Plateforme Intelligente de Services Urbains Interopérables

Smart City Platform

Rapport de Projet

Service Oriented Computing

REST SOAP gRPC GraphQL

Décembre 2025

Année universitaire 2025-2026

Ecole Nationale d'Ingénieurs de Carthage

Ce document a été rédigé en LaTeX
Version 1.0.0 — Décembre 2024

Table des matières

Table des figures

Liste des tableaux

Résumé Exécutif

Ce projet consiste en la conception et l'implémentation d'une **plateforme de services interopérables** pour une ville intelligente. La plateforme intègre quatre services distincts utilisant des protocoles différents (**REST**, **SOAP**, **GraphQL**, **gRPC**), orchestrés via un service central et exposés à travers une **API Gateway** unique.

Objectifs Atteints

- ✓ **Implémentation de 4 services** avec protocoles différents
- ✓ **Orchestration** de workflows métier complexes
- ✓ **Architecture microservices** complète basée sur Spring Boot
- ✓ **Déploiement** avec Docker et Docker Compose
- ✓ **Interface client web** fonctionnelle (React)
- ✓ **Documentation technique** complète
- ✓ **Base de données H2** avec données de test pré-chargées

Technologies Clés

Java 17 **Spring Boot 3.2** **Docker** **React 18**

Chapitre 1

Architecture Globale

Vue d'Ensemble

Notre architecture suit le pattern **microservices** avec les composants suivants, permettant une **séparation claire des responsabilités** et une **scalabilité horizontale**.

