegarder uniquement l'état actuel d'une entité (comme dans une base de données classique), on enregistre chaque événement qui a conduit à cet état. Les événements sont stockés dans un **event store** (base de messages optimisée pour les flux d'événements, comme Apache Kafka).

- Saga pattern pour transactions distribuées : permet de gérer des transactions longues et complexes qui s'étendent sur plusieurs services indépendants. Chaque transaction locale (microservice) est compensable : si une étape échoue, des actions compensatoires sont déclenchées pour annuler les effets des étapes précédentes.

- Strangler Fig pour migration progressive (on remplace les fonctionnalités de l'ancien système une par une en interceptant progressivement les requêtes destinées à l'ancien système.

1.5.4. Plus d'informations sur l'architecture et les composants

Notre architecture est conçue pour offrir une expérience utilisateur performante, sécurisée et conforme aux réglementations locales, tout en garantissant une haute disponibilité et une résilience aux pannes. Voici une description détaillée de notre architecture et des choix que nous avons faits.

1. Approche Multi-Régionale

Nous avons déployé notre application dans trois régions principales : Europe (EU), États-Unis (US), et Royaume-Uni (UK). Cette approche présente plusieurs avantages :

- **Latence réduite** : Les utilisateurs sont servis par la région la plus proche, ce qui réduit la latence et améliore l'expérience utilisateur.
- **Conformité réglementaire** : Chaque région gère ses propres données, ce qui nous permet de nous conformer aux réglementations locales telles que le GDPR en Europe, le CCPA aux États-Unis, et les réglementations post-Brexit au Royaume-Uni.

- **Résilience accrue** : En ayant des répliques de nos services et de nos données dans plusieurs régions, nous pouvons continuer à fonctionner même en cas de panne dans une région.

Le trafic est dirigé vers la bonne région grâce à un DNS global (Route 53) et un CDN (CloudFront). CloudFront distribue également les assets statiques (comme les fichiers de notre application Angular) avec une latence optimisée.

2. Load Balancing et API Gateway

Dans chaque région, nous avons un load balancer (avec Nginx) qui distribue le trafic entre les instances de nos microservices. De plus, nous utilisons Spring Cloud Gateway comme API Gateway dans chaque région. L'API Gateway joue un rôle crucial dans notre architecture :

- **Routage des requêtes** : Il achemine les requêtes entrantes vers le microservice approprié en fonction du chemin de l'URL (par exemple, /api/auth/ vers le Auth Service).
- **Sécurité** : Il centralise la gestion de la sécurité, comme l'authentification et l'autorisation, ce qui simplifie la configuration des microservices individuels.
- **Transformation des requêtes** : Il peut transformer les requêtes et les réponses pour adapter les formats de données entre les clients et les microservices.
- **Limitation du débit** : Il permet de limiter le débit des requêtes pour protéger nos services contre les surcharges.

3. Microservices et Kubernetes

Notre application est divisée en plusieurs microservices, chacun responsable d'une fonctionnalité spécifique :

- **Auth Service** : Gère l'authentification et l'autorisation des utilisateurs.
- **User Service** : Gère les informations des utilisateurs.
- **Reservation Service** : Gère les réservations de véhicules.
- **Vehicle Service** : Gère les informations sur les véhicules.
- **Support Service** : Gère les interactions de support avec les utilisateurs.
- **Notification Service** : Gère l'envoi de notifications aux utilisateurs.

Chaque microservice est déployé dans un cluster Kubernetes régional. Kubernetes nous offre plusieurs avantages :

- **Orchestration des conteneurs** : Il gère automatiquement le déploiement, la mise à l'échelle, et la récupération des conteneurs.
- **Scalabilité** : Nous pouvons facilement faire évoluer le nombre de répliques d'un microservice en fonction de la charge.
- **Résilience** : Kubernetes redémarre automatiquement les conteneurs qui échouent et redistribue les charges de travail en cas de défaillance d'un nœud.

4. Bases de Données et Caching

Chaque microservice a sa propre base de données PostgreSQL, suivant le pattern Database-per-service. Ce choix présente plusieurs avantages :

- **Couplage lâche** : Les microservices peuvent évoluer indépendamment sans se soucier des schémas de base de données des autres services.
- **Isolation des données** : Une panne dans une base de données n'affecte qu'un seul microservice.
- **Scalabilité** : Chaque base de données peut être mise à l'échelle indépendamment en fonction des besoins du microservice.

Nous utilisons des services de bases de données managés et clusterisés (comme AWS RDS) pour héberger ces bases de données. Ces services offrent une haute disponibilité, des sauvegardes automatiques, et une gestion simplifiée des bases de données. Les microservices accèdent à ces bases de données via des connexions JDBC, et les informations de connexion sont gérées de manière sécurisée via Kubernetes Secrets.

Pour améliorer les performances, chaque région dispose d'un cluster Redis pour le caching. Par exemple :

- Le Auth Service utilise Redis pour mettre en cache les sessions utilisateur.
- Le User Service utilise Redis pour mettre en cache les profils utilisateur.
- Le Vehicle Service utilise Redis pour mettre en cache les informations des véhicules.

