

École Marocaine des Sciences de l'Ingénieur (EMSI)

INGÉNIERIE INFORMATIQUE ET RÉSEAUX

Rapport Technique

Event Platform — Architecture Microservices

(Spring Cloud Config, Eureka, API Gateway, Services Métier)

Réalisé par :

Anas KRIR

Adam EL YOURI

Ismail MACHHOUR

Mohamed Taha MALLOUK

Encadrant :

Pr. Abdelmalek JEOUIT

Année universitaire : 2025–2026

Rabat, le 29 décembre 2025

Table des matières

1 Contexte et objectifs	3
2 Choix d'architecture	3
2.1 Pourquoi des microservices ?	3
2.2 Briques Spring Cloud retenues	3
2.3 Base de données	3
3 Vue globale de l'architecture	3
3.1 Composants et ports	4
4 Communication et flux de données	4
4.1 Routage via l'API Gateway	4
4.2 Découverte des services (Eureka)	5
4.3 Scénario métier : réservation d'un ticket	5
5 Déploiement et exécution	5
5.1 Ordre de démarrage (mode local)	5
5.2 Variables d'environnement	5
6 Difficultés rencontrées et solutions	5
6.1 Problèmes Docker (ports et containers)	5
6.2 Dépendances Maven et instabilités réseau	6
6.3 Démarrage distribué (dépendances entre services)	6
6.4 CORS et accès Frontend → API	6
6.5 Gestion des erreurs métier	6
7 Conclusion et perspectives	6

1 Contexte et objectifs

- Le projet **Event Platform** vise à fournir une plateforme web permettant :
- l'**authentification** et la gestion des utilisateurs,
 - la **publication** et la consultation d'événements,
 - la **réservation** de tickets,
 - la gestion du **paiement** et la génération d'une confirmation,
 - l'envoi de **notifications** (ex. réservation réussie).

Le système adopte une approche **microservices** afin d'isoler les responsabilités, faciliter la maintenance, et permettre une montée en charge plus ciblée (par exemple, le service de réservation peut être répliqué indépendamment).

2 Choix d'architecture

2.1 Pourquoi des microservices ?

- Nous avons choisi une architecture microservices pour :
- **Séparer les domaines** (utilisateurs, événements, réservations, paiements, notifications).
 - **Réduire le couplage** : chaque service expose une API REST claire.
 - **Améliorer l'évolutivité** : possibilité de scaler uniquement les services les plus sollicités.
 - **Faciliter l'évolution** : ajout d'une nouvelle fonctionnalité sans impacter tout le monolith.

2.2 Briques Spring Cloud retenues

L'écosystème Spring Cloud a été sélectionné car il apporte des composants standards pour les architectures distribuées :

- **Config Server** : centralisation et versioning de la configuration.
- **Eureka Server** : découverte dynamique des services (service discovery).
- **API Gateway** : point d'entrée unique, routage, filtrage, CORS, et simplification côté frontend.

2.3 Base de données

Une base **MySQL** est utilisée pour persister les données (via JPA/Hibernate). Dans une architecture microservices « idéale », chaque service peut posséder sa base dédiée (*pattern Database per Service*). Dans notre version, une base MySQL est utilisée de manière centralisée pour simplifier la mise en place, tout en gardant des schémas/tables séparés par domaine.

3 Vue globale de l'architecture

La Figure 1 illustre les composants et les flux principaux.