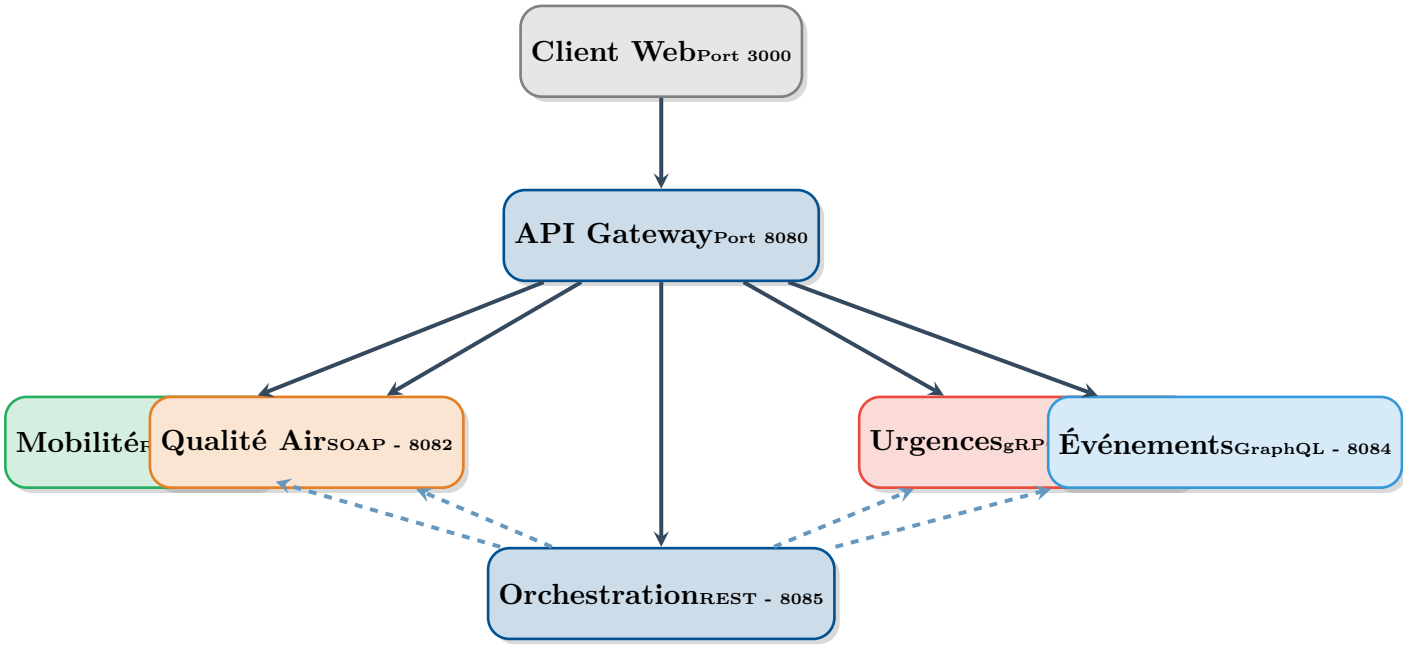

FIGURE 1.1 – Architecture globale de la plateforme Smart City

Choix Technologiques

Table 1.1 – Technologies et justifications

Composant	Technologie	Version	Justification
Runtime Backend	Java	17	LTS, performance, écosystème mature
Framework	Spring Boot	3.2.0	Productivité, configuration automatique
Services REST	Spring Web	3.2.0	Standard Spring, annotations simples
SOAP	Spring WS	4.0.x	Support WSDL complet, JAX-WS
gRPC	grpc-starter	3.1.0	Intégration Spring, haute performance
GraphQL	graphql-starter	15.0.0	SDL, résolveurs automatiques
Gateway	Spring Cloud	4.1.0	Réactif, filtres personnalisables
Base de données	H2 Database	2.2.x	Tests rapides, pas de setup externe
Client Web	React	18.2.0	UI moderne et réactive
Conteneurs	Docker	20.10+	Isolation et portabilité

Réseau Docker

Information

Tous les services communiquent via un réseau Docker bridge nommé `smart-city-network`, permettant la communication inter-conteneurs par nom de service.

Chapitre 2

Services Implémentés

Service Mobilité (REST)

accentgreen

Service Mobilité — REST
Protocole : REST (HTTP/JSON) **Port :** 8081
Context Path : `/mobility` **Framework :** Spring Boot + Spring Web

Fonctionnalités principales :
- Gestion des lignes de transport (Bus, Métro, Train)
- Consultation des horaires en temps réel
- État du trafic et informations stations
- Opérations CRUD complètes

API Endpoints

```
GET    /mobility/api/transport-lines              # Liste toutes les lignes
GET    /mobility/api/transport-lines/number/{num} # Ligne par numero
GET    /mobility/api/transport-lines/type/{type}  # Lignes par type
GET    /mobility/api/schedules/line/{lineNumber}  # Horaires d'une ligne
GET    /mobility/api/traffic-info                 # Infos trafic
GET    /mobility/api/traffic-info/active          # Trafic actif seulement
POST   /mobility/api/transport-lines              # Creer une ligne
PUT    /mobility/api/transport-lines/{id}         # Modifier une ligne
DELETE /mobility/api/transport-lines/{id}         # Supprimer une ligne
```

Listing 2.1 – Endpoints REST du service Mobilité

URLs d'Accès

Type	URL
Direct	http://localhost:8081/mobility/api/transport-lines
Via Gateway	http://localhost:8080/api/mobility/transport-lines
Health Check	http://localhost:8081/mobility/actuator/health
H2 Console	http://localhost:8081/mobility/h2-console

Données de test

- **5 lignes :** BUS-101, BUS-202, METRO-RED, METRO-BLUE, TRAIN-EX1
- **Stations :** Central Station, Downtown Mall, City Park, North Station, etc.
- **Horaires :** Générés automatiquement de 6h à 23h

Justification du Choix REST

Standard web le plus utilisé

Facilité d'utilisation et de debug

Support universel dans tous les langages

Idéal pour opérations CRUD sur ressources

Documentation Swagger auto-générée

Service Qualité de l'Air (SOAP)

accentorange

Service Qualité de l'Air — SOAP
Protocole : SOAP (XML) **Port :** 8082
Context Path : `/airquality` **Framework :** Spring Boot + Spring WS

Fonctionnalités principales :
- Consultation de l'indice AQI par zone
- Détails des polluants (PM2.5, PM10, NO2, CO2, O3)
- Comparaison entre zones
- Liste de toutes les zones surveillées

Opérations SOAP

```
GetAirQuality       - Obtenir l'AQI d'une zone specifique
GetAllZones         - Liste toutes les zones
GetZoneDetails      - Details complets d'une zone
```

```
GetPollutants       - Details des polluants
CompareZones        - Comparer deux zones
```