5. Communication entre Microservices

La communication entre nos microservices se fait principalement via Kafka, un système de messagerie distribué. Nous avons choisi Kafka pour plusieurs raisons :

- **Découplage** : Les microservices ne communiquent pas directement entre eux, ce qui réduit le couplage et améliore la résilience.
- **Scalabilité** : Kafka peut gérer un grand volume de messages sans devenir un goulot d'étranglement.
- **Durabilité** : Les messages sont persistés sur disque, ce qui permet de les retrouver en cas de panne.

Voici comment cela fonctionne dans la pratique :

- Lorsqu'un événement significatif se produit dans un microservice (par exemple, la création d'un utilisateur), il publie un message sur un topic Kafka approprié.
- Les autres microservices intéressés par cet événement s'abonnent à ce topic et réagissent en conséquence.

Par exemple, lorsque le User Service crée un nouvel utilisateur, il publie un événement "UserCreated" sur Kafka. Le Auth Service écoute ce topic et crée un compte d'authentification pour le nouvel utilisateur. Le Notification Service écoute également ce topic et envoie un email de bienvenue à l'utilisateur.

Pour maintenir la cohérence des données entre les microservices, nous utilisons le pattern Saga pour les transactions distribuées. Par exemple, lorsqu'un utilisateur effectue une réservation, une saga est initiée :

1. Le Reservation Service envoie un événement "ReservationInitiated".
2. Le Vehicle Service écoute cet événement, vérifie la disponibilité du véhicule, et le marque comme provisoirement réservé s'il est disponible. Il publie ensuite un événement "VehicleReserved".
3. Le Reservation Service écoute l'événement "VehicleReserved" et crée la réservation dans sa base de données. Il publie ensuite un événement "ReservationConfirmed".
4. Si une étape échoue (par exemple, le véhicule n'est pas disponible), des événements de compensation sont publiés pour annuler les actions précédentes (par exemple, libérer le véhicule provisoirement réservé).

6. Services Globaux et Externes

En plus des services régionaux, nous avons des services globaux et externes :

- **Master Vehicle Catalog** : Un catalogue global des véhicules accessible en lecture seule par tous les microservices Vehicle régionaux. Cela garantit que tous les utilisateurs voient les mêmes informations sur les véhicules, quel que soit leur emplacement.

- **ACRISS Standards Service** : Un service qui fournit les standards ACRISS (un système de classification des véhicules) aux microservices Vehicle.
- **Global Agency Directory** : Un répertoire global des agences de location.
- **Official ACRISS API** : Une API externe qui fournit des informations à jour sur les standards ACRISS. Nos microservices Vehicle peuvent interroger cette API ou utiliser un cache local pour améliorer les performances.

7. Réplication et Résilience

Pour assurer la résilience et la cohérence des données entre les régions, nous avons mis en place une réplication cross-region pour Kafka. Cela signifie que les événements publiés dans une région sont répliqués dans les autres régions, ce qui permet aux microservices de toutes les régions de rester synchronisés.

En cas de panne dans une région, les autres régions peuvent continuer à fonctionner normalement. Lorsque la région défaillante revient en ligne, les événements manquants sont synchronisés via Kafka, ce qui permet de rétablir la cohérence des données.

8. Sécurité et Conformité

Notre architecture est conçue pour se conformer aux réglementations locales en matière de protection des données :

- **Europe (GDPR)** : Les données des utilisateurs européens sont stockées et traitées dans la région européenne.
- **États-Unis (CCPA)** : Les données des utilisateurs américains sont stockées et traitées dans la région américaine.
- **Royaume-Uni (Post-Brexit)** : Les données des utilisateurs britanniques sont stockées et traitées dans la région britannique.

De plus, nous utilisons des mécanismes de sécurité tels que :

- **Chiffrement des données** : Toutes les données sensibles sont chiffrées au repos et en transit.
- **Contrôle d'accès** : Les accès aux bases de données et aux services sont strictement contrôlés via des politiques d'accès.
- **Audit et surveillance** : Nous surveillons en continu notre infrastructure pour détecter et répondre aux incidents de sécurité.

9. Performance et Évolutivité

Notre architecture est conçue pour être hautement performante et évolutive :

- **Caching** : Nous utilisons Redis pour mettre en cache les données fréquemment accédées, ce qui réduit la charge sur les bases de données.
- **Load Balancing** : Les load balancers régionaux distribuent le trafic de manière équilibrée entre les instances de nos microservices.
- **Auto-scaling** : Kubernetes nous permet de faire évoluer automatiquement le nombre de répliques de nos microservices en fonction de la charge.
- **CDN** : CloudFront distribue les assets statiques avec une latence minimisée, ce qui améliore les temps de chargement pour les utilisateurs.

Conclusion

Notre architecture multi-régionale basée sur des microservices offre une combinaison puissante de performance, de résilience, de sécurité et de conformité. Grâce à Kubernetes, Kafka, et des services de bases de données managés, nous pouvons offrir une expérience utilisateur fluide et fiable, tout en respectant les réglementations locales et en assurant la disponibilité de notre service même en cas de pannes.

