Analyse Architecturale et Évaluation Technique du Système ERP

1. Introduction

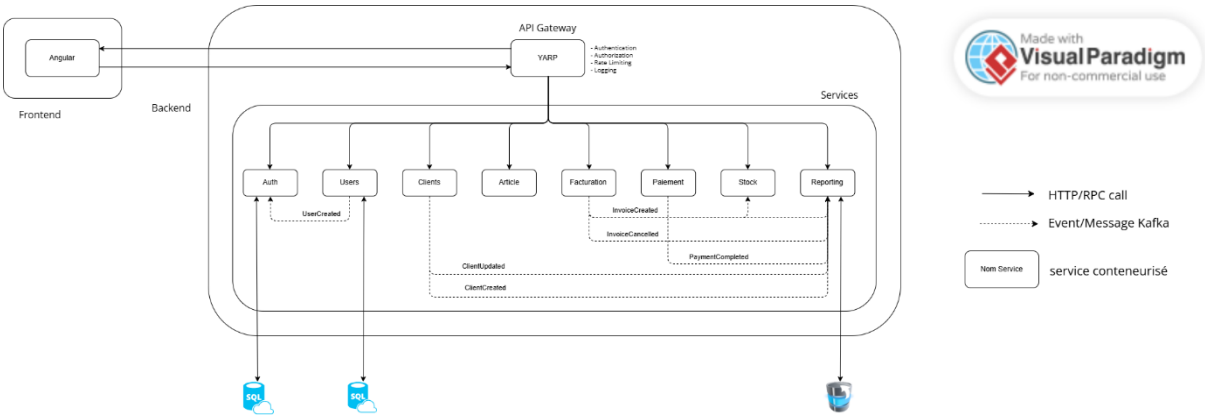
Ce document présente une analyse architecturale formelle du système ERP proposé, conçu selon une architecture microservices et orientée événements. Le système combine une communication synchrone via HTTP à travers un API Gateway et une communication asynchrone via Apache Kafka (mode KRaft).

L'objectif de cette architecture est d'assurer :

- La modularité
- La scalabilité
- La résilience
- La maintenabilité
- L'indépendance des domaines métier

---

2. Vue d'Ensemble de l'Architecture



Le système suit une structure modulaire composée de :

- Une couche Frontend (Angular)
- Un API Gateway (YARP)
- Des microservices indépendants

- Une communication asynchrone via Kafka
- Une base de données par service

Le modèle de communication est hybride :

- **Communication synchrone** : entre le client et les services via l'API Gateway
- **Communication asynchrone** : entre services via des événements Kafka

Cette approche permet un fort découplage entre les services tout en maintenant un point d'entrée centralisé.

---

3. Analyse par Couches

3.1 Couche Frontend

L'application Angular constitue l'interface utilisateur et communique exclusivement avec l'API Gateway.

Aucun accès direct aux microservices n'est autorisé.

Cette architecture garantit :

- Un contrôle centralisé de l'authentification
- Une meilleure sécurité
- Une abstraction des services internes
- Une simplification des configurations côté client

3.2 API Gateway (YARP)

L'API Gateway agit comme point d'entrée unique vers le backend.

Ses responsabilités incluent :

- Le routage des requêtes HTTP vers les services appropriés
- La validation des tokens JWT
- Le transfert des informations d'identité aux services backend
- La limitation du débit (rate limiting)
- La journalisation centralisée des requêtes
- La gestion du CORS

Cette centralisation évite la duplication des préoccupations transversales dans chaque service.

3.3 Couche Microservices

Chaque microservice :

- Est déployé dans son propre conteneur Docker
- Possède sa propre base de données
- Est indépendant et autonome
- Communique via HTTP ou Kafka

Les services principaux sont :

- Service Auth
- Service Users
- Service Clients
- Service Article
- Service Facturation
- Service Paiement
- Service Stock
- Service Reporting
- Service Log

Cette séparation respecte les principes du Domain-Driven Design (DDD).

---

4. Séparation des Domaines

4.1. Authentification et Gestion des Utilisateurs

Le service Auth est responsable de :

- L'authentification
- La génération des tokens JWT
- La gestion des rôles et permissions

Le service Users gère les informations de profil.

Il est recommandé de clarifier la frontière entre Auth et Users afin d'éviter toute ambiguïté fonctionnelle.

4.2. Services Facturation et Paiement

Le service Facturation produit les événements :

- InvoiceCreated
- InvoiceCancelled

Le service Paiement produit :

- PaymentCompleted

Cette séparation permet une gestion claire des responsabilités financières.

4.3. Service Stock

Le service Stock consomme les événements liés aux factures afin d'ajuster les niveaux d'inventaire.

Il ne communique pas directement avec la base de données du service Facturation, ce qui garantit un découplage fort.