Listing 2.2 – Opérations SOAP disponibles

Exemple de Requête SOAP

```
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:air="http://smartcity.com/airquality">
   <soapenv:Body>
      <air:GetAirQualityRequest>
         <air:zoneName>Centre-ville</air:zoneName>
      </air:GetAirQualityRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Listing 2.3 – Exemple de requête SOAP GetAirQuality

URLs d'Accès

Type	URL
SOAP Endpoint	http://localhost:8082/airquality/ws
WSDL	http://localhost:8082/airquality/ws/airquality.wsdl
Health Check	http://localhost:8082/airquality/actuator/health
H2 Console	http://localhost:8082/airquality/h2-console

Niveaux AQI

TABLE 2.1 – Échelle de l'indice de qualité de l'air (AQI)

Plage	Niveau	Description
0–50	Bon	Qualité de l'air satisfaisante
51–100	Modéré	Qualité acceptable
101–150	Mauvais pour sensibles	Groupes sensibles affectés
151–200	Mauvais	Effets sur la santé possibles
201+	Très mauvais	Alerte sanitaire

Justification du Choix SOAP

Protocole standard pour systèmes legacy

Contrat strict avec WSDL

Forte typage des données

Support des transactions complexes

Conformité réglementaire requise pour données environnementales

Service Urgences (gRPC)

accentred

Service Urgences — gRPC
Protocole : gRPC (Protocol Buffers) **Ports :** 9090 (gRPC), 8083 (HTTP)
Framework : Spring Boot + grpc-spring-boot-starter

Fonctionnalités principales :
- Création d'alertes d'urgence en temps réel
- Suivi des interventions
- Gestion des ressources (ambulances, pompiers, police)
- Streaming d'alertes
- Statistiques sur les urgences

Définition Protocol Buffers

```
service EmergencyService {
    rpc CreateEmergency(EmergencyRequest) returns (EmergencyResponse);
    rpc GetEmergency(GetEmergencyRequest) returns (EmergencyResponse);
    rpc ListEmergencies(ListRequest) returns (EmergencyListResponse);
    rpc UpdateEmergency(UpdateRequest) returns (EmergencyResponse);
    rpc DeleteEmergency(DeleteRequest) returns (DeleteResponse);
    rpc GetStatistics(StatsRequest) returns (StatsResponse);
}
```

Listing 2.4 – Méthodes RPC définies dans emergency.proto

Types et Niveaux de Sévérité

Table 2.2 – Types d'urgence

Code	Description
FIRE	Incendie
ACCIDENT	Accident circulation
MEDICAL	Urgence médicale
CRIME	Incident criminel
NATURAL	Catastrophe naturelle

Table 2.3 – Niveaux de sévérité

Niveau	Priorité
LOW	Faible
MEDIUM	Moyen
HIGH	Élevé
CRITICAL	Critique

Exemple de Requête REST (Wrapper)

```
POST http://localhost:8083/api/emergencies
Content-Type: application/json

{
  "reporterId": "user123",
  "emergencyType": "FIRE",
  "severityLevel": "HIGH",
  "location": "Downtown",
  "latitude": 48.8566,
  "longitude": 2.3522,
  "description": "Building fire",
  "affectedPeople": 10,
  "tags": ["fire", "urgent"]
}
```

Listing 2.5 – Création d'une urgence via le wrapper REST

Justification du Choix gRPC

Performance critique pour les urgences

— Performance extrême (format binaire Protocol Buffers)
— Communication bidirectionnelle (streaming)
— Latence minimale — essentiel pour les urgences
— HTTP/2 multiplexing
— Typage fort et compact

Service Événements Urbains (GraphQL)

secondaryblue

Service Événements — GraphQL
Protocole : GraphQL **Port :** 8084
Framework : Spring Boot + graphql-spring-boot-starter

Fonctionnalités principales :
- Gestion des événements urbains
- Calendrier et recherche flexible
- Catégories variées (festivals, conférences, sports)
- Inscriptions aux événements
- Requêtes personnalisées

Schéma GraphQL

```
type Query {
    getAllEvents: [Event!]!
    getEvent(id: ID!): Event
    getEventsByCategory(category: String!): [Event!]!
    getUpcomingEvents: [Event!]!
    searchEvents(query: String!): [Event!]!
}

type Mutation {
    createEvent(input: EventInput!): Event!
    updateEvent(id: ID!, input: EventInput!): Event
    deleteEvent(id: ID!): Boolean!
    registerForEvent(eventId: ID!): Registration!
}
```

Listing 2.6 – Queries GraphQL disponibles

Exemple de Query

```
query {
  getAllEvents {
    id
    title
    description
    location
    startDateTime
    endDateTime
```