Chaque composant de notre architecture a été soigneusement choisi pour répondre à des besoins spécifiques, et leurs interactions sont conçues pour minimiser le couplage et maximiser la flexibilité. Cela nous permet d'évoluer rapidement en réponse aux besoins changeants de notre entreprise et de nos utilisateurs.

1.5.4.1. Amazon Route 53

Intérêts :

1. **Résolution DNS fiable et scalable** : Route 53 offre une infrastructure DNS mondiale qui traduit les noms de domaine en adresses IP avec une faible latence.
2. **Gestion de trafic avancée** : Il propose diverses politiques de routage comme le routage pondéré, basé sur la latence, géolocalisé, et de géo-proximité pour diriger le trafic de manière intelligente.
3. **Health Checks et Failover** : Il surveille l'état de santé des ressources et peut rediriger le trafic vers des ressources saines en cas de défaillance, assurant une haute disponibilité.

1.5.4.2. Amazon CloudFront CDN

Intérêts :

1. **Performance améliorée** : CloudFront utilise un réseau mondial de edge locations pour mettre en cache le contenu près des utilisateurs, réduisant ainsi la latence.
2. **Scalabilité et résilience** : Il gère des pics de trafic élevés et offre une protection contre certaines attaques DDoS.
3. **Sécurité améliorée** : CloudFront prend en charge le protocole HTTPS, l'intégration avec AWS Shield pour la protection contre les attaques DDoS, et permet de signer des cookies ou des URL pour contrôler l'accès.
4. **Coût optimisé** : En réduisant la charge sur les serveurs d'origine grâce à la mise en cache, CloudFront peut aider à réduire les coûts de bande passante et d'infrastructure.

1.5.4.3. Équilibreurs de charge régionaux (HAProxy et NGINX)

Intérêts :

1. **Répartition de la charge** : Ils répartissent le trafic entrant entre plusieurs serveurs backend, améliorant la performance et la disponibilité des applications.
2. **Haute disponibilité** : En cas de défaillance d'un serveur backend, ils redirigent le trafic vers des serveurs sains, maintenant ainsi la disponibilité de l'application.
3. **Terminaison SSL** : Ils peuvent terminer les connexions SSL/TLS, réduisant ainsi la charge sur les serveurs backend.
4. **Flexibilité et contrôle** : Ils offrent une grande flexibilité dans la configuration des algorithmes d'équilibrage de charge et des règles avancées pour le routage du trafic.
5. **Optimisation des ressources** : En répartissant la charge de manière efficace, ils aident à optimiser l'utilisation des ressources.
6. **Sécurité** : Ils fournissent une couche supplémentaire de sécurité en masquant la structure interne du réseau et en protégeant les serveurs backend des attaques directes.

4. Interaction entre les composants

Dans une architecture typique, ces composants interagissent de la manière suivante :

1. **Route 53** résout le nom de domaine en une adresse IP appropriée, en utilisant éventuellement des stratégies de routage avancées.
2. **CloudFront** distribue le contenu depuis des emplacements proches de l'utilisateur, améliorant ainsi la performance et réduisant la latence.
3. Les **équilibreurs de charge régionaux (HAProxy/NGINX)** répartissent le trafic entre plusieurs instances de serveurs backend, assurant ainsi la haute disponibilité et l'équilibrage de la charge.

Cette combinaison permet de construire une architecture web hautement performante, scalable, et résiliente, capable de servir des utilisateurs à l'échelle mondiale tout en maintenant une haute disponibilité et une sécurité robuste.

1.5.4.4. Intérêts de Spring Cloud Gateway dans l'architecture

Point d'entrée unique :

Dans chaque région (EU, US, UK), Spring Cloud Gateway sert de point d'entrée unique pour toutes les requêtes API.

Simplifie la configuration des clients, qui n'ont besoin de connaître qu'une seule URL par région.

Routage intelligent :

Masque la complexité du système aux clients en routant les requêtes vers les microservices appropriés.

Exemple : Une requête à /api/reservations est automatiquement routée vers le Reservation Service.

Sécurité centralisée :

Valide les tokens JWT, vérifie les rôles et permissions, et rejette les requêtes non autorisées.

Exemple : Protection des endpoints /api/admin/** pour les utilisateurs avec le rôle "admin".

Résilience améliorée :

En combinant avec le Load Balancer régional et le CDN global, notre architecture bénéficie d'une haute disponibilité.

Les Circuit Breakers protègent les microservices contre les cascades de pannes.

Gestion des échecs régionaux :

En cas de panne d'une région, le trafic peut être redirigé vers d'autres régions via le CDN.

La passerelle dans chaque région peut traiter les requêtes localement grâce aux données répliquées via Kafka Mirror Maker.

Observabilité améliorée :

Centralise les logs et métriques pour toutes les requêtes, facilitant la surveillance et le débogage.

