République Tunisienne
Ministère de l'Enseignement Supérieur
Université de Tunis El Manar
Institut Supérieur d'Informatique

# Rapport de Projet

## Conception de Systèmes Complexes

## Système de Gestion et d'Optimisation de Livraison

***Réalisé par***

Chiheb ELLEFI
Ibtihel Khmili
Rayen Fehri
Chadha Ammar

**Année Universitaire : 2025–2026**

**Cours :** Conception de Systèmes Complexes
**Niveau :** 2ème Année Ingénieur

# Table des matières

**1 Introduction générale** **3**
1.1 Contexte de la logistique moderne . . . 3
1.2 Objectif du projet . . . 3
1.3 Problématique . . . 3

**2 Description du projet** **3**
2.1 Contexte . . . 3
2.2 Objectifs . . . 3
2.3 Portée du projet . . . 4
2.3.1 Version MVP (Minimum Viable Product) . . . 4
2.3.2 Améliorations futures . . . 4

**3 Conception** **4**
3.1 Spécification des besoins . . . 4
3.1.1 Identification des acteurs . . . 4
3.1.2 Besoins fonctionnels . . . 4
3.1.3 Besoins non fonctionnels . . . 5
3.1.4 Diagrammes de cas d'utilisation . . . 5

**4 Architecture du système** **7**
4.1 Architecture de haut niveau . . . 7
4.2 Composants principaux . . . 7
4.2.1 Services AWS utilisés . . . 8
4.3 Décomposition des services . . . 9
4.3.1 Services métier principaux . . . 9
4.3.2 Moteur d'optimisation . . . 10
4.3.3 Services d'infrastructure . . . 10
4.4 Couche de données . . . 10
4.5 Communication inter-services . . . 10
4.6 Stack d'observabilité . . . 11
4.7 Services externes . . . 11
4.8 Applications clientes . . . 11
4.8.1 Portail web Angular . . . 11
4.8.2 Application mobile chauffeur . . . 11

**5 Réalisation** **13**
5.1 Technologies utilisées . . . 13
5.1.1 Backend – Services Spring Boot . . . 13
5.1.2 Moteur d'optimisation . . . 13
5.1.3 Bases de données . . . 14
5.1.4 Infrastructure . . . 14
5.1.5 Messaging et Communication inter-services . . . 14
5.1.6 Monitoring & Logging . . . 15
5.2 Implémentation des Services d'Infrastructure . . . 15
5.2.1 Registre de Services (Netflix Eureka) . . . 15
5.2.2 Serveur d'Autorisation (Spring Authorization Server) . . . 16
5.2.3 Serveur de Configuration (Spring Cloud Config) . . . 19
5.2.4 Passerelle API (Spring Cloud Gateway) . . . 19

**6 Conclusion** **21**

**A Glossaire** **22**

# 1 Introduction générale

## 1.1 Contexte de la logistique moderne

L'industrie de la logistique et de la livraison connaît une croissance exponentielle, alimentée par l'essor du commerce électronique et les attentes croissantes des consommateurs en matière de rapidité et d'efficacité. Les entreprises de livraison font face à des défis majeurs : optimisation des itinéraires, affectation intelligente des chauffeurs, gestion des contraintes de temps et de capacité, tout en minimisant les coûts opérationnels.

Dans ce contexte, l'automatisation et l'intelligence artificielle jouent un rôle crucial pour améliorer la performance opérationnelle. Les algorithmes d'optimisation permettent de réduire les distances parcourues, d'équilibrer la charge de travail entre les chauffeurs, et de respecter les fenêtres de temps de livraison.

## 1.2 Objectif du projet

Ce projet vise à concevoir et développer un système complet de gestion et d'optimisation de livraison basé sur une architecture microservices moderne et déployé sur une infrastructure cloud scalable. Le système permettra aux entreprises de livraison d'automatiser l'affectation des chauffeurs, de calculer les itinéraires optimaux, et de suivre les livraisons en temps réel.

## 1.3 Problématique

Comment développer un système automatisé capable d'optimiser les livraisons en temps réel, tout en garantissant la scalabilité, la fiabilité et une expérience utilisateur fluide pour les gestionnaires, chauffeurs et clients, en utilisant une infrastructure cloud moderne ?

# 2 Description du projet

## 2.1 Contexte

Le système de gestion de livraison vise à résoudre les problèmes suivants :

— Affectation manuelle des livraisons aux chauffeurs, source d'inefficacité
— Routes non optimisées entraînant des coûts de carburant élevés
— Manque de visibilité en temps réel sur la localisation des chauffeurs
— Déséquilibre de la charge de travail entre les chauffeurs
— Difficulté à respecter les fenêtres de temps de livraison
— Infrastructure on-premise coûteuse et difficile à maintenir

## 2.2 Objectifs

Le système proposé vise à atteindre les objectifs suivants :

— Automatiser l'affectation intelligente des livraisons aux chauffeurs
— Calculer les itinéraires optimaux en minimisant la distance et le temps
— Assurer le suivi en temps réel de la position des chauffeurs
— Équilibrer la charge de travail entre les chauffeurs disponibles
— Respecter les contraintes de capacité des véhicules et les fenêtres horaires
— Fournir des analyses et rapports de performance
— Déployer sur une infrastructure cloud évolutive et résiliente

## 2.3 Portée du projet

### 2.3.1 Version MVP (Minimum Viable Product)

— Authentification et gestion des utilisateurs
— Gestion des véhicules et chauffeurs
— Affectation intelligente des livraisons
— Calcul d'itinéraires optimaux
— Applications web pour clients et gestionnaires
— Application mobile pour chauffeurs
— Déploiement sur AWS avec Docker Compose

### 2.3.2 Améliorations futures

— Suivi en temps réel avec WebSocket
— Tableau de bord analytique avancé avec Machine Learning
— Communication asynchrone complète avec RabbitMQ
— Déploiement Kubernetes (EKS) avec auto-scaling
— Système de notifications multi-canaux (SMS, Email, Push)
— Analyse prédictive des délais de livraison
— Intégration avec systèmes ERP externes

# 3 Conception

## 3.1 Spécification des besoins

### 3.1.1 Identification des acteurs

— **Gestionnaire (Manager)** : Gère les chauffeurs et véhicules, crée et assigne les livraisons, génère les plannings, consulte les tableaux de bord via l'application web Angular.
— **Chauffeur** : Consulte les livraisons assignées, accède aux itinéraires, met à jour le statut, partage sa position en temps réel via l'application mobile (React Native/Flutter/Native).
— **Client (Vendeur)** : Crée des demandes de livraison, consulte l'historique, suit le statut des commandes, gère ses préférences via le portail web Angular.
— **Administrateur** : Gère tous les utilisateurs, configure les paramètres système, accède aux métriques d'infrastructure, gère les clés API et les services cloud.