4.4. Service Reporting

Le service Reporting consomme plusieurs événements métier afin de construire une base de données optimisée pour l'analyse.

Il ne réalise aucun accès direct aux bases des autres services.

Cela permet :

- Une indépendance complète
- Une architecture orientée événements
- Une scalabilité accrue

---

5. Communication Événementielle

5.1. Introduction

Apache Kafka (mode KRaft) agit comme bus d'événements central.

Chaque événement respecte une structure standardisée comprenant :

- Un identifiant unique
- Un type d'événement
- Une version
- Un horodatage
- Une charge utile (payload)

L'utilisation de topics par domaine (invoice-events, payment-events, etc.) assure une organisation claire.

Cette architecture garantit :

- Découplage des services
- Persistance des messages
- Tolérance aux pannes
- Possibilité de relecture des événements

5.2. Exemple de Scénario Événementiel

Lors de la création d'une facture :

1. Le service Facturation enregistre la facture dans sa base de données.
2. Il publie un événement InvoiceCreated dans le topic invoice-events.
3. Le service Stock consomme cet événement et met à jour les quantités disponibles.
4. Le service Reporting consomme également cet événement pour alimenter ses données analytiques.

Ce mécanisme illustre le découplage des services et la propagation asynchrone des changements d'état.

---

6. Stratégie de Gestion des Données

Le système applique strictement le principe :

Une base de données par service

Cela implique :

- Aucune base partagée
- Aucun accès SQL inter-services
- Communication exclusivement via API ou événements

Cette règle est essentielle pour préserver l'autonomie des services.

---

7. Risques Architecturaux et Stratégies d'Atténuation

7.1. Transactions Distribuées et Cohérence des Données

Risque
Dans une architecture microservices, chaque service possède sa propre base de données.
Il n'existe pas de transaction globale couvrant plusieurs services, ce qui peut entraîner des incohérences temporaires.

Stratégie mise en œuvre

Le système adopte un modèle de cohérence éventuelle.
Les services communiquent via des événements Kafka afin de propager les changements d'état.

Par exemple :

- Le service Facturation publie un événement InvoiceCreated
- Le service Stock consomme cet événement pour ajuster les quantités

Cette approche garantit un découplage fort entre les services tout en acceptant des incohérences temporaires contrôlées.

7.2. Supervision et Observabilité

Risque
La multiplication des services rend le suivi des erreurs plus complexe.

Stratégies mises en œuvre

- Journalisation par service via les logs .NET
- Visualisation des événements Kafka via Kafka UI
- Isolation des services via Docker Compose
- Analyse des logs via les conteneurs Docker

Dans un contexte académique, ces mécanismes sont suffisants pour assurer le suivi du système.

Perspectives d'amélioration

Dans un environnement de production, l'intégration d'outils tels que ELK, Prometheus ou OpenTelemetry permettrait d'améliorer l'observabilité et la traçabilité distribuée.

7.3. Gestion des Défaillances

Risque

- Indisponibilité temporaire d'un service
- Échec de traitement d'un message Kafka
- Problèmes réseau entre conteneurs

Stratégies mises en œuvre

- Isolation des services via Docker
- Redémarrage automatique des conteneurs via Docker Compose
- Gestion des exceptions côté consommateur Kafka
- Persistance des messages Kafka (broker KRaft)

Perspectives futures

- Implémentation du pattern Outbox pour fiabiliser la publication des événements
- Mise en place d'un Dead Letter Topic
- Orchestration via Kubernetes

---

8. Environnement de Déploiement

Le système est déployé localement via Docker Compose.

Chaque service est exécuté dans un conteneur indépendant incluant :

- Son runtime .NET
- Sa configuration
- Son accès réseau interne

Les composants suivants sont également conteneurisés :

- Apache Kafka (mode KRaft)
- Kafka UI
- Les bases de données

Docker Compose permet :

- L'orchestration simplifiée
- Le démarrage coordonné des services
- L'isolation réseau via un bridge interne

9. Scalabilité et Extensibilité

L'architecture permet :

- Partitionnement Kafka
- Déploiement indépendant
- Migration vers le cloud

L'architecture permet théoriquement une scalabilité horizontale par service, notamment dans un environnement orchestré (ex: Kubernetes).

Améliorations futures :

- Kubernetes
- Pattern Saga
- Schema Registry
- Support multi-tenant

---

10. Évaluation Globale

L'architecture démontre :

- Une forte adhérence aux principes microservices
- Une séparation claire des domaines
- Une intégration correcte orientée événements
- Une robustesse adaptée aux systèmes distribués modernes

L'architecture mise en œuvre démontre la compréhension et l'application des principes fondamentaux des architectures distribuées modernes, adaptés au contexte académique du projet.