Intègre avec des outils comme Prometheus, Grafana, et Zipkin pour le monitoring et le tracing distribué.

Flexibilité et extensibilité :

Peut être étendu avec des filtres personnalisés pour répondre à des besoins spécifiques.

Exemple : Ajout de filtres pour la gestion des régions ou la conformité aux réglementations locales (RGPD, CCPA).

Intégration avec Kubernetes :

S'intègre parfaitement avec l'environnement Kubernetes pour la découverte de services et l'équilibrage de charge.

Peut être déployé avec des Horizontal Pod Autoscalers pour une scalabilité automatique.

Exemple de flux de requête dans l'architecture :

Un utilisateur en Europe envoie une requête pour obtenir ses réservations.

Le DNS/CDN dirige la requête vers le Load Balancer européen (Nginx).

Nginx distribue la requête vers une instance de Spring Cloud Gateway EU.

La passerelle :

Valide le token JWT de l'utilisateur.

Applique le rate limiting (technique utilisée pour contrôler le nombre de requêtes qu'un client peut faire à une API ou à un service dans une période donnée => évite attaques ddos)

Route la requête vers le Reservation Service EU.

Le Reservation Service EU traite la requête et renvoie la réponse.

La passerelle peut appliquer des transformations à la réponse (comme l'ajout d'en-têtes de cache) avant de la renvoyer au client via Nginx et le CDN.

1.5.4.5. Kubernetes

Kubernetes est essentiel pour notre architecture de microservices pour plusieurs raisons, offrant une plateforme robuste pour le déploiement, la gestion, la mise à l'échelle, et la sécurisation des conteneurs.

Voici ses intérêts principaux :

- **1. Orchestration de Conteneurs**

Kubernetes automatise le déploiement, la mise à l'échelle, et la gestion des applications conteneurisées, ce qui est idéal pour une architecture de microservices où chaque service est déployé dans son propre conteneur.

- **2. Auto-réparation et Haute Disponibilité**

Kubernetes assure que nos applications fonctionnent comme prévu en redémarrant automatiquement les conteneurs défaillants, en rescheduling les conteneurs sur des nœuds sains en cas de défaillance, et en surveillant constamment l'état de santé des applications.

- **3. Scalabilité Horizontale**

Kubernetes facilite la mise à l'échelle automatique et manuelle des microservices en fonction de la demande, permettant aux services de gérer des charges accrues indépendamment les uns des autres.

- **4. Gestion des Configurations et des Secrets**

Avec Kubernetes, on peut gérer les configurations et les informations sensibles à l'aide de ConfigMaps et de Secrets, ce qui simplifie les déploiements et les mises à jour.

- **5. Découverte de Services et Équilibrage de Charge**

Kubernetes offre des mécanismes intégrés pour la découverte de services et l'équilibrage de charge, facilitant la communication entre les microservices et distribuant le trafic réseau de manière efficace.

- **6. Déploiements Progressifs**

Kubernetes permet des stratégies de déploiement avancées comme les déploiements progressifs (rolling updates), réduisant ainsi les risques et les temps d'arrêt lors des mises à jour de services.

- **7. Isolation et Sécurité**

Kubernetes améliore la sécurité et l'isolation des microservices en définissant des limites de ressources, en contrôlant le trafic réseau avec des politiques de réseau, et en gérant les identités et les accès.

- **8. Portabilité et Flexibilité**

Kubernetes fonctionne sur divers environnements cloud et locaux, offrant une grande flexibilité et évitant le verrouillage par un fournisseur spécifique (vendor lock-in).

- **9. Écosystème Riche**

Kubernetes bénéficie d'un écosystème riche avec des outils comme Helm pour la gestion des paquets, et des intégrations avec des solutions de monitoring et de logging comme Prometheus, Grafana, et ELK.

- **10. Optimisation des Coûts**

En permettant une meilleure utilisation des ressources matérielles grâce à l'orchestration intelligente des conteneurs, Kubernetes peut aider à réduire les coûts d'infrastructure.

Rôle de Kubernetes dans notre Architecture

Dans notre architecture multi-régions, Kubernetes joue un rôle central en :

- Déployant et gérant les microservices dans chaque région, assurant que chaque service est exécuté avec le nombre approprié de réplicas pour gérer la charge locale.
- Facilitant la communication entre les microservices à l'intérieur de chaque région et entre les régions.
- Assurant la haute disponibilité et la résilience des services en surveillant leur état et en réagissant aux défaillances.
- Permettant la mise à l'échelle automatique des services pour répondre à la demande locale, garantissant ainsi des performances optimales pour les utilisateurs dans chaque région.

En résumé, Kubernetes est crucial pour notre architecture microservices car il simplifie la gestion des environnements complexes et distribués, permettant aux équipes de se concentrer sur le développement de fonctionnalités plutôt que sur la gestion de l'infrastructure. Son écosystème riche et ses capacités d'automatisation avancées en font un choix idéal pour exécuter des architectures de microservices à grande échelle.

1.5.4.6. Rôle de kafka

Kafka agit comme un bus d'événements où chaque microservice peut publier des événements et s'abonner aux événements qui l'intéressent.