### 3.1.2 Besoins fonctionnels

— **Gestion des livraisons** : CRUD des livraisons, adresses, fenêtres horaires, suivi statut, historique.
— **Optimisation et affectation** : Affectation automatique basée sur algorithmes, calcul d'itinéraires optimaux, gestion des contraintes de capacité, équilibrage de charge, re-optimisation dynamique.
— **Suivi et tracking** : GPS temps réel, calcul ETA dynamique, géofencing, historique des déplacements.
— **Analytics et reporting** : Tableaux de bord interactifs, KPI en temps réel, rapports de performance, analyse d'efficacité opérationnelle.
— **Applications utilisateur** :

— Portail web client (Angular)
— Application web gestionnaire (Angular)
— Application mobile chauffeur (React Native/Flutter/Kotlin-Swift)

### 3.1.3 Besoins non fonctionnels

— **Scalabilité** : 300+ utilisateurs simultanés, 10 000+ livraisons/jour, auto-scaling horizontal via Kubernetes HPA.
— **Performance** : Calcul itinéraire $< 2$s, API REST $< 500$ms, tracking temps réel $< 1$s.
— **Fiabilité** : 99.9% disponibilité (SLA), gestion robuste des erreurs, retry automatique, tolérance aux pannes.
— **Sécurité** : TLS 1.3, JWT avec rotation, RBAC granulaire, chiffrement des données au repos et en transit, audit trail complet.
— **Ergonomie** : Interface intuitive et responsive, support multi-plateforme (web, mobile iOS/Android).
— **Cloud-native** : Déploiement sur AWS, utilisation de services managés, infrastructure as code.

### 3.1.4 Diagrammes de cas d'utilisation

La figure suivante présente une vue globale des interactions entre les acteurs et le système.