```
9       category
10      capacity
11      availableSpots
12      registeredCount
13    }
14  }
```

Listing 2.7 – Exemple de requête GraphQL

Catégories d'Événements

TABLE 2.4 – Catégories d'événements supportées

Catégorie	Description	Icône
FESTIVAL	Festivals et fêtes	
CONFERENCE	Conférences professionnelles	
SPORT	Événements sportifs	
CULTURAL	Événements culturels	
COMMUNITY	Événements communautaires	
WORKSHOP	Ateliers et formations	

URLs d'Accès

Type	URL
GraphQL Endpoint	http://localhost:8084/graphql
GraphiQL UI	http://localhost:8084/graphiql
Via Gateway	http://localhost:8080/api/events/graphql
Health Check	http://localhost:8084/actuator/health

Justification du Choix GraphQL

Requêtes flexibles et personnalisées

Évite l'over-fetching et under-fetching

Un seul endpoint pour toutes les opérations

Typage fort avec Schema Definition Language

Introspection du schéma

Excellent pour exploration de données variées

Service d'Orchestration

primaryblue

Service d'Orchestration
Protocole : REST **Port :** 8085
Context Path : `/orchestration` **Framework :** Spring Boot + WebClient

L'orchestrateur coordonne les appels à plusieurs services pour réaliser des **workflows métier complexes** qui nécessitent des données de plusieurs sources.

Workflow 1 : Planification de Trajet Intelligent

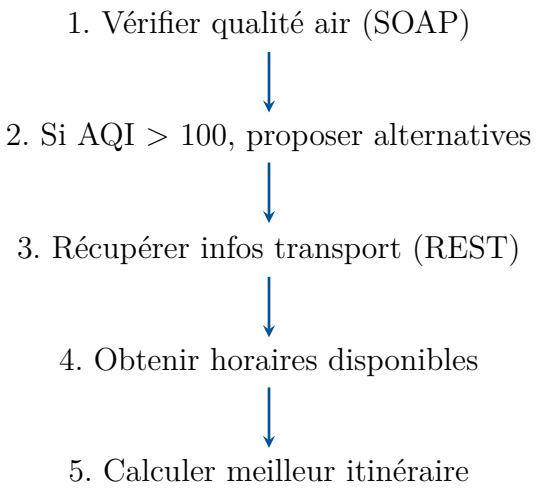

FIGURE 2.1 – Workflow de planification de trajet

Endpoint : `POST /orchestration/plan-journey`
Technologies : SOAP + REST

Workflow 2 : Gestion d'Urgence Contextualisée

Endpoint : `POST /orchestration/emergency-response`
Technologies : gRPC + SOAP + REST

1. Créer l'alerte d'urgence (gRPC)
2. Analyser la qualité de l'air de la zone (SOAP)
3. Rechercher les ressources disponibles (gRPC)
4. Évaluer l'impact sur le trafic (REST)
5. Générer des recommandations d'intervention

Workflow 3 : Dashboard Ville Intelligente

Endpoint : `GET /orchestration/dashboard`
Technologies : Tous les protocoles (REST, SOAP, gRPC, GraphQL)

Information

Appels parallèles à tous les services pour agréger une vue d'ensemble complète incluant : état du transport public, qualité de l'air globale, urgences actives et événements à venir.

Chapitre 3

API Gateway

Rôle et Responsabilités

L'API Gateway (Spring Cloud Gateway) sert de **point d'entrée unique** pour tous les clients.

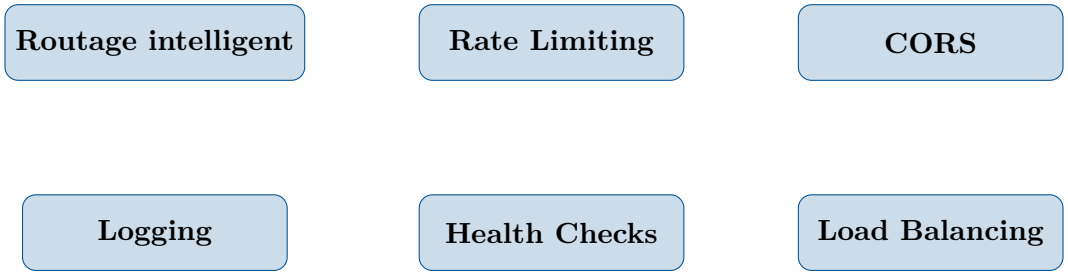

FIGURE 3.1 – Fonctionnalités de l'API Gateway

Routes Exposées

