Rapport ARC42 Architecture Microservices avec API Gateway

Évolution Labs 2 à 5 – LOG430

Auteur : Chawki Benzeghiba
Date : Août 2025
Projet : Architecture Microservices avec API Gateway – Labo 2,4,5 LOG430

GitHub Lab2 : https://github.com/ChawkiBenzeghiba/Lab2-LOG430
GitHub Lab4 : https://github.com/ChawkiBenzeghiba/Lab4-LOG430
GitHub Lab5 : https://github.com/ChawkiBenzeghiba/Lab5-LOG430

---

1. Introduction et Objectifs

1.1 Contexte et portée

Ce document décrit l'évolution du système multi-magasins (Lab 2) vers une architecture microservices moderne (Lab 5), en passant par les tests de performance et monitoring (Lab 4). L'architecture finale intègre un API Gateway, du load balancing, et une observabilité complète avec Prometheus et Grafana.

1.2 Objectifs architecturaux

Objectifs fonctionnels

- **Migration vers microservices** : Décomposer l'architecture monolithique en services indépendants
- **API Gateway centralisé** : Point d'entrée unique pour tous les services avec routage intelligent
- **Load balancing** : Distribution de charge entre instances multiples du service Panier
- **Observabilité complète** : Monitoring des performances et métriques en temps réel
- **Comparaison de performance** : Analyse comparative entre architecture monolithique et microservices

Objectifs non-fonctionnels (attributs de qualité)

- **Scalabilité** : Support de multiples instances par service et distribution de charge
- **Résilience** : Isolation des pannes entre services et récupération automatique
- **Performance** : Optimisation via cache, compression et connection pooling
- **Maintenabilité** : Services indépendants et déploiement automatisé
- **Observabilité** : Monitoring complet avec métriques, logs et alertes

1.3 Parties prenantes

Rôle	Responsabilités	Enjeux principaux

Rôle	Responsabilités	Enjeux principaux
Architecte	Conception de l'architecture microservices	Scalabilité et performance
Développeur	Implémentation des microservices	Maintenabilité et tests
DevOps	Déploiement et monitoring	Observabilité et résilience
Utilisateur final	Utilisation de l'interface web	Performance et disponibilité

2. Contraintes

2.1 Contraintes techniques

- **Plate-forme** : Node.js v18
- **Framework** : Express pour tous les services
- **ORM** : Sequelize pour la persistance
- **Base de données** : PostgreSQL (une par service)
- **Conteneurisation** : Docker & Docker Compose
- **API Gateway** : Express avec http-proxy-middleware
- **Load Balancing** : Round-robin entre instances
- **Monitoring** : Prometheus & Grafana
- **Sécurité** : CORS, Helmet, validation des entrées

2.2 Contraintes organisationnelles

- **Évolution progressive** : Lab 2 → Lab 4 → Lab 5
- **Réutilisation** de la logique métier du Lab 2
- **Tests de performance** basés sur les expériences du Lab 4
- **Documentation obligatoire** : README, ARC42, UML et ADR
- **Déploiement reproductible** : un seul `docker-compose up` suffit

2.3 Contraintes d'évolution (Lab 2 → Lab 5)

Éléments à conserver

- Logique métier des cas d'utilisation (UC1, UC2, UC3)
- Modèles de données de base : `Produit`, `Vente`, `Stock`
- Tests automatisés et pipeline CI/CD

Éléments à modifier

- **Architecture** : passer du monolithique 3-tiers aux microservices
- **Persistance** : séparer les bases de données par service
- **Communication** : remplacer les appels directs par HTTP REST

Éléments à ajouter

- **API Gateway** : routage et load balancing centralisé
- **Microservices** : services indépendants pour chaque domaine

- **Observabilité** : monitoring avancé avec Prometheus/Grafana
- **Load Balancing** : distribution de charge entre instances

---

3. Contexte

3.1 Contexte Métier

- **Domaine** : Gestion de ventes et de stocks pour un réseau de magasins physiques
- **Évolution** : Passage d'une architecture centralisée à une architecture distribuée
- **Enjeux** : Scalabilité, résilience et performance pour supporter la croissance

3.2 Évolution des Labs

Lab 2 – API RESTful Multi-Magasins (3-tiers)

```
[ Client HTTP ]  ⇄  [ API Express ]  ⇄  [ PostgreSQL ]
     Tier 1              Tier 2            Tier 3
```