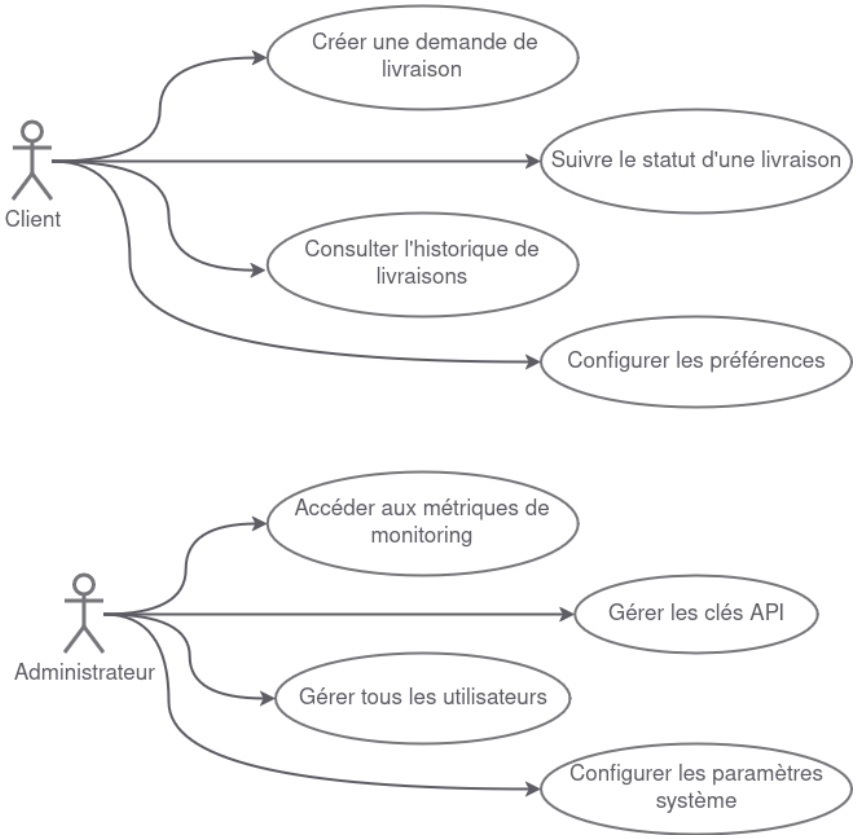



FIGURE 1 – Diagramme de cas d'utilisation du système

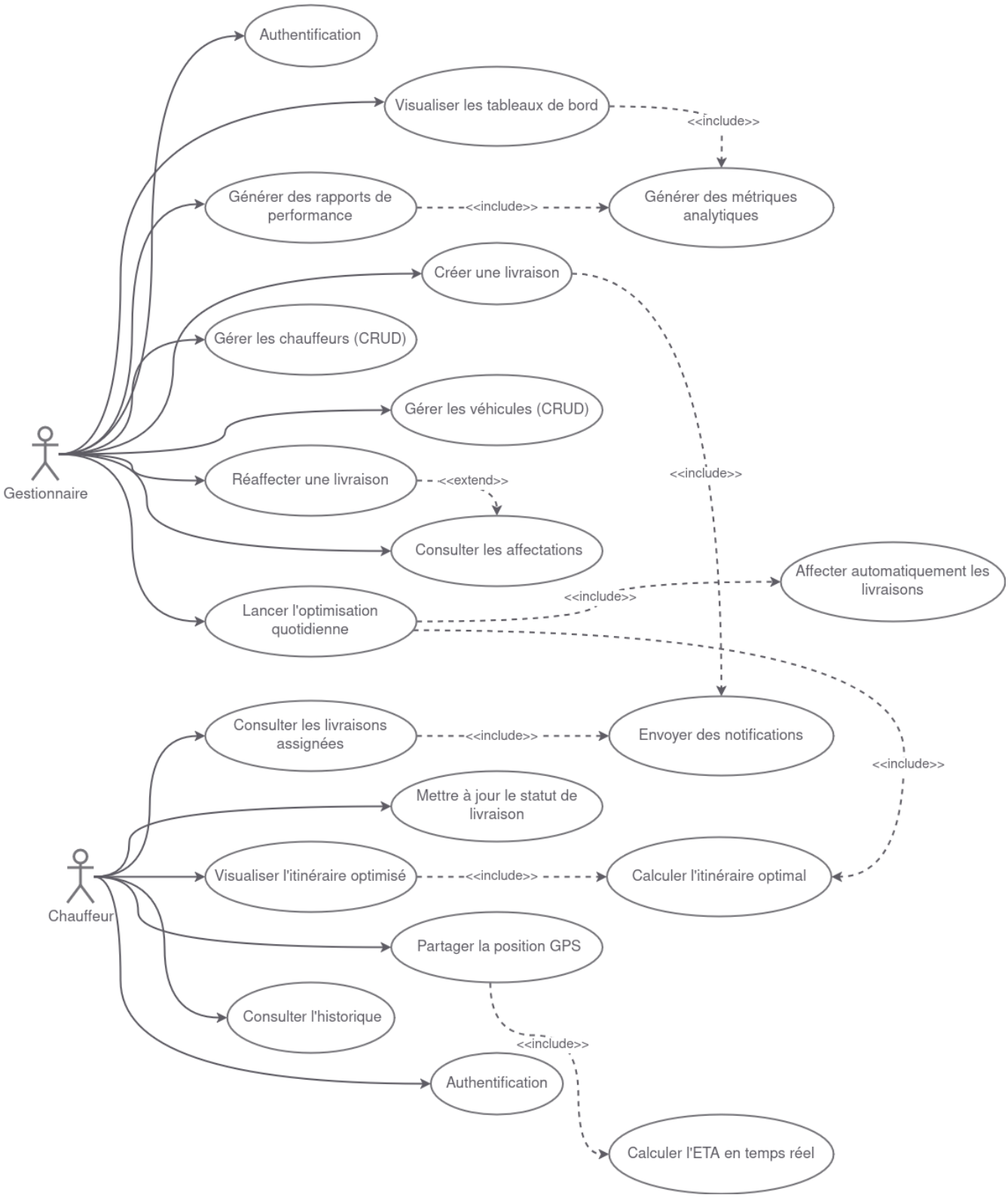



Figure 2 – Diagramme de cas d'utilisation du système

# 4 Architecture du système

## 4.1 Architecture de haut niveau

Le système adopte une architecture microservices cloud-native moderne pour assurer la scalabilité, la maintenabilité et la résilience. Chaque service est indépendant, conteneurisé avec Docker, déployable séparément sur Kubernetes (Amazon EKS), et communique via des API REST/gRPC et des événements asynchrones.

L'infrastructure est entièrement hébergée sur Amazon Web Services (AWS) pour garantir une haute disponibilité, une scalabilité élastique et une réduction des coûts opérationnels.

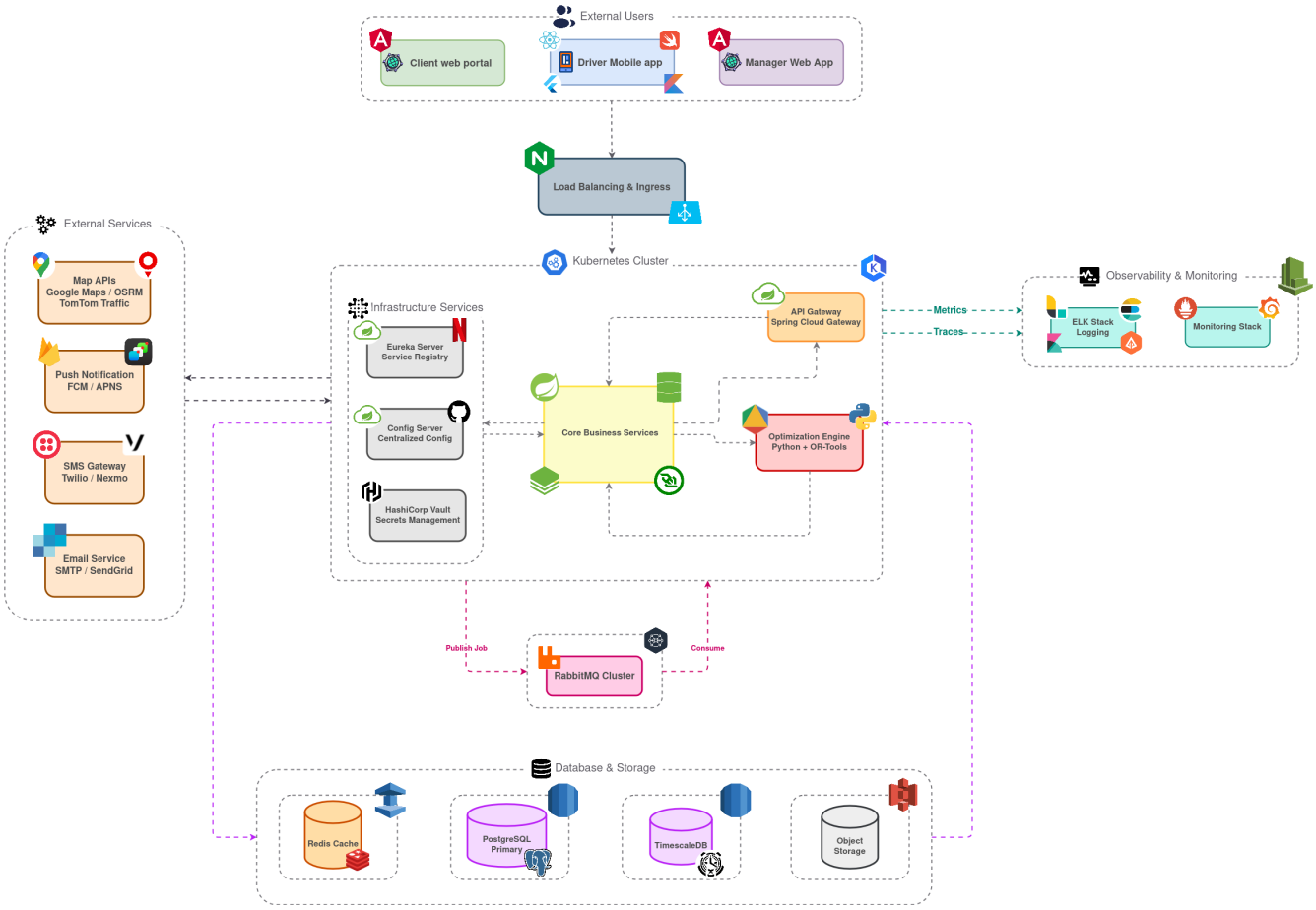



FIGURE 3 – Architecture globale du système

## 4.2 Composants principaux

- **API Gateway** : Point d'entrée unique (Spring Cloud Gateway)
- **Services métier** : Microservices applicatifs (Spring Boot)
- **Moteur d'optimisation** : Service Python avec OR-Tools de Google
- **Couche de données** : Amazon RDS PostgreSQL, TimescaleDB, Amazon ElastiCache Redis
- **Message Broker** : Amazon MQ (RabbitMQ managé)
- **Services d'infrastructure** : Authentification, découverte de services, configuration centralisée
- **Stack d'observabilité** : ELK Stack, Prometheus, Grafana, Zipkin, Amazon CloudWatch

— **Applications clientes** :
  — Portail web Angular (Client & Manager)
  — Application mobile (React Native, Flutter, ou Native Kotlin/Swift)

### 4.2.1 Services AWS utilisés

| Service AWS | Icône | Utilisation | Justification |
|---|---|---|---|
| Amazon EKS |  | Orchestration Kubernetes des microservices | Scalabilité automatique, haute disponibilité, gestion simplifiée |
| Amazon RDS PostgreSQL |  | Base de données relationnelle principale | Service managé, backups automatiques, multi-AZ |
| Amazon ElastiCache Redis |  | Cache mémoire distribué | Haute performance, réplication automatique |
| Amazon MQ (RabbitMQ) |  | Message broker pour communication asynchrone | Service managé, haute disponibilité |
| Amazon S3 |  | Stockage objets (logs, rapports, exports) | Durabilité 99.999999999%, versioning, lifecycle policies |
| Amazon CloudWatch |  | Monitoring, logs, métriques | Intégration native AWS, alertes automatiques |