```
routes:
  /api/mobility/**          -> mobility-service (8081)
  /api/air-quality/**       -> air-quality-service (8082)
  /api/emergency/**         -> emergency-service (8083)
  /api/events/**            -> urban-events-service (8084)
  /api/orchestration/**     -> orchestration-service (8085)
```

Listing 3.1 – Configuration des routes

Sécurité

- **CORS :** Configuration pour permettre l'accès depuis le client web
- **Headers HTTP sécurisés :** Protection XSS, clickjacking
- **Error Handling :** Pas de fuite d'informations sensibles
- **Rate Limiting :** Protection DDoS (configurable)

Chapitre 4

Déploiement avec Docker

Architecture de Déploiement

Chaque service est emballé dans son propre conteneur Docker avec un processus de **multi-stage build**.

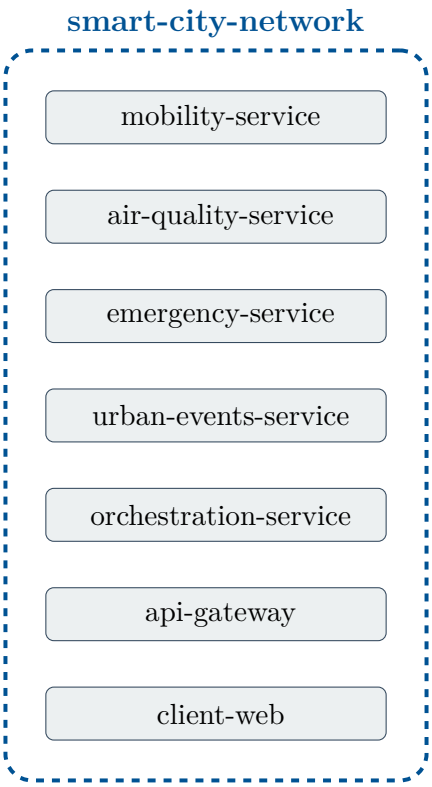

FIGURE 4.1 – Conteneurs Docker sur le réseau smart-city-network

Multi-Stage Build

```
# Stage 1: Build avec Maven
FROM maven:3.9.5-eclipse-temurin-17 AS build
WORKDIR /app
COPY pom.xml .
COPY src ./src
RUN mvn clean package -DskipTests

```

```
# Stage 2: Runtime avec OpenJDK
FROM eclipse-temurin:17-jre-alpine
WORKDIR /app
COPY --from=build  target/*.jar app.jar
EXPOSE 8081
ENTRYPOINT ["java", "-jar", " app.jar"]
```

Listing 4.1 – Dockerfile multi-stage pour services Java

Avantages du multi-stage build

- Image finale légère (150MB vs 700MB+)
- Pas d'outils de build en production
- Sécurité améliorée
- Temps de déploiement réduit

Health Checks

```
healthcheck:
  test: ["CMD", "curl", "-f",
         "http://localhost:8081/mobility/actuator/health"]
  interval: 30s
  timeout: 10s
  retries: 3
  start_period: 60s
```

Listing 4.2 – Configuration des health checks

Commandes de Déploiement

TABLE 4.1 – Commandes Docker Compose essentielles

Commande	Description
`docker-compose build -no-cache`	Construction complète
`docker-compose up -d`	Démarrage en arrière-plan
`docker-compose ps`	Vérification des statuts
`docker-compose logs -f`	Logs en temps réel
`docker-compose down`	Arrêt des services
`docker-compose down -v`	Arrêt + suppression volumes

Chapitre 5

Interface Client Web

Technologies Frontend

Technologie	Version	Rôle
React	18.2.0	Framework UI
Vite	5.0.0	Build tool
Axios	1.6.0	Client HTTP
Nginx	Alpine	Serveur web production

Fonctionnalités

1. **Dashboard :** Vue d'ensemble via Orchestration Service
2. **Mobilité :** Consultation lignes, horaires, trafic
3. **Qualité de l'Air :** Vérification AQI par zone
4. **Urgences :** Création et liste d'alertes
5. **Événements :** Exploration via GraphQL
6. **Workflows :** Exécution de workflows orchestrés

Architecture Frontend