---

11. Étude Comparative : Architecture Monolithique vs Architecture Microservices

11.1. Introduction

Le choix architectural constitue une décision stratégique dans la conception d'un système ERP. Deux approches principales existent :

- L'architecture monolithique
- L'architecture microservices

Cette section présente une comparaison technique et conceptuelle entre ces deux modèles afin de justifier le choix d'une architecture microservices pour le projet.

11.2. Architecture Monolithique

11.2.1. Définition

Une architecture monolithique regroupe l'ensemble des fonctionnalités d'une application dans une seule base de code et un seul processus exécutable.

Dans un ERP monolithique :

- Tous les modules (Clients, Facturation, Stock, Paiement, etc.) partagent :
    - La même application
    - La même base de données

- Le même cycle de déploiement

11.2.2. Caractéristiques

- Application unique
- Base de données partagée
- Déploiement unique
- Communication interne via appels directs (méthodes)

11.2.3. Avantages

- Simplicité initiale de développement
- Déploiement plus simple
- Moins de complexité DevOps
- Débogage plus direct
- Moins de latence interne

11.2.4. Inconvénients

- Forte dépendance entre modules
- Difficulté de mise à l'échelle partielle
- Risque élevé lors des mises à jour
- Couplage fort au niveau de la base de données
- Difficulté d'évolution à long terme

Dans un ERP, cela signifie que toute modification dans le module Facturation peut impacter le module Stock ou Paiement.

11.3. Architecture Microservices

11.3.1. Définition

Une architecture microservices divise l'application en plusieurs services indépendants, chacun responsable d'un domaine métier spécifique.

Chaque service :

- Possède sa propre base de données
- Est déployable indépendamment
- Communique via API ou événements

11.3.2. Caractéristiques

- Services autonomes
- Base de données par service
- Communication HTTP et événementielle
- Déploiement indépendant
- Isolation des domaines métier

11.3.3. Avantages

- Scalabilité horizontale par service
- Résilience accrue
- Déploiement indépendant
- Meilleure séparation des responsabilités
- Adapté aux architectures cloud et distribuées
- Favorise l'évolution progressive du système

Dans le projet ERP proposé :

- Facturation produit des événements
- Stock consomme ces événements
- Reporting construit des projections indépendantes

Cela garantit un découplage fort entre les domaines.

11.3.4. Inconvénients

- Complexité accrue
- Besoin d'infrastructure (Kafka, Docker, Gateway)
- Gestion des transactions distribuées

- Supervision plus complexe
- Cohérence éventuelle au lieu de cohérence immédiate

11.4. Comparaison Technique

Critère	Monolithe	Microservices
Structure	Application unique	Services multiples
Base de données	Partagée	Une par service
Déploiement	Global	Indépendant
Scalabilité	Globale	Par service
Couplage	Fort	Faible
Complexité initiale	Faible	Élevée
Résilience	Limitée	Élevée
Évolution long terme	Difficile	Flexible
Communication interne	Appels directs	HTTP + Événements
Adaptation cloud	Moyenne	Excellente

11.5. Justification du Choix des Microservices pour le Projet

Le choix d'une architecture microservices dans ce projet ERP est justifié par :

1. La nécessité d'isoler les domaines métier (Facturation, Stock, Paiement).
2. La volonté d'implémenter une architecture orientée événements.
3. L'objectif d'apprentissage des systèmes distribués modernes.
4. La préparation à une scalabilité future.
5. L'alignement avec les pratiques d'architecture d'entreprise contemporaines.

Bien que l'architecture microservices introduise une complexité supplémentaire, elle correspond davantage aux exigences d'un ERP moderne et extensible.

11.6. Conclusion Comparative

L'architecture monolithique convient aux projets de petite taille ou aux phases initiales de développement.

Cependant, pour un système ERP évolutif, modulaire et orienté cloud, l'architecture microservices offre :

- Une meilleure isolation des responsabilités
- Une plus grande résilience
- Une capacité d'évolution supérieure
- Une adaptation aux environnements distribués modernes

Le choix architectural adopté dans ce projet reflète ainsi une volonté d'alignement avec les standards actuels de l'ingénierie logicielle distribuée.

12. Limites du Projet

Dans le cadre académique et temporel du projet, certaines implémentations industrielles avancées (Outbox Pattern, orchestration Kubernetes, monitoring centralisé complet) n'ont pas été intégrées.

Cependant, l'architecture adoptée permettrait leur intégration dans une évolution future du système.

13. Conclusion

Le système ERP proposé intègre avec succès :

- Le pattern API Gateway
- Une architecture microservices
- Une communication événementielle via Kafka (KRaft)
- Une isolation stricte des données

En intégrant des mécanismes de cohérence, de supervision et de tolérance aux pannes, l'architecture répond aux exigences d'un système distribué d'entreprise moderne.