## 4.3 Décomposition des services

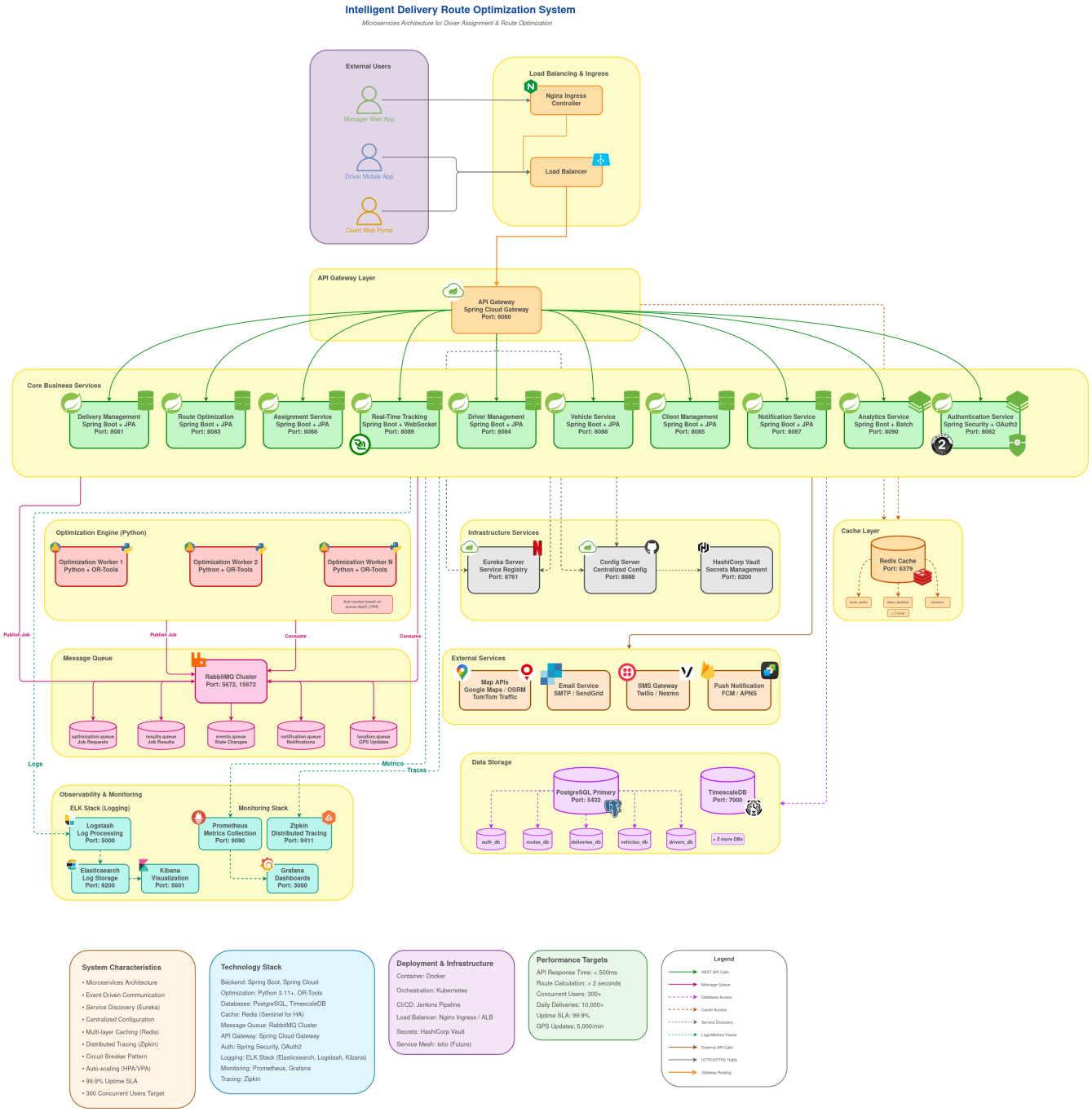



FIGURE 4 – Architecture logique simplifiée

### 4.3.1 Services métier principaux

— **Delivery Management Service** : Gestion du cycle de vie complet des livraisons (création, modification, suppression, consultation)

— **Route Optimization Service** : Orchestration de l'optimisation des itinéraires, interface avec le moteur Python OR-Tools

— **Assignment Service** : Gestion des affectations chauffeur-livraison, équilibrage de charge, gestion des contraintes

- **Real-Time Tracking Service** : Suivi GPS en temps réel, calcul ETA dynamique, géofencing
- **Driver Management Service** : Gestion des chauffeurs (CRUD, disponibilité, shifts, performance)
- **Vehicle Service** : Gestion de la flotte de véhicules (CRUD, maintenance, capacités, statut)
- **Client Management Service** : Gestion des clients vendeurs (CRUD, préférences, historique)
- **Notification Service** : Notifications multi-canaux
- **Analytics Service** : Génération d'insights, rapports de performance, tableaux de bord

#### 4.3.2 Moteur d'optimisation

**Optimization Worker Service (Python + OR-Tools)**

- Exécution des algorithmes d'optimisation de routage (VRP - Vehicle Routing Problem)
- Gestion des contraintes complexes :
  - Fenêtres de temps de livraison
  - Capacités des véhicules (poids, volume)
  - Compétences requises des chauffeurs
  - Temps de service et de déplacement
  - Priorités des livraisons
- Technologies : Python 3.11+, OR-Tools, RabbitMQ

#### 4.3.3 Services d'infrastructure

- **Authentication Service** : Gestion JWT, OAuth2
- **API Gateway** : Spring Cloud Gateway avec rate limiting et circuit breaker
- **Service Registry** : Eureka Server pour la découverte de services
- **Configuration Service** : Spring Cloud Config avec backend Git

### 4.4 Couche de données

- **PostgreSQL** :
  - Base relationnelle principale
- **TimescaleDB** :
  - Séries temporelles (positions GPS des chauffeurs)
- **Redis** :
  - Cache mémoire distribué
  - Session storage (tokens JWT, sessions utilisateurs)
  - Cache des itinéraires calculés

### 4.5 Communication inter-services

- **Amazon MQ (RabbitMQ)** :
  - Communication asynchrone entre microservices
- **REST API** :
  - Communication synchrone (environnement développement)
  - Documentation automatique avec Swagger/OpenAPI
- **gRPC** :
  - Communication synchrone haute performance (production)
  - Protocol Buffers pour sérialisation efficace

## 4.6 Stack d'observabilité

- **ELK Stack (sur EKS)** :
  - Elasticsearch : Indexation et recherche de logs
  - Logstash : Collecte et transformation de logs
  - Kibana : Visualisation et analyse
- **Prometheus + Grafana (sur EKS)** :
  - Prometheus : Collecte de métriques temps réel
  - Grafana : Dashboards interactifs
  - Alertmanager : Gestion des alertes
- **Zipkin (sur EKS)** :
  - Distributed tracing entre microservices
  - Analyse des latences et goulots d'étranglement
  - Visualisation des flux de requêtes