Cela permet aux microservices de communiquer de manière asynchrone et non bloquante.

Exemple Concret: Création d'une Réservation de voiture

Imaginons qu'un utilisateur crée une réservation de voiture. Voici comment Kafka facilite cette interaction:

- Reservation Service crée une nouvelle réservation dans sa base de données.

- Reservation Service publie un événement "Booking Created" sur Kafka.

- Notification Service écoute cet événement et envoie une notification à l'utilisateur confirmant la réservation.

- Vehicle Service écoute également cet événement et met à jour l'inventaire des véhicules pour refléter que la voiture est maintenant réservée.

Support Service pourrait également écouter cet événement pour créer un ticket de support préventif ou pour logs.

Avantages de Kafka dans ce contexte

Découplage: Les microservices n'ont pas besoin de se connaître directement. Ils communiquent via Kafka, ce qui réduit les dépendances directes entre eux.

Scalabilité: Kafka peut gérer un grand nombre de messages, ce qui permet de scaler individuellement chaque microservice sans affecter les autres.

Résilience: Si un microservice est temporairement indisponible, Kafka peut stocker les événements jusqu'à ce que le service soit à nouveau disponible, garantissant que aucun message n'est perdu.

Visualisation

Le diagramme montre que chaque microservice publie des événements spécifiques sur Kafka (par exemple, "User Events", "Profile Events", etc.). Kafka agit comme un hub central où tous ces événements sont gérés et distribués aux microservices intéressés.

En résumé, Kafka permet aux microservices de collaborer de manière cohérente en fournissant un moyen de communication asynchrone et fiable. Les événements publiés sur Kafka permettent aux différents services de réagir aux changements d'état dans le système de manière découplée mais coordonnée.

1.5.4.7. Importance de Kafka Mirror Maker

Kafka Mirror Maker est utilisé pour assurer la synchronisation des événements critiques entre les régions. Voici une explication détaillée de ce que sont les événements critiques dans notre contexte et comment cette synchronisation est mise en œuvre :

Événements critiques :

Dans le contexte de notre architecture de réservation de véhicules multi-régionale, les événements critiques sont des événements qui affectent l'état global du système et doivent être propagés de manière cohérente dans toutes les régions.

Exemples :

Événements liés aux utilisateurs :

UserCreated : Un nouvel utilisateur a été créé. Cet événement doit être propagé à toutes les régions pour que les autres services (comme Auth Service) puissent créer des comptes d'authentification pour le nouvel utilisateur.

UserUpdated : Les informations d'un utilisateur ont été mises à jour. Cela doit être propagé pour que tous les services utilisant ces informations restent synchronisés.

UserDeleted : Un utilisateur a été supprimé. Cela doit être propagé pour que tous les services puissent nettoyer leurs données associées à cet utilisateur.

Événements liés aux réservations :

ReservationCreated : Une nouvelle réservation a été effectuée. Cet événement doit être propagé pour que les services Vehicle puissent mettre à jour l'état des véhicules et que les services Notification puissent envoyer des confirmations aux utilisateurs.

ReservationUpdated : Une réservation a été mise à jour (par exemple, changement de dates ou de véhicule).

ReservationCancelled : Une réservation a été annulée. Cela doit être propagé pour libérer le véhicule et informer les autres services concernés.

Événements liés aux véhicules :

VehicleAdded : Un nouveau véhicule a été ajouté au catalogue. Cela doit être propagé pour que toutes les régions aient une vue cohérente du catalogue de véhicules.

VehicleUpdated : Les informations d'un véhicule ont été mises à jour (par exemple, changement de prix ou de disponibilité).

VehicleRemoved : Un véhicule a été retiré du catalogue.

Événements liés aux paiements :

PaymentProcessed : Un paiement a été traité avec succès. Cela doit être propagé pour mettre à jour l'état des réservations et des utilisateurs concernés.

PaymentFailed : Un paiement a échoué. Cela doit être propagé pour informer les autres services et éventuellement annuler la réservation associée.

Événements liés au support :

SupportTicketCreated : Un nouveau ticket de support a été créé. Cela doit être propagé pour que tous les agents de support puissent voir et répondre aux tickets.

SupportTicketUpdated : Un ticket de support a été mis à jour (par exemple, changement de statut ou ajout de commentaires).

SupportTicketClosed : Un ticket de support a été fermé. Cela doit être propagé pour informer les autres services et éventuellement déclencher des actions de suivi.

Pourquoi ces événements sont-ils critiques ?

Ces événements sont critiques car ils affectent l'état global du système et doivent être traités de manière cohérente dans toutes les régions. Par exemple :

Si une réservation est créée dans une région, mais que cette information n'est pas propagée aux autres régions, cela pourrait entraîner des incohérences (par exemple, un véhicule pourrait être réservé deux fois dans des régions différentes).

Si un utilisateur est créé dans une région, mais que cette information n'est pas propagée, l'utilisateur pourrait ne pas pouvoir se connecter ou effectuer des réservations dans d'autres régions.