- Architecture monolithique en 3 tiers
- Gestion centralisée des magasins et stocks
- Tests automatisés et pipeline CI/CD

Lab 4 – Tests de Performance et Monitoring

```
[ Load Tests ]  ⇄  [ Nginx ]  ⇄  [ App Instances ]  ⇄  [ PostgreSQL +
Redis ]
```

- Tests de charge et de stress
- Monitoring avec Prometheus et Grafana
- Optimisations de performance (cache Redis)
- Analyse des métriques de latence et throughput

Lab 5 – Architecture Microservices avec API Gateway

```
[ Frontend ]  ⇄  [ API Gateway ]  ⇄  [ Microservices ]  ⇄  [ PostgreSQL ]
     Tier 1           Tier 2              Tier 3              Tier 4
```

- Architecture microservices distribuée
- API Gateway avec load balancing
- Observabilité complète
- Comparaison de performance avec l'architecture monolithique

---

4. Stratégie de Solution

4.1 Architecture cible

- **Microservices** : 6 services indépendants (Produits, Clients, Stock, Panier, Commandes, Ventes)
- **API Gateway** : Point d'entrée unique avec routage et load balancing
- **Load Balancing** : Round-robin entre 2 instances du service Panier
- **Observabilité** : Prometheus pour les métriques, Grafana pour la visualisation
- **Base de données** : PostgreSQL séparée par service

4.2 Approche Domain-Driven Design (DDD)

- **Domaine Principal (Core Domain) : Gestion des Ventes**
  - **Description** : cœur métier, enregistrement et traitement des ventes
  - **Contexte délimité** : Ventes
  - **Langage Ubiquitaire** : Vente(id, produitId, magasinId, quantite, prixUnitaire, dateVente), produitId, magasinId, quantite, prixUnitaire
  - **Composants d'architecture** :
    - Service : `service-ventes`
    - Modèles : `Vente`
    - Controller : `venteController.js`
- **Domaine Support (Supporting Domain) : Gestion des Paniers**
  - **Description** : gestion des paniers d'achat avec load balancing
  - **Contexte délimité** : Panier d'achat
  - **Langage Ubiquitaire** : Panier(id, clientId, items, total), clientId, produitId, quantite, prixUnitaire, total
  - **Composants d'architecture** :
    - Service : `service-panier` (2 instances)
    - Modèles : `Panier`
    - Controller : `panierController.js`
- **Domaine Support (Supporting Domain) : Gestion des Commandes**
  - **Description** : validation et traitement des commandes
  - **Contexte délimité** : Commandes
  - **Langage Ubiquitaire** : Commande(id, clientId, items, statut, dateCreation), clientId, statut, adresseLivraison, methodePaiement
  - **Composants d'architecture** :
    - Service : `service-commandes`
    - Modèles : `Commande`, `CommandeItem`
    - Controller : `commandeController.js`
- **Domaine Support (Supporting Domain) : Gestion des Clients**
  - **Description** : gestion des comptes et profils clients
  - **Contexte délimité** : Clients

    - **Langage Ubiquitaire** : Client(id, nom, email, adresse), nom, email, adresse, telephone
    - **Composants d'architecture** :
        - Service : `service-clients`
        - Modèles : `Client`
        - Controller : `clientController.js`

- **Domaine Générique (Generic Domain) : Catalogue des Produits**
    - **Description** : gestion du référentiel produit
    - **Contexte délimité** : Catalogue de produits
    - **Langage Ubiquitaire** : Produit(id, nom, description, prix, categorie), nom, description, prix, categorie, imageUrl
    - **Composants d'architecture** :
        - Service : `service-produits`
        - Modèles : `Produit`
        - Controller : `produitController.js`

- **Domaine Générique (Generic Domain) : Gestion des Stocks**
    - **Description** : gestion des inventaires et réapprovisionnements
    - **Contexte délimité** : Stock et inventaire
    - **Langage Ubiquitaire** : StockCentral(id, produitId, quantiteStock), Magasin(id, nom, adresse), produitId, quantiteStock, quantiteMin, inventaire
    - **Composants d'architecture** :
        - Service : `service-stock`
        - Modèles : `StockCentral`, `Magasin`, `Produit`
        - Controller : `stockController.js`

---

5. Vue Architecturale

Dans cette section, les différentes vues architecturales basées sur le modèle 4+1 seront présentées.