## 4.7 Services externes

- **APIs de cartographie** :
  - TomTom API : Calcul d'itinéraires, matrices de distance
  - Google Maps API : Géocodage, reverse geocoding
  - OSRM (Open Source Routing Machine) : Alternative open-source
- **Services de notifications** :
  - Amazon SES : Envoi d'emails transactionnels
  - Amazon SNS : SMS et notifications push
  - Twilio (optionnel) : SMS et appels vocaux

## 4.8 Applications clientes

### 4.8.1 Portail web Angular

**Client Web Portal & Manager Web Application**

- **Framework** : Angular
- **Fonctionnalités** :
  - Dashboard interactif avec KPI temps réel
  - Gestion des livraisons (CRUD, filtres, recherche)
  - Visualisation des itinéraires sur carte
  - Suivi temps réel des chauffeurs
  - Génération de rapports et exports
  - Gestion des utilisateurs et permissions (Manager uniquement)

### 4.8.2 Application mobile chauffeur

**Options technologiques** :

**Option 1 : React Native**

- **Avantages** : Code partagé iOS/Android, large communauté, performance native

**Option 2 : Flutter**

- **Avantages** : Performance excellente, UI cohérente, hot reload rapide

**Option 3 : Native (Kotlin + Swift)**

- **Android** : Kotlin, Jetpack Compose, Coroutines, Room
- **iOS** : Swift, SwiftUI, Combine, CoreData
- **Avantages** : Performance maximale, accès complet aux APIs natives
- **Inconvénient** : Développement et maintenance séparés

**Fonctionnalités communes** :

- Authentification biométrique (Face ID/Touch ID)
- Liste des livraisons assignées du jour
- Navigation GPS turn-by-turn vers destinations
- Mise à jour statut livraison (en transit, livrée, échec)
- Scan de codes-barres/QR codes
- Signature électronique du client
- Photo de preuve de livraison
- Partage automatique de position GPS en arrière-plan
- Notifications push en temps réel
- Mode offline avec synchronisation

# 5 Réalisation

## 5.1 Technologies utilisées

### 5.1.1 Backend – Services Spring Boot

| Technologie | Logo | Rôle |
|---|---|---|
| Spring Boot |  | Framework principal pour les microservices |
| Spring Cloud Gateway |  | API Gateway avec routage dynamique |
| Spring Security + OAuth2 |  | Authentification et autorisation |
| Spring Data JPA |  | Accès aux données PostgreSQL |
| Spring WebSocket |  | Communication temps réel |
| Spring Cloud Config |  | Configuration centralisée des microservices |
| Spring Cloud Netflix Eureka |  | Service de découverte des microservices |

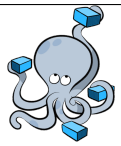
### 5.1.2 Moteur d'optimisation

| Technologie | Logo | Rôle |
|---|---|---|
| Python |  | Langage principal du moteur d'optimisation |
| OR-Tools |  | Bibliothèque d'optimisation de Google |

### 5.1.3 Bases de données

| Technologie | Logo | Rôle |
|---|---|---|
| PostgreSQL |  | Base de données relationnelle principale |
| TimescaleDB |  | Extension pour séries temporelles |
| Redis |  | Cache et session store |

### 5.1.4 Infrastructure

| Technologie | Logo | Rôle |
|---|---|---|
| Docker |  | Conteneurisation des services |
| Docker Compose |  | Orchestration locale (développement) |
| Kubernetes |  | Orchestration en production |

### 5.1.5 Messaging et Communication inter-services

| Technologie | Logo | Rôle |
|---|---|---|
| RabbitMQ |  | Message broker pour communication asynchrone |
| REST (Spring Web) |   | Communication synchrone via API HTTP |
| gRPC |   | Communication synchrone haute performance entre microservices |

### 5.1.6 Monitoring & Logging

| Technologie | Logo | Rôle |
|---|---|---|
| ELK Stack |  | Centralisation et visualisation des logs |
| Prometheus |  | Collecte de métriques |
| Grafana |  | Dashboards et visualisation |
| Zipkin |  | Distributed tracing |

## 5.2 Implémentation des Services d'Infrastructure

Cette section détaille l'implémentation effective des quatre services d'infrastructure réalisés, avec les configurations clés et les captures d'écran de validation.

### 5.2.1 Registre de Services (Netflix Eureka)

**Vue d'ensemble** Le Registre de Services est le premier service à démarrer dans l'écosystème. Il sert d'annuaire téléphonique pour tous les microservices : chaque service s'y enregistre au démarrage, y envoie des battements de cœur périodiques, et les autres services l'interrogent pour découvrir dynamiquement les adresses réseau sans les coder en dur.

**Configuration clé** Le service écoute sur le **port 8761**, convention standard Eureka reconnue automatiquement par tous les clients Spring. La communication est sécurisée par TLS avec un keystore PKCS12 partagé. L'auto-enregistrement est explicitement désactivé (`register-with-eureka: false`) car le registre est lui-même le serveur — il n'a pas de pair auprès duquel s'enregistrer.

Listing 1 – Registre de Services – paramètres essentiels

```
eureka:
  server:
    enable-self-preservation: true      # Protection contre les partitions
       reseau
    renewal-percent-threshold: 0.85     # Seuil : 85% des battements
       attendus
    eviction-interval-timer-in-ms: 60000
  instance:
    lease-renewal-interval-in-seconds: 30   # Battement toutes les 30s
    lease-expiration-duration-in-seconds: 90 # Expulsion apres 90s de
       silence
  client:
    register-with-eureka: false  # Le registre ne s'enregistre pas lui-meme
    fetch-registry: false
```

Le **mode de préservation automatique** est un mécanisme de protection essentiel : si moins de 85% des battements de cœur attendus sont reçus dans une fenêtre de temps, Eureka

suspend les évictions de services plutôt que de les supprimer à tort — un comportement critique lors d'une partition réseau temporaire.

**Validation** La figure 5 montre le tableau de bord Eureka en fonctionnement. Deux services sont enregistrés et en état `UP` : l'`AUTHORIZATIONSERVICE` sur le port 9000 et le `GATEWAY` sur le port 8080. Le registre indique 6 renouvellements de baux dans la dernière minute, confirmant le bon fonctionnement des battements de cœur.