```
client-web/
|-- src/
|   |-- App.jsx              # Composant principal
|   |-- components/          # Composants reutilisables
|   |-- services/            # API calls
|   |-- styles/              # CSS
|-- public/
|-- Dockerfile               # Multi-stage build
|-- nginx.conf               # Configuration Nginx
|-- package.json
```

Listing 5.1 – Structure du projet client-web

Communication API

```
const API_BASE_URL = 'http://localhost:8080';

// Recuperer les lignes de transport
axios.get(`${API_BASE_URL}/api/mobility/transport-lines`)
  .then(response => setTransportLines(response.data))
  .catch(error => console.error(error));
```

Listing 5.2 – Exemple d'appel API via Axios

Chapitre 6

Tests et Validation

Tests Réalisés

TABLE 6.1 – Résumé des tests effectués

Type	Description	Statut
Unitaires	Health checks fonctionnels	✓
Unitaires	Endpoints REST validés	✓
Unitaires	Opérations SOAP testées avec SoapUI	✓
Unitaires	Méthodes gRPC testées avec grpcurl	✓
Unitaires	Queries GraphQL testées avec GraphiQL	✓
Intégration	Communication inter-services	✓
Intégration	Workflows de bout en bout	✓
Intégration	Gestion d'erreurs cascade	✓
Intégration	API Gateway routing	✓

Métriques de Performance

TABLE 6.2 – Latence moyenne par protocole

Protocole	Latence	Throughput
REST	10–50 ms	100+ req/s
SOAP	20–80 ms	100+ req/s
gRPC	**5–20 ms**	100+ req/s
GraphQL	15–60 ms	100+ req/s

Information

Le protocole gRPC offre la meilleure performance grâce à son format binaire Protocol Buffers et HTTP/2, ce qui le rend idéal pour les communications temps réel comme le service d'urgences.

Chapitre 7

Résultats et Performances

Métriques Globales

TABLE 7.1 – Métriques de la plateforme

Métrique	Valeur
Temps de démarrage complet	∼2–3 minutes (tous services)
Temps de réponse orchestrateur	100–300 ms (appels multiples)
Utilisation mémoire par service Java	∼300–500 MB
Utilisation mémoire API Gateway	∼400 MB
Utilisation mémoire Client Web (Nginx)	∼50 MB
Total RAM recommandé	2–3 GB

Avantages de l'Architecture

	Avantage	Description
✓	Scalabilité	Chaque service peut être scalé indépendamment
✓	Résilience	Isolation des pannes entre services
✓	Maintenabilité	Code modulaire et bien organisé
✓	Interopérabilité	Support de multiples protocoles
✓	Déployabilité	Conteneurisation complète avec Docker
✓	Testabilité	Services isolés faciles à tester
✓	Performance	gRPC pour urgences, GraphQL pour flexibilité

Chapitre 8

Difficultés Rencontrées et Solutions

Problèmes et Résolutions

Problème	Cause	Solution
Communication inter-services	Protocoles hétérogènes (REST, SOAP, gRPC, GraphQL)	Orchestrateur comme médiateur + WebClient réactif
Configuration SOAP	Génération WSDL complexe	`@Endpoint`, `@PayloadRoot`, config XSD
gRPC avec Spring Boot	Intégration non native	`grpc-spring-boot-starter` + REST wrapper
GraphQL Schema	Définition résolveurs	Fichier `schema.graphqls` + annotations mapping
Context Path Services	Conflits de paths	Context paths distincts par service
H2 Console Access	URLs avec context paths	Configuration explicite `spring.h2.console.path`
Images Docker lourdes	Build tools en production	Multi-stage builds Maven → JRE Alpine
Gestion des erreurs	Propagation entre services	`@ControllerAdvice` + logging SLF4J

TABLE 8.1 : Synthèse des difficultés et solutions

Chapitre 9

Améliorations Futures

Court Terme

- ☐ Ajouter authentification JWT/OAuth2
- ☐ Implémenter base de données persistante (PostgreSQL)
- ☐ Ajouter tests automatisés (JUnit 5, Mockito)
- ☐ Améliorer gestion d'erreurs avec Circuit Breaker
- ☐ Ajouter métriques Prometheus
- ☐ Implémenter API versioning

Moyen Terme

- ☐ Service mesh (Istio)
- ☐ Cache distribué (Redis)
- ☐ Message queue (RabbitMQ, Kafka)
- ☐ ELK Stack pour logging centralisé
- ☐ Distributed tracing (Zipkin/Jaeger)

Long Terme

- ☐ Déploiement Kubernetes
- ☐ CI/CD complet (Jenkins/GitLab CI)
- ☐ Monitoring (Prometheus + Grafana)
- ☐ Service discovery avec Eureka
- ☐ Configuration centralisée (Spring Cloud Config)

Conclusion

Ce projet démontre avec succès l'implémentation d'une **plateforme de services interopérables** utilisant les 4 protocoles demandés (REST, SOAP, gRPC, GraphQL). L'architecture microservices adoptée offre flexibilité, scalabilité et résilience.

Points Clés Réalisés

Réalisations principales

- **Architecture Microservices Complète** — 6 services indépendants
- **4 Protocoles de Communication** — REST, SOAP, gRPC, GraphQL
- **Orchestration** — Workflows métier complexes multi-protocoles
- **Déploiement** — Dockerisation complète avec multi-stage builds
- **Documentation** — Documentation technique complète

Enseignements Tirés

Technique :

- Chaque protocole a ses forces et cas d'usage optimaux
- L'interopérabilité est possible avec une bonne architecture
- L'orchestration centralise la logique métier complexe

Méthodologique :

- Architecture modulaire facilite maintenance et évolution
- Tests à chaque niveau sont essentiels
- Documentation claire économise du temps

Perspectives

Cette plateforme constitue une base solide pour une véritable application de ville intelligente. Elle pourrait être étendue avec :

- Gestion de l'énergie (smart grid)
- Parking intelligent
- Sécurité publique
- Gestion des déchets
- Éclairage public intelligent

Annexe A

Commandes Utiles