5.1 Vue des Cas d'Utilisation

Cette vue affiche les différents cas d'utilisations implémentés dans le laboratoire 5.

Emplacement dans le projet: docs/UML/vue_cas_utilisation.puml

5.2 Vue de Déploiement

Cette vue décrit le côté physique de l'architecture avec laquelle le système est déployé. Elle montre les conteneurs Dockers et leurs communications.

Emplacement dans le projet: docs/UML/vue_deploiement.puml

5.3 Vue Implémentation

Cette vue démontre l'organisation du code et des différentes relations entre les composants. On y voit aisément la séparation des couches de la structure microservices.

Emplacement dans le projet: docs/UML/vue_implementation.puml

5.4 Vue Logique

Cette vue présente le modèle de domaine du système. On y voit les classes modèles, les contrôleurs, les services ainsi que les relations entre elles.

Emplacement dans le projet: docs/UML/vue_logique_classes_MVC.puml

5.5 Vue Processus

Voici les vues qui représentent les diagrammes de séquences des cas d'utilisation réalisés dans le laboratoire 5.

Emplacement dans le projet: docs/UML/vue_processus_UC1_migration_microservices.puml

Emplacement dans le projet: docs/UML/vue_processus_UC2_load_balancing.puml

Emplacement dans le projet: docs/UML/vue_processus_UC3_observabilite.puml

---

6. Concepts Transversaux

Cette section couvre les mécanismes et bonnes pratiques applicables à l'architecture microservices dans son ensemble.

6.1 Sécurité

- **CORS configuré**
  L'API Gateway implémente une configuration CORS stricte pour contrôler les origines autorisées.
- **Validation des entrées**
  Chaque microservice valide les données d'entrée avant traitement.
- **Headers de sécurité**
  Utilisation de Helmet pour sécuriser les en-têtes HTTP.
- **Protection contre l'injection SQL**
  Sequelize utilise des requêtes paramétrées dans tous les services.

6.2 Performance

- **Load Balancing**
  Distribution de charge round-robin entre instances du service Panier.
- **Connection Pooling**
  Configuration optimisée des pools de connexions PostgreSQL par service.
- **Cache intégré**
  L'API Gateway implémente un cache pour optimiser les performances.
- **Compression**
  Réponses compressées pour réduire la latence réseau.

6.3 Observabilité

- **Métriques Prometheus**
  Collecte de métriques de latence, throughput et erreurs pour tous les services.
- **Dashboards Grafana**
  Visualisation en temps réel des performances et de la santé du système.
- **Logging centralisé**
  Logs structurés pour tracer les requêtes à travers les microservices.
- **Health checks**
  Endpoints de santé pour chaque service et l'API Gateway.

6.4 Résilience

- **Isolation des pannes**
  Chaque microservice peut fonctionner indépendamment des autres.
- **Circuit Breaker**
  Protection contre les cascades de pannes entre services.
- **Retry policies**
  Tentatives automatiques en cas d'échec de communication.
- **Graceful degradation**
  Le système continue de fonctionner même si certains services sont indisponibles.

---

7. Tests

7.1 Tests de Performance

- **Fichiers** :
  `test-final-performance.sh`, `test-load-balancing.sh`, `test-gateway-exposure.sh`
- **Exécution** :
  `./test-final-performance.sh`
- **Types** :
  Tests de latence, load balancing, exposition via Gateway
- **Résultats** :
  Comparaison des performances entre architectures

7.2 Tests Fonctionnels

- **Tests unitaires** : Jest pour chaque microservice
- **Tests d'intégration** : Supertest pour les APIs
- **Tests de charge** : Scripts de test de performance
- **Tests de sécurité** : Validation CORS et entrées

7.3 Résultats de Performance

Architecture	Temps Moyen	Amélioration
Monolithique	~42ms	Référence
Microservices Direct	~38ms	-10%
Microservices via Gateway	~36ms	-14%

---

8. Architectural Decision Records (ADR)

ADR-001 — Migration vers Architecture Microservices

Statut

Accepté

Contexte

Le système multi-magasins (Lab 2) atteint ses limites en termes de scalabilité et maintenabilité. Il faut évoluer vers une architecture plus flexible pour supporter la croissance et faciliter les évolutions futures.