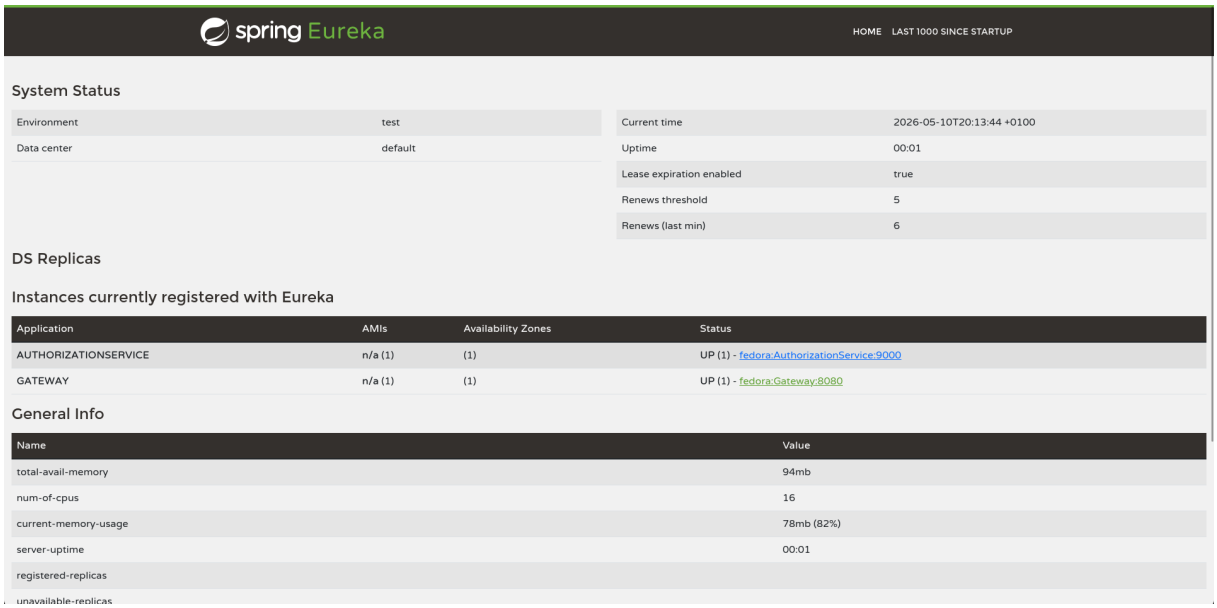



FIGURE 5 – Tableau de bord Eureka – AUTHORIZATIONSERVICE et GATEWAY enregistrés et UP

### 5.2.2 Serveur d'Autorisation (Spring Authorization Server)

**Vue d'ensemble** Le serveur d'autorisation est implémenté sur Spring Authorization Server 1.x avec Spring Boot 3.x et Spring Security 6.x. Il implémente OAuth 2.1 avec OpenID Connect et gère l'ensemble du cycle de vie des jetons JWT : émission, validation, révocation et détection de réutilisation des jetons de rafraîchissement.

**Configuration clé** Le service s'exécute sur le **port 9000** avec TLS activé. Il utilise PostgreSQL comme base de données relationnelle pour les clients OAuth et les autorisations, et Redis comme store de liste noire pour les JTI révoqués.

Listing 2 – Serveur d'Autorisation – paramètres essentiels

```
token:
  format: self-contained          # Jetons JWT autonomes (pas opaques)
  access-token-ttl: 300           # 5 minutes
  refresh-token-ttl: 2592000      # 30 jours
  reuse-refresh-tokens: true
  id-token-signature-algorithm: RS256

auth:
  allowed-grant-types: authorization_code, refresh_token,
      client_credentials
```

Les jetons d'accès ont une durée de vie courte (5 minutes) pour limiter l'exposition en cas de compromission, tandis que les jetons de rafraîchissement (30 jours) permettent le renouvellement silencieux côté client. La signature RS256 garantit que seul le serveur d'autorisation, détenteur de la clé privée, peut émettre des jetons valides.

**Flux OAuth 2.1 implémentés** Trois flux sont supportés :

— **Authorization Code + PKCE** : flux principal pour les applications utilisateur. Le client génère un `code_verifier` aléatoire et calcule le `code_challenge = BASE64URL(SHA256(code_ve` Spring vérifie l'équation avant d'émettre les jetons, rendant le code d'autorisation inutilisable en cas d'interception.

— **Refresh Token** : renouvellement silencieux de l'accès sans réauthentification. Une détection de réutilisation par script Lua atomique révoque toute la famille de jetons en cas de rejeu.

— **Client Credentials** : authentification machine-à-machine (M2M) sans utilisateur — un jeton d'accès est émis directement au service appelant.

**Structure des jetons JWT**

| Claim | Description |
|---|---|
| `iss` | Émetteur — URL du serveur d'autorisation |
| `sub` | Sujet — UUID de l'utilisateur |
| `aud` | Audience — liste des APIs autorisées |
| `exp` / `iat` | Expiration / émission (timestamp Unix) |
| `jti` | JWT ID — UUID v4, clé de liste noire Redis pour la révocation |
| `scope` | Périmètres accordés (`openid profile email`) |
| `roles` | Rôles de l'utilisateur (`ROLE_USER`, `ROLE_ADMIN`) |

**Gestion de la liste noire Redis** Chaque jeton révoqué est inscrit dans Redis avec une clé `revoked:{type}:{jti}` et un TTL égal à la durée de vie restante du jeton — garantissant un nettoyage automatique sans tâche planifiée. Les familles de jetons de rafraîchissement sont trackées via `token_family:{userId}` pour permettre la révocation groupée en cas de compromission de compte.

**Validation – Émission des jetons** La figure 6 présente l'interface de gestion des jetons après une authentification réussie. On observe les trois jetons émis : l'`access_token` (Bearer), le `refresh_token` et l'`id_token` OpenID Connect, avec le périmètre `openid profile`.

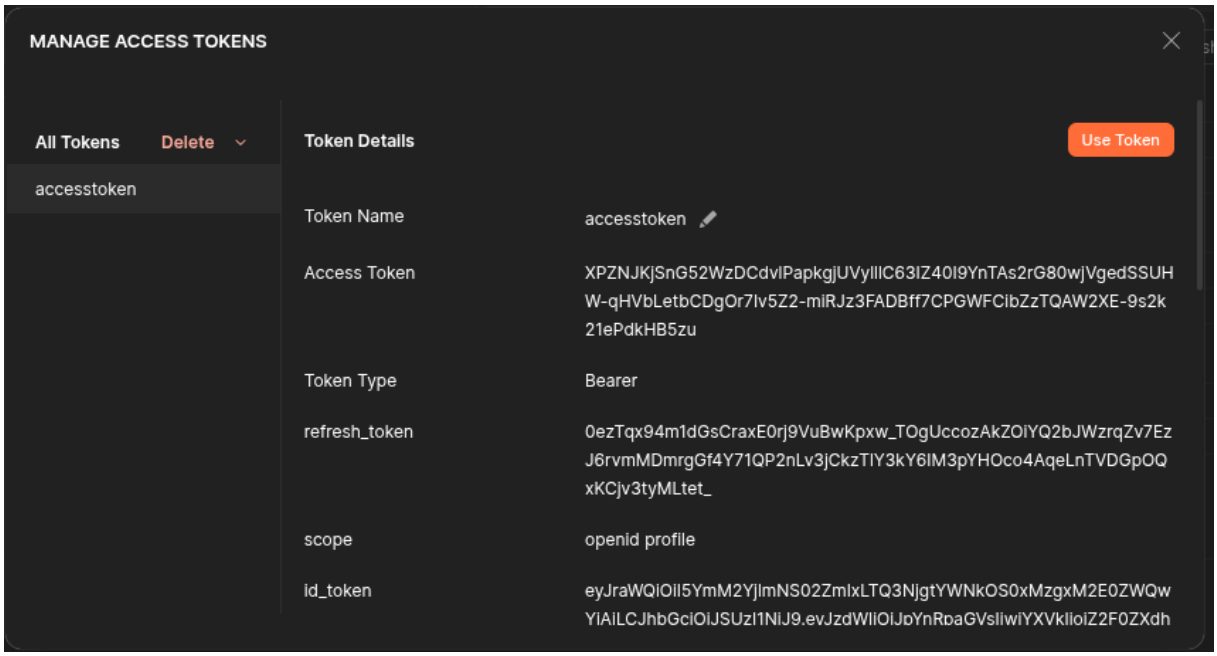



FIGURE 6 – Jetons émis par le Serveur d'Autorisation – access token, refresh token et id token (scope : openid profile)