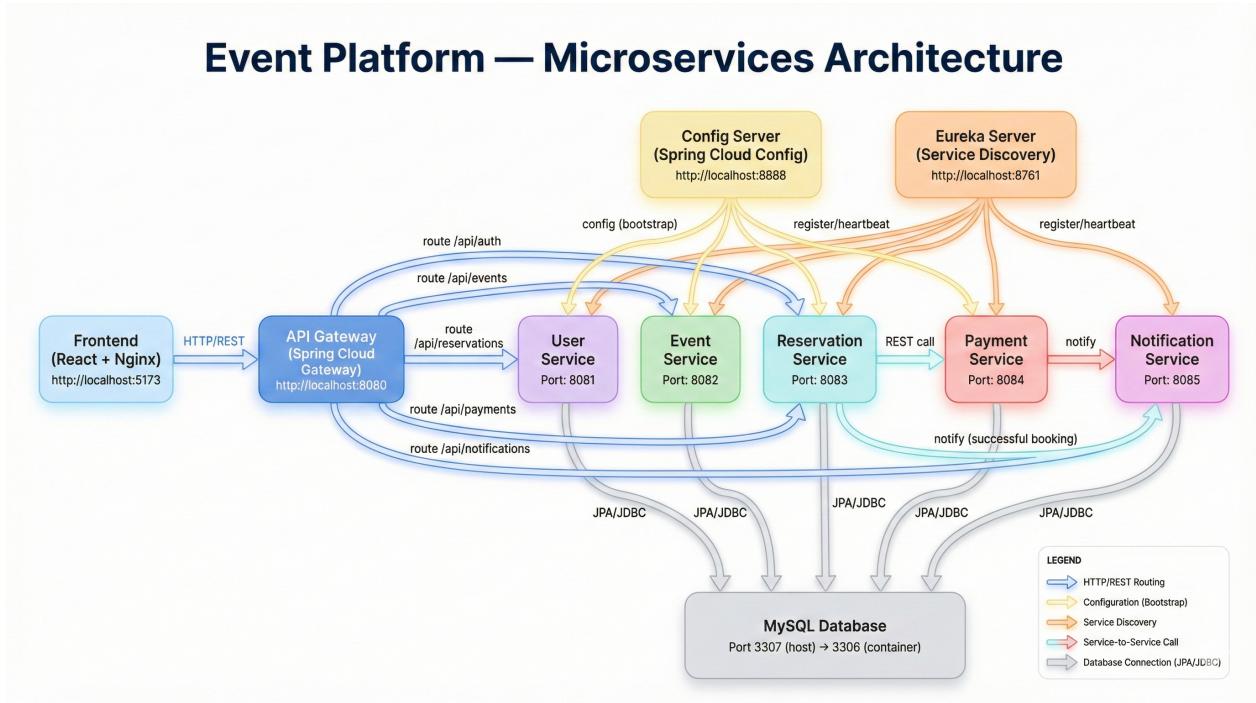


FIGURE 1 – Diagramme d'architecture — microservices, flux et communication

3.1 Composants et ports

Le projet est composé des services suivants (ports par défaut) :

Composant	Port	Rôle
Config Server	8888	Configuration centralisée (Spring Cloud Config)
Eureka Server	8761	Registre de services (Service Discovery)
API Gateway	8080	Point d'entrée unique, routage & filtres
User Service	8081	Authentification, utilisateurs, rôles
Event Service	8082	CRUD d'événements, tickets, disponibilité
Reservation Service	8083	Réservations, logique métier de booking
Payment Service	8084	Paiement, validation et statut
Notification Service	8085	Notifications (confirmation, messages)
Frontend (React + Nginx)	5173	Interface utilisateur web
MySQL	3306/3307	Persistance des données (JPA/JDBC)

TABLE 1 – Services, ports et responsabilités

4 Communication et flux de données

4.1 Routage via l'API Gateway

Le frontend ne communique pas directement avec tous les microservices : il envoie ses requêtes vers l'**API Gateway**, qui :

- route les chemins (`/api/auth`, `/api/events`, `/api/reservations`, etc.),
- applique éventuellement des règles transverses (CORS, logs, headers),
- simplifie la configuration côté client (un seul endpoint).

4.2 Découverte des services (Eureka)

Les microservices s'enregistrent automatiquement sur **Eureka** (register/heartbeat). Ainsi, la Gateway peut router vers un service sans dépendre d'une IP fixe : cela facilite le déploiement et la scalabilité.