Comment Kafka Mirror Maker assure la synchronisation des événements critiques :

Kafka Mirror Maker fonctionne en lisant les messages d'un cluster Kafka source et en les écrivant dans un cluster Kafka cible.

Voici comment cela est mis en œuvre dans notre architecture :

Configuration des clusters Kafka :

Chaque région a son propre cluster Kafka (EU_Kafka, US_Kafka, UK_Kafka).

Chaque cluster Kafka contient des topics où les microservices publient et consomment des événements.

Configuration de Mirror Maker :

Pour chaque région, un Mirror Maker est configuré pour répliquer les événements critiques vers les autres régions.

Par exemple, le Mirror Maker dans la région Europe (EU_Kafka) va répliquer les événements critiques vers les clusters Kafka des régions US et UK.

Sélection des événements critiques :

Seuls certains topics Kafka, contenant les événements critiques, sont répliqués entre les régions.

Par exemple, on pourrait avoir des topics comme user-events, reservation-events, etc., qui sont marqués comme critiques et donc répliqués.

Processus de réplication :

Mirror Maker lit les messages des topics critiques dans le cluster source (par exemple, EU_Kafka).

Il envoie ensuite ces messages aux clusters cibles (par exemple, US_Kafka et UK_Kafka).

Les microservices dans les autres régions consomment ces événements répliqués et mettent à jour leurs données locales en conséquence.

Gestion des conflits et de la cohérence :

Dans un système distribué, il peut y avoir des conflits ou des incohérences si les mêmes données sont modifiées dans différentes régions.

Pour gérer cela, on peut utiliser des stratégies comme le "last write wins" (la dernière écriture l'emporte) ou des mécanismes de résolution des conflits plus sophistiqués.

Dans notre architecture, Kafka Mirror Maker assure la propagation des événements, et chaque microservice est responsable de traiter ces événements de manière appropriée pour maintenir la cohérence locale.

Exemple concret de synchronisation d'un événement critique :

Imaginons qu'un utilisateur crée une réservation dans la région Europe.

Voici comment cet événement est synchronisé avec les autres régions :

Étape 1 : Création de la réservation

L'utilisateur effectue une réservation via l'application dans la région Europe.

Le Reservation Service EU traite la requête et crée une nouvelle réservation dans la base de données européenne (EU_ReservationDB).

Le Reservation Service EU publie ensuite un événement ReservationCreated sur le topic reservation-events dans le cluster Kafka européen (EU_Kafka).

Étape 2 : Réplication de l'événement

Kafka Mirror Maker, configuré pour répliquer les événements du topic reservation-events, lit l'événement ReservationCreated depuis EU_Kafka.

Mirror Maker envoie ensuite cet événement aux clusters Kafka des autres régions (US_Kafka et UK_Kafka).

Étape 3 : Traitement de l'événement dans d'autres régions

Dans la région US, le Reservation Service US consomme l'événement répliqué depuis US_Kafka.

Il met à jour la base de données US_ReservationDB pour refléter la nouvelle réservation.

Le Vehicle Service US fait de même pour mettre à jour l'état du véhicule dans US_VehicleDB.

Étape 4 : Traitement de l'événement dans la région UK

Le Reservation Service UK consomme l'événement répliqué depuis UK_Kafka.

Il met à jour la base de données UK_ReservationDB.

Le Vehicle Service UK met à jour l'état du véhicule dans UK_VehicleDB.

Étape 5 : Cohérence globale

Maintenant, toutes les régions ont une vue cohérente de la réservation.

Si l'utilisateur se connecte depuis une autre région, il verra sa réservation car celle-ci a été répliquée et traitée dans toutes les régions.

Avantages de cette approche :

Cohérence des données :

La réplication des événements critiques assure que toutes les régions ont une vue cohérente des données importantes, comme les réservations et les utilisateurs.

Résilience :

Même si une région devient indisponible, les autres régions peuvent continuer à fonctionner et à traiter les événements. Une fois la région indisponible de nouveau opérationnelle, elle peut synchroniser les événements manquants.

Évolutivité :

Cette approche permet d'ajouter facilement de nouvelles régions à l'architecture, car chaque nouvelle région peut simplement se connecter au système de réplication des événements critiques.

Performance :

Les événements non critiques peuvent être traités localement sans nécessiter de réplication, ce qui réduit la charge sur le système de réplication.

Gestion des conflits et de la cohérence éventuelle :

Dans un système distribué comme le vôtre, où les mêmes données peuvent être modifiées dans différentes régions, il est important de gérer les conflits et de maintenir la cohérence éventuelle (eventual consistency). Voici quelques stratégies pour y parvenir :

Horodatage et versionnement :

Chaque événement peut inclure un horodatage ou un numéro de version.

Lorsqu'un conflit est détecté (par exemple, deux événements modifiant la même réservation), on peut utiliser l'horodatage pour déterminer quel événement est le plus récent et doit être appliqué.

Idempotence des traitements :

Les consommateurs d'événements doivent être conçus pour être idempotents, c'est-à-dire que traiter le même événement plusieurs fois ne change pas l'état du système.