Décision

Adopter une architecture microservices avec API Gateway centralisé. Chaque domaine métier (Produits, Ventes, Stock, Panier, Commandes) devient un service indépendant avec sa propre base de données PostgreSQL.

Raisons

- Permet une scalabilité horizontale indépendante par service
- Facilite le développement et déploiement indépendants
- Améliore la résilience par isolation des pannes
- Simplifie la maintenance et les évolutions futures

Conséquences

- Complexité accrue de la communication inter-services
- Nécessité d'un API Gateway pour la coordination
- Gestion distribuée des données et de la cohérence
- Overhead de latence compensé par les optimisations

ADR-002 — API Gateway avec Load Balancing

Statut

Accepté

Contexte

L'architecture microservices nécessite un point d'entrée unique pour coordonner les requêtes et gérer la distribution de charge entre instances multiples.

Décision

Implémenter un API Gateway basé sur Express avec http-proxy-middleware pour le routage et un load balancer round-robin pour le service Panier.

Raisons

- Point d'entrée unique simplifie l'intégration client
- Load balancing améliore la disponibilité et les performances
- Centralisation de la sécurité et du monitoring
- Facilité d'ajout de nouvelles fonctionnalités (cache, compression)

Conséquences

- Single point of failure pour l'API Gateway
- Overhead de latence compensé par les optimisations
- Complexité de configuration et maintenance
- Nécessité de monitoring spécifique

ADR-003 — Observabilité avec Prometheus et Grafana

Statut

Accepté

Contexte

L'architecture distribuée nécessite une visibilité complète sur les performances, la santé et le comportement de chaque service.

Décision

Implémenter un système d'observabilité basé sur Prometheus pour la collecte de métriques et Grafana pour la visualisation.

Raisons

- Visibilité en temps réel sur tous les services
- Détection précoce des problèmes de performance
- Analyse comparative entre architectures
- Support des alertes et notifications

Conséquences

- Complexité de configuration et maintenance
- Consommation de ressources supplémentaires
- Nécessité de formation sur les outils
- Gestion des données de monitoring

---

9. Scénarios de Qualité

Cette section traduit les attributs de qualité en cas concrets et mesurables.

Scénario 1 : Scalabilité – Ajout d'une nouvelle instance de service

- **Source** : DevOps de l'équipe
- **Stimulus** : Le service Panier atteint sa capacité maximale et nécessite une troisième instance
- **Artefact** : Configuration Docker Compose et API Gateway
- **Environnement** : Environnement de production
- **Réponse** :
  1. Ajouter une troisième instance dans `docker-compose.yml`.
  2. Mettre à jour la configuration du load balancer dans l'API Gateway.
  3. Redéployer avec `docker-compose up --scale service-panier=3`.
- **Mesure de la réponse** : Le déploiement et la mise en service doivent prendre **moins de 5 minutes**.

Scénario 2 : Résilience – Panne d'un microservice

- **Source** : Utilisateur final
- **Stimulus** : Le service Stock tombe en panne pendant l'utilisation
- **Artefact** : Architecture microservices
- **Environnement** : Environnement de production
- **Réponse** :
  1. Les autres services continuent de fonctionner normalement.
  2. L'API Gateway retourne des erreurs 503 pour les requêtes vers le service Stock.
  3. Le monitoring Prometheus détecte la panne et génère une alerte.
  4. Le service peut être redémarré indépendamment.
- **Mesure de la réponse** : L'impact sur les autres services doit être **nul** et la récupération doit prendre **moins de 2 minutes**.

Scénario 3 : Performance – Comparaison avec l'architecture monolithique

- **Source** : Architecte de l'équipe

- **Stimulus** : Besoin de valider que l'architecture microservices offre de meilleures performances
- **Artefact** : Scripts de test de performance
- **Environnement** : Environnement de test
- **Réponse** :
    1. Exécuter les tests de performance sur l'architecture monolithique.
    2. Exécuter les tests de performance sur l'architecture microservices.
    3. Comparer les résultats de latence, throughput et taux d'erreur.
- **Mesure de la réponse** : L'architecture microservices doit être **au moins 10% plus rapide** que l'architecture monolithique.