4.3 Scénario métier : réservation d'un ticket

Un scénario typique se déroule ainsi :

1. Le client consulte la liste des événements (**Event Service**) via la Gateway.
2. Le client effectue une réservation (**Reservation Service**).
3. Le service réservation vérifie la disponibilité (tickets restants) et persiste la réservation.
4. Il déclenche ensuite l'étape de paiement (**Payment Service**) via appel REST.
5. En cas de succès, le **Notification Service** envoie une confirmation (email/message) et l'état final est enregistré.

Ce découpage met en évidence une problématique classique : la **cohérence des données** entre services (réservation/paiement/notification). Dans notre version, nous restons sur un flux synchrone REST, tout en veillant à gérer correctement les échecs (annulation/rollback logique, messages d'erreur clairs).

5 Déploiement et exécution

5.1 Ordre de démarrage (mode local)

Lorsque Docker Compose n'est pas stable dans l'environnement, l'application peut être lancée manuellement en respectant l'ordre suivant :

- Config Server (8888)
- Eureka Server (8761)
- API Gateway (8080)
- User Service (8081)
- Event Service (8082)
- Reservation Service (8083)
- Payment Service (8084)
- Notification Service (8085)

Cette séquence limite les erreurs de démarrage : les services ont besoin que la configuration soit disponible (Config Server) et que l'annuaire Eureka soit prêt pour l'enregistrement et le routage.

5.2 Variables d'environnement

L'utilisation de fichiers `.env` (ou profils Spring) permet de paramétrier :

- URL de Config Server, URL de Eureka,
- credentials MySQL, URL de base de données,
- ports, profils (`dev`, `docker`, etc.).

6 Difficultés rencontrées et solutions

6.1 Problèmes Docker (ports et containers)

Une difficulté fréquente est le conflit de ports, par exemple :

- **3306 déjà utilisé** sur la machine hôte (MySQL local déjà démarré).

Solution : changer le port exposé côté host (ex. 3307:3306) ou arrêter le service MySQL local avant de lancer Docker.

6.2 Dépendances Maven et instabilités réseau

Lors des builds Docker, Maven peut échouer (erreurs temporaires du dépôt central, erreurs 500, timeouts). **Solutions appliquées :**

- ajout d'un **cache Maven** dans la build Docker (gain de performance & stabilité),
- ajout d'un **settings.xml** si nécessaire (mirrors/retry),
- relancer la build après une erreur serveur temporaire.

6.3 Démarrage distribué (dépendances entre services)

Les microservices démarrent parfois trop tôt, avant que Config/Eureka soit prêt. **Solutions :**

- imposer un **ordre de démarrage** en local,
- configurer des **retries** côté clients (Spring Cloud),
- vérifier l'enregistrement sur Eureka avant de tester via la Gateway.

6.4 CORS et accès Frontend → API

Le frontend (port 5173) peut être bloqué par le navigateur si CORS n'est pas correctement configuré. **Solution :** activer CORS au niveau Gateway (ou services) et normaliser les routes d'API.

6.5 Gestion des erreurs métier

Dans un flux réservation → paiement → notification, une erreur de paiement doit être gérée proprement. **Solution :** définir des statuts explicites (*PENDING*, *PAID*, *FAILED*) et retourner des réponses REST cohérentes.

7 Conclusion et perspectives

Cette architecture microservices a permis de structurer le projet autour de domaines clairs, tout en s'appuyant sur les bonnes pratiques Spring Cloud (Config, Eureka, Gateway). Malgré les difficultés liées au déploiement (Docker/ports), aux dépendances Maven et au démarrage distribué, les solutions mises en place ont permis d'obtenir une plateforme fonctionnelle et extensible.

Améliorations possibles

- Introduire une communication asynchrone (Kafka/RabbitMQ) pour les notifications et la résilience.
- Observabilité : logs centralisés, metrics (Prometheus/Grafana), tracing (Zipkin).

Fin du rapport.