```
# Construction et demarrage
docker-compose up --build -d

# Verification des services
docker-compose ps

# Logs en temps reel
docker-compose logs -f

# Health checks
curl http://localhost:8080/actuator/health
curl http://localhost:8081/mobility/actuator/health
curl http://localhost:8082/airquality/actuator/health
curl http://localhost:8083/api/emergencies/health
curl http://localhost:8084/actuator/health
curl http://localhost:8085/orchestration/health

# Arret
docker-compose down

# Nettoyage complet
docker system prune -a --volumes
```

Listing A.1 – Commandes de déploiement et vérification

Annexe B

Ports Utilisés

TABLE B.1 – Récapitulatif des ports

Service	Port(s)	Protocole	Context Path
Client Web	3000	HTTP	/
API Gateway	8080	HTTP	/
Mobility Service	8081	HTTP/REST	/mobility
Air Quality Service	8082	HTTP/SOAP	/airquality
Emergency Service	8083, 9090	HTTP, gRPC	/
Urban Events Service	8084	HTTP/GraphQL	/
Orchestration Service	8085	HTTP/REST	/orchestration

Annexe C

Structure du Projet

```
smart-city-platform/
|-- api-gateway/
|   |-- src/main/java/
|   |-- src/main/resources/
|   |-- pom.xml
|   |-- Dockerfile
|-- mobility-service/
|   |-- src/main/java/
|   |-- src/main/resources/
|   |   |-- application.yml
|   |   |-- data.sql
|   |-- pom.xml
|   |-- Dockerfile
|-- air-quality-service/
|   |-- src/main/java/
|   |-- src/main/resources/
|   |   |-- application.yml
|   |   |-- airquality.xsd
|   |   |-- data.sql
|   |-- pom.xml
|   |-- Dockerfile
|-- emergency-service/
|   |-- src/main/java/
|   |-- src/main/proto/
|   |   |-- emergency.proto
|   |-- src/main/resources/
|   |-- pom.xml
|   |-- Dockerfile
|-- urban-events-service/
|   |-- src/main/java/
|   |-- src/main/resources/
|   |   |-- graphql/
|   |   |   |-- schema.graphqls
|   |-- pom.xml
|   |-- Dockerfile
|-- orchestration-service/
|-- client-web/
|-- docker-compose.yml
|-- README.md
```

Listing C.1 – Arborescence complète du projet

Projet réalisé par : Rana ROMDHANE
Encadré par : Mme. Abir CHAABANI

Matière : Service Oriented Computing

Année universitaire : 2025-2026

Version 1.0.0 — Décembre 2025