Cela peut être réalisé en utilisant des identifiants d'événements uniques et en vérifiant si un événement a déjà été traité avant de l'appliquer.

Compensation des transactions :

Dans certains cas, il peut être nécessaire d'annuler les effets d'un événement précédent. Par exemple, si une réservation est créée puis annulée, un événement de compensation peut être publié pour annuler la réservation.

Synchronisation périodique :

En plus de la réplication des événements, une synchronisation périodique des données entre les régions peut être mise en place pour corriger les éventuelles incohérences.

1.5.4.8. Redis et gestion ACRISS

Pour comprendre l'intérêt de Redis et de la gestion ACRISS dans Notre architecture, examinons chaque composant séparément et voyons comment ils peuvent s'intégrer et bénéficier à une architecture de microservices.

Redis

Redis est un magasin de données en mémoire open source qui est souvent utilisé comme base de données, cache, et broker de messages. Il est particulièrement connu pour sa haute performance, sa flexibilité, et son large éventail de structures de données.

Intérêts de Redis dans une architecture de microservices :

1. **Cache** :
   - **Amélioration des performances** : Redis est souvent utilisé comme un cache pour stocker des résultats de requêtes fréquentes ou coûteuses en calcul, réduisant ainsi la charge sur les bases de données principales et accélérant l'accès aux données souvent utilisées.
   - **Réduction de la latence** : En servant les données directement à partir de la mémoire (RAM), Redis peut réduire considérablement la latence pour les accès répétés aux mêmes données.
2. **Session Store** :
   - **Gestion des sessions utilisateur** : Redis peut être utilisé pour stocker les sessions utilisateur dans une architecture distribuée, ce qui est essentiel lorsque vos services sont déployés à travers plusieurs instances ou régions et doivent partager des informations de session.
3. **File d'attente et Broker de messages** :
   - **Gestion des tâches asynchrones** : Redis peut agir comme un broker de messages pour gérer des files d'attente de tâches ou des flux de travail asynchrones, utiles pour les opérations comme le traitement en arrière-plan ou la communication entre microservices.
   - **Support pour Pub/Sub** : Redis supporte la messagerie de type publie-abonne (Pub/Sub), ce qui peut être utilisé pour implémenter des architectures événementielles où les microservices communiquent via des événements.
4. **Base de données en mémoire** :

- **Stockage temporaire rapide** : Pour certaines applications qui ont besoin d'un accès très rapide aux données, Redis peut agir comme une base de données principale, généralement pour des données temporaires ou des cas d'utilisation spécifiques.

5. **Réplication et Persistance** :
   - **Durabilité** : Bien que Redis soit une base de données en mémoire, il offre des options pour la persistance des données sur disque, ainsi que des mécanismes de réplication pour assurer la durabilité et la disponibilité des données.
6. **Partitionnement et Scalabilité** :
   - **Scalabilité horizontale** : Redis peut être configuré pour le sharding (partitionnement), ce qui permet de scaler horizontalement en distribuant les données sur plusieurs nœuds Redis.

Gestion ACRISS

ACRISS (Association of Car Rental Industry Systems Standards) est une norme internationale largement utilisée dans l'industrie de la location de voitures pour classer et décrire les types de véhicules.

Intérêts de la gestion ACRISS dans notre architecture :

1. **Standardisation des données** :
   - ACRISS fournit un moyen standardisé de décrire les véhicules, facilitant la compréhension et la comparaison des véhicules à travers différents systèmes et régions.
   - Cela assure une cohérence dans la façon dont les informations sur les véhicules sont stockées et présentées, ce qui est essentiel dans un contexte international ou multi-régional.
2. **Interopérabilité** :
   - En utilisant une norme comme ACRISS, différents systèmes (comme ceux de partenaires ou de fournisseurs dans l'industrie de la location) peuvent facilement échanger et comprendre des informations sur les véhicules sans nécessiter de conversion complexe ou de mappage personnalisé.
3. **Amélioration de l'expérience utilisateur** :
   - Pour les clients, voir des descriptions de véhicules standardisées (comme "Economy", "Compact", "Luxury") rend l'expérience de réservation plus intuitive et compréhensible, surtout lorsqu'ils naviguent sur des plateformes internationales.
4. **Optimisation des opérations** :
   - La standardisation permet de simplifier les processus internes, comme la gestion des flottes de véhicules, la tarification, et la disponibilité.
   - Elle peut également faciliter l'intégration avec des systèmes externes (comme les systèmes de gestion de flottes ou de réservation).
5. **Flexibilité et extensibilité** :
   - Bien qu'ACRISS soit une norme, elle est conçue pour être flexible, permettant d'ajouter des extensions ou des détails spécifiques aux besoins d'une entreprise sans perdre l'interopérabilité de base.