---

10. Risques et Dette Technique

10.1 Risques Techniques

- **RISK-001 : Latence de communication inter-services**
    - **Description** : Les appels HTTP entre microservices peuvent introduire une latence significative.
    - **Probabilité** : Élevée
    - **Impact** : Moyen (dégradation des performances utilisateur)
    - **Stratégie de mitigation** : Implémentation de cache, compression et connection pooling.
- **RISK-002 : Complexité de gestion des données distribuées**
    - **Description** : La cohérence des données entre services peut devenir complexe.
    - **Probabilité** : Moyenne
    - **Impact** : Élevé (incohérences métier)
    - **Stratégie de mitigation** : Utilisation de transactions distribuées et patterns de saga.
- **RISK-003 : Single point of failure de l'API Gateway**
    - **Description** : L'API Gateway constitue un point de défaillance unique.
    - **Probabilité** : Faible
    - **Impact** : Critique (arrêt complet du système)
    - **Stratégie de mitigation** : Déploiement de multiples instances avec load balancer.

10.2 Dette Technique

- **DEBT-001 : Absence de service mesh**
    - **Description** : Pas d'implémentation d'un service mesh (Istio, Linkerd) pour la gestion avancée du trafic.
    - **Urgence** : Faible
    - **Effort estimé** : 2 semaines
    - **Impact** : Complexité de gestion du trafic inter-services.

- **DEBT-002 : Monitoring limité**
  - **Description** : Pas de tracing distribué (Jaeger, Zipkin) pour suivre les requêtes à travers les services.
  - **Urgence** : Moyenne
  - **Effort estimé** : 1 semaine
  - **Impact** : Difficulté de debugging et d'optimisation.
- **DEBT-003 : Tests de charge insuffisants**
  - **Description** : Les tests de charge ne couvrent pas tous les scénarios de production.
  - **Urgence** : Moyenne
  - **Effort estimé** : 3 jours
  - **Impact** : Risque de problèmes de performance en production.

---

11. Glossaire

Termes Métiers

Terme	Signification
Microservice	Service indépendant qui implémente une fonctionnalité métier spécifique.
API Gateway	Point d'entrée unique qui route les requêtes vers les microservices appropriés.
Load Balancing	Distribution de charge entre plusieurs instances d'un même service.
Observabilité	Capacité à comprendre l'état interne d'un système à partir de ses sorties externes.
Service Mesh	Infrastructure de communication entre services avec gestion avancée du trafic.
Circuit Breaker	Pattern pour prévenir les cascades de pannes entre services.
Saga Pattern	Pattern pour gérer les transactions distribuées entre microservices.

Termes Techniques

Terme	Signification
Node.js	Runtime JavaScript côté serveur utilisé pour tous les microservices.
Express	Framework web minimaliste sur Node.js, utilisé pour l'API Gateway et les microservices.
Docker Compose	Outil d'orchestration de conteneurs Docker pour déployer l'ensemble des services.

Terme	Signification
Prometheus	Système de monitoring et d'alerting pour collecter les métriques des services.
Grafana	Plateforme de visualisation et d'analyse des métriques collectées par Prometheus.
http-proxy-middleware	Middleware Express pour le routage et le proxy dans l'API Gateway.
Round-robin	Algorithme de load balancing qui distribue les requêtes de manière cyclique entre les instances.
CORS	Cross-Origin Resource Sharing, mécanisme de sécurité pour les requêtes cross-origin.
Helmet	Middleware de sécurité pour Express qui définit des en-têtes HTTP sécurisés.
Sequelize	ORM JavaScript pour PostgreSQL, utilisé dans chaque microservice pour la persistance.
3-tiers	Architecture répartie en trois niveaux : client ↔ service ↔ base de données.
Microservices	Architecture où l'application est décomposée en services indépendants et communicants.

12. Conclusion

Ce document retrace l'évolution d'une architecture monolithique 3-tiers (Lab 2) vers une architecture microservices moderne (Lab 5), en passant par les tests de performance et monitoring (Lab 4). L'architecture finale offre une excellente scalabilité, résilience et observabilité, avec des performances supérieures à l'architecture monolithique (-14% de latence via l'API Gateway).

Les microservices permettent une évolution indépendante des services, le load balancing assure une haute disponibilité, et l'observabilité complète garantit une visibilité en temps réel sur l'état du système. Cette architecture constitue une base solide pour la croissance future du système de gestion multi-magasins.