**Validation – Flux de rafraîchissement** La figure 7 illustre le flux de rafraîchissement en action. La requête `POST /oauth2/token` avec `grant_type=refresh_token` retourne un HTTP 200 OK en **118 ms** avec un nouveau triplet de jetons et une durée de validité de 299 secondes, confirmant la rotation correcte des jetons.

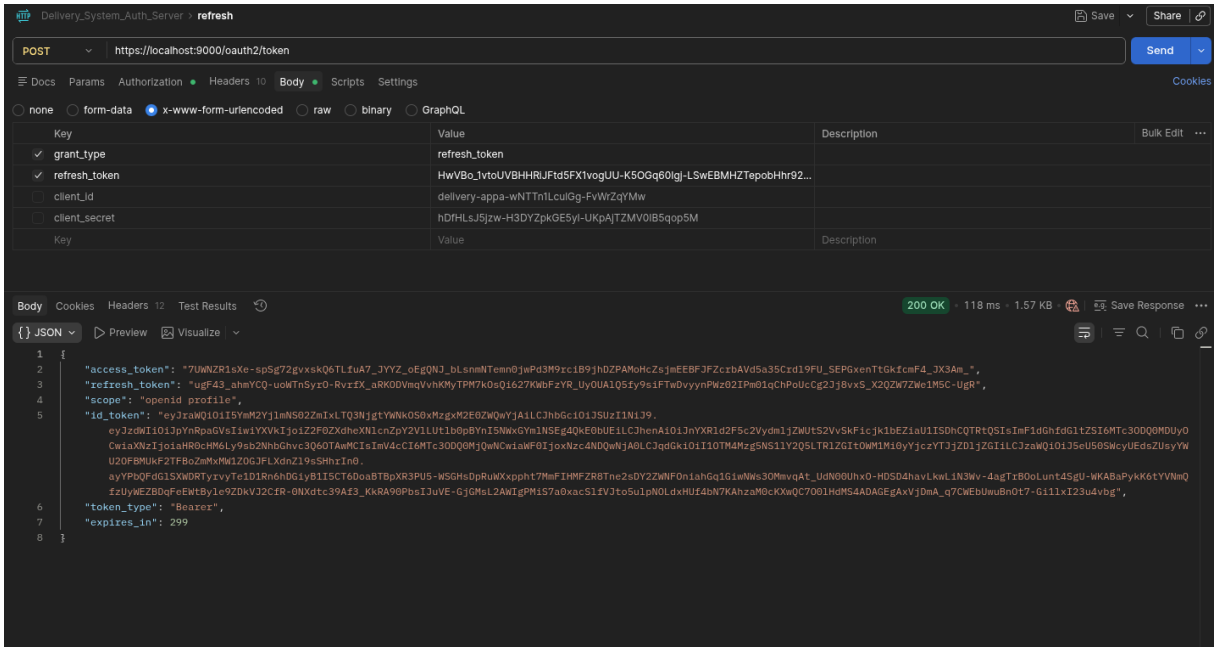



FIGURE 7 – Flux de rafraîchissement – POST /oauth2/token (200 OK en 118 ms, nouveaux access token et refresh token)

### 5.2.3 Serveur de Configuration (Spring Cloud Config)

**Vue d'ensemble** Le Serveur de Configuration centralise toute la configuration de l'écosystème. Sans ce composant, chaque service porterait ses propres fichiers de configuration déployés localement, rendant la gestion des environnements et la rotation des secrets extrêmement fragile.

**Configuration clé** Deux backends sont utilisés en ordre de priorité décroissante : **HashiCorp Vault** (ordre 1) pour les secrets, et un **dépôt Git privé** (ordre 2) pour la configuration applicative. En cas de clé présente dans les deux sources, Vault l'emporte toujours — les secrets ne sont jamais écrasés par la configuration Git.

Listing 3 – Serveur de Configuration – backends et routage

```
spring:
  cloud:
    config:
      server:
        vault:
          order: 1              # Priorite maximale -- secrets sensibles
          kv-version: 2         # Versioning des secrets Vault
        git:
          uri: git@github.com:org/delivery-system-config.git
          force-pull: true      # Toujours servir la version GitHub
              autoritaire
          refresh-rate: 60      # Re-pull toutes les 60 secondes
          order: 2
          repos:
            infrastructure:
              search-paths: [ "INFRA/{application}" ]
              pattern: [ "gateway*", "registry*" ]
            business:
              search-paths: [ "BL/{application}" ]
              pattern: [ "delivery*", "driver*", "tracking*" ]
      fail-fast: true           # Echec immediat si Config Server
          inaccessible
      retry:
        max-attempts: 6
        multiplier: 1.1         # Backoff exponentiel
```

La structure du dépôt Git sépare proprement les services d'infrastructure (`INFRA/`) des services métier (`BL/`), chacun avec des fichiers de profil distincts (`service.yml` et `service-prod.yml`). Le mécanisme **fail-fast** garantit qu'aucun service ne démarre avec une configuration vide : après 6 tentatives avec backoff exponentiel, il échoue immédiatement plutôt que de fonctionner dans un état incohérent. Les valeurs `{cipher}` dans Git sont déchiffrées automatiquement avant livraison aux microservices.

### 5.2.4 Passerelle API (Spring Cloud Gateway)

**Vue d'ensemble** La Passerelle API est le service d'infrastructure le plus complexe. Implémentée avec Spring Cloud Gateway sur le modèle réactif (non bloquant) de Project Reactor et Netty, elle constitue l'unique point d'entrée pour toutes les requêtes clients. Son moteur réactif permet à un seul thread de gérer des milliers de connexions simultanées sans blocage d'I/O.

**Configuration clé** La découverte automatique des routes via Eureka (`discovery.locator.enabled: true`) évite toute configuration manuelle des routes. La validation JWT est effectuée localement grâce aux clés publiques JWKS mises en cache, sans aller-retour vers le serveur d'autorisation.