Intégration de Redis et de la gestion ACRISS dans Notre architecture

Dans notre architecture de microservices, Redis et la gestion ACRISS pourraient être intégrés de la manière suivante :

1. **Redis pour la gestion des sessions et le cache** :
   - **Cache des classifications ACRISS** : Les descriptions de véhicules et leurs classifications ACRISS peuvent être mises en cache dans Redis pour un accès rapide. Cela est particulièrement utile si ces informations sont fréquemment accédées, comme dans un système de réservation où les clients naviguent entre différentes catégories de véhicules.
   - **Sessions utilisateur** : Si notre application web nécessite une gestion des sessions (par exemple, pour garder une trace des préférences de l'utilisateur ou du panier de réservation), Redis pourrait être utilisé pour stocker ces sessions.
2. **Gestion ACRISS dans le microservice de véhicules** :
   - Un microservice dédié pourrait être responsable de la gestion des informations sur les véhicules, incluant leurs classifications ACRISS. Ce service pourrait exposer une API pour récupérer les informations sur les véhicules par leur code ACRISS.
   - Ce microservice pourrait utiliser Redis pour cacher les requêtes fréquentes, comme la liste des véhicules disponibles pour une catégorie ACRISS spécifique.
3. **Architecture de microservices intégrant Redis et ACRISS** :

Exemple de Scénario d'Intégration

Imaginons un scénario où un utilisateur effectue une réservation de véhicule :

1. **Accès à l'application** : L'utilisateur se connecte à l'application, et le User Service authentifie l'utilisateur.
   Les détails de la session sont stockés dans Redis pour être accessibles par d'autres services si nécessaire.
2. **Navigation et sélection de véhicules** :
   - L'utilisateur navigue vers la section des véhicules disponibles. Le frontend envoie une requête à l'API Gateway, qui la route vers le Vehicle Service.
   - Le Vehicle Service vérifie d'abord Redis pour voir si les informations sur les véhicules (classées par ACRISS) sont déjà en cache. Si c'est le cas, il les retourne immédiatement.
   - Si les données ne sont pas en cache, le Vehicle Service interroge la base de données principale pour obtenir les informations, puis les stocke dans Redis pour les requêtes futures.
3. **Réservation d'un véhicule** :
   - L'utilisateur sélectionne un véhicule et fait une réservation. Le frontend envoie une requête à l'API Gateway, qui la route vers le Reservation Service.
   - Le Reservation Service vérifie la disponibilité du véhicule auprès du Vehicle Service et procède à la réservation.
   - Des détails de la réservation pourraient également être mis en cache dans Redis pour un accès rapide ultérieur, par exemple, pour afficher l'historique des réservations de l'utilisateur.
4. **Traitement en arrière-plan** :
   - Si des traitements supplémentaires sont nécessaires (comme l'envoi d'un email de confirmation), ces tâches pourraient être placées dans une file d'attente gérée par Redis.
   - Un service de traitement en arrière-plan consommerait cette file d'attente pour effectuer les actions nécessaires.

Conclusion

Redis et la gestion ACRISS jouent des rôles complémentaires dans Notre architecture de microservices :

- **Redis** améliore la performance, la scalabilité, et l'expérience utilisateur en agissant comme un cache rapide, un magasin de sessions, et un broker de messages. Il aide à réduire la latence et la charge sur les systèmes de stockage principaux, tout en fournissant des outils pour gérer des tâches asynchrones et des communications événementielles entre services.
- **La gestion ACRISS** standardise et simplifie la gestion des informations sur les véhicules, améliorant l'interopérabilité entre différents systèmes et assurant une expérience utilisateur cohérente et intuitive. Cela est particulièrement important dans un contexte international où la standardisation permet de surmonter les barrières linguistiques et culturelles.

Ensemble, ces technologies permettent de construire une architecture microservices qui est à la fois performante, scalable, et facile à maintenir, tout en offrant une expérience utilisateur fluide et cohérente.

1.6. Architectures de transition

Étant donné la complexité de migration des systèmes existants vers la nouvelle architecture microservices, nous proposons une approche progressive :

1.6.1. Phase 1: Core Services (Mois 1-3)

- Déploiement des services Auth et User
- Migration progressive des profils utilisateurs
- Mise en place de l'API Gateway
- Frontend Angular pour inscription/connexion

1.6.2. Phase 2: Booking Engine (Mois 4-6)

- Déploiement des services Vehicle et Reservation
- Intégration ACRISS externe avec cache
- Migration du catalogue véhicules
- Fonctionnalités de recherche et réservation

1.6.3. Phase 3: Support & Notifications (Mois 7-8)

- Déploiement des services Support et Notification
- Chat en temps réel et visioconférence
- Système de tickets et base de connaissances

1.6.4. Phase 4: Multi-Regions (Mois 9-12)

- Déploiement US et UK
- Réplication cross-region
- Conformité réglementaire complète
- Monitoring et optimisations

1.6.5. Phase 5: Migration Legacy (Mois 12+)

- Migration complète des données historiques
- Arrêt progressif des anciennes applications
- Formation des équipes d'agence
- Optimisations et nouvelles fonctionnalités

Cette approche permet de minimiser les risques, valider l'architecture à chaque étape, et maintenir la continuité de service pendant la transition.