Listing 4 – Passerelle API – resilience et securite

```
resilience4j:
  circuitbreaker:
    configs:
      default:
        sliding-window-size: 100
        failure-rate-threshold: 50       # Ouvre des 50% d'echecs
        wait-duration-in-open-state:
          seconds: 60
      critical:
        failure-rate-threshold: 30       # Seuil reduit pour services
           critiques
        wait-duration-in-open-state:
          seconds: 30
  timelimiter:
    configs:
      fast:
        timeout-duration:
          seconds: 2                     # Tracking temps reel
      batch:
        timeout-duration:
          seconds: 30                    # Operations longues (exports)

spring:
  security:
    oauth2:
      resourceserver:
        jwt:
          jwk-set-uri: ${JWKS_URI}      # Cles publiques mises en cache
             localement
```

Trois profils de tolérance aux pannes sont définis selon la criticité du service : `critical` (authentification, seuil 30%), `default` (services métier standard, seuil 50%) et `lenient` (analytics, seuil 70%). De même, les time limiters varient de 2 secondes pour le tracking temps réel à 30 secondes pour les exports groupés. Le rate limiting distribué est assuré par Redis avec l'algorithme Token Bucket, avec quatre profils allant de 1 req/s (endpoints sensibles) à 100 req/s (services internes).

**Pipeline de traitement d'une requête** Chaque requête entrante traverse les dix étapes suivantes dans l'ordre :

1. **Terminaison SSL** — Netty gère le handshake TLS avec le keystore PKCS12.
2. **Vérification JWT** — Le Bearer token est validé localement via les clés publiques JWKS récupérées et mises en cache depuis `/oauth2/jwks`. Tout jeton invalide, expiré ou de mauvais émetteur retourne un 401.
3. **Rate Limiting** — Le filtre Redis Token Bucket est appliqué. Chaque IP dispose de son propre seau de jetons ; un seau vide retourne HTTP 429.
4. **Correspondance de route** — Routage automatique via la découverte Eureka.
5. **Équilibrage de charge** — Sélection d'une instance UP en zone prioritaire.
6. **Bulkhead** — Vérification de la limite de requêtes simultanées vers le service cible.
7. **Circuit Breaker** — Rejet immédiat si le circuit est en état OPEN.
8. **Time Limiter** — Annulation de la requête si le service ne répond pas dans le délai configuré.
9. **Transmission** — Forwarding vers le service backend avec en-têtes `X-Forwarded-*`.

10. **Réponse** — Retournée au client ; le circuit breaker enregistre le résultat dans sa fenêtre glissante.

# 6 Conclusion

Ce projet de système de gestion et d'optimisation de livraison représente une solution moderne et complète pour les entreprises de logistique. En adoptant une architecture microservices, nous garantissons la scalabilité, la maintenabilité et la résilience du système.

L'implémentation des quatre services d'infrastructure — Serveur d'Autorisation, Registre de Services, Serveur de Configuration et Passerelle API — constitue le fondement technique solide sur lequel reposent tous les microservices métier. Ces services assurent collectivement la sécurité (OAuth 2.1 / JWT / PKCE), la découverte dynamique des services, la centralisation de la configuration et la résilience du point d'entrée unique.

Les points forts du système incluent :

— Architecture microservices découplée et scalable

— Sécurité robuste : OAuth 2.1, PKCE, liste noire Redis, détection de réutilisation des jetons

— Configuration centralisée avec séparation secrets (Vault) / configuration (Git)

— Découverte de services dynamique avec protection contre les partitions réseau

— Passerelle résiliente : rate limiting distribué, circuit breaker, bulkhead, time limiter

— Optimisation intelligente avec OR-Tools de Google

— Suivi en temps réel des chauffeurs et livraisons

— Observabilité complète (logs, métriques, tracing)

— Pipeline CI/CD automatisé

— Haute disponibilité et tolérance aux pannes

La version MVP fournit les fonctionnalités essentielles pour démarrer les opérations, tandis que les améliorations futures permettront d'étendre les capacités du système avec des fonctionnalités avancées comme l'application mobile pour chauffeurs, les notifications multi-canaux, et les analyses prédictives.

Ce système est conçu pour évoluer avec les besoins de l'entreprise et peut supporter une croissance significative grâce à son architecture scalable et ses capacités d'auto-scaling.

# A Glossaire

**Microservices** Architecture où une application est composée de petits services indépendants, chacun exécutant un processus unique.

**OR-Tools** Bibliothèque open-source d'optimisation développée par Google pour résoudre des problèmes de routage, d'ordonnancement, etc.

**API Gateway** Point d'entrée unique qui gère les requêtes vers les microservices.

**Circuit Breaker** Patron de conception pour prévenir les cascades de défaillances dans les systèmes distribués. Trois états : CLOSED, OPEN, HALF-OPEN.

**Bulkhead** Patron de tolérance aux pannes qui limite le nombre de requêtes simultanées vers un service, empêchant un service de monopoliser toutes les ressources.

**Token Bucket** Algorithme de rate limiting : un seau contient des jetons rechargés à débit fixe ; chaque requête consomme un jeton. Seau vide = requête refusée (HTTP 429).

**PKCE** Proof Key for Code Exchange — extension du flux Authorization Code qui empêche le rejeu du code d'autorisation.

**JTI** JWT ID — identifiant unique d'un jeton JWT, utilisé comme clé dans le système de liste noire.

**JWKS** JSON Web Key Set — URL publiant les clés publiques du serveur d'autorisation pour vérifier les signatures JWT.

**gRPC** Framework RPC haute performance développé par Google utilisant HTTP/2 et Protocol Buffers.

**TimescaleDB** Extension de PostgreSQL optimisée pour les séries temporelles.

**RabbitMQ** Message broker open-source implémentant le protocole AMQP.

**JWT** JSON Web Token, standard pour les tokens d'authentification.

**RBAC** Role-Based Access Control, contrôle d'accès basé sur les rôles.

**ELK Stack** Elasticsearch, Logstash, Kibana - suite pour la gestion et visualisation de logs.

**Prometheus** Système de monitoring et d'alerte open-source.

**Grafana** Plateforme de visualisation et d'analyse de métriques.

**Zipkin** Système de traçage distribué open-source.

**Docker** Plateforme de conteneurisation.

**Kubernetes** Orchestrateur de conteneurs open-source.

**CI/CD** Continuous Integration/Continuous Deployment.

**Self-Preservation** Mode de préservation automatique d'Eureka : suspend les évictions lors d'une partition réseau pour éviter de retirer des services sains du registre.

**Vault** HashiCorp Vault — système dédié au stockage sécurisé des secrets (mots de passe, clés API, certificats) avec chiffrement, contrôle d'accès et audit complets.

**Fail-Secure** Comportement de sécurité : en cas de défaillance d'un composant critique (ex. Redis), la requête est bloquée plutôt que d'être autorisée.

**Fail-Safe** Comportement de disponibilité : en cas de défaillance d'un composant non critique, la requête est autorisée avec journalisation de l'avertissement.