Rapport ARC42 Architecture d'un système de magasins multiples

Évolution Labs 0 à 2 – LOG430

Auteur : Chawki Benzeghiba
Date : 05 juillet 2025
Projet : Système de Gestion Multi-Magasins – Labo 0,1,2 LOG430

GitHub Lab0 : https://github.com/ChawkiBenzeghiba/Lab0-LOG430
GitHub Lab1 : https://github.com/ChawkiBenzeghiba/Lab1-LOG430
GitHub Lab2 : https://github.com/ChawkiBenzeghiba/Lab2-LOG430

---

1. Introduction et Objectifs

1.1 Contexte et portée

Ce document décrit l'évolution du système de caisse console (Lab 1) vers une API web multi-magasins (Lab 2), structurée de façon modulaire et empaquetée dans des conteneurs Docker pour un déploiement simplifié.

1.2 Objectifs architecturaux

Objectifs fonctionnels

- **Supervision multi-magasins** : Regrouper la gestion des ventes et des stocks de l'ensemble des boutiques.
- **Accès au stock central** : Offrir aux employés la possibilité de consulter le stock global et de soumettre des demandes de réapprovisionnement.
- **Rapports consolidés** : Fournir aux décideurs des tableaux de bord synthétiques sur les performances commerciales.
- **Gestion du catalogue** : Permettre à la maison-mère de créer, modifier et supprimer les produits disponibles.

Objectifs non-fonctionnels (attributs de qualité)

- **Maintenabilité** : Adopter une architecture MVC claire pour faciliter les évolutions et extensions.

- **Déployabilité** : Garantir un déploiement reproductible sur tout environnement via Docker Compose.
- **Scalabilité** : Concevoir le système pour accueillir de nouveaux magasins sans refonte majeure.
- **Performance** : Assurer des temps de réponse optimaux pour l'enregistrement des ventes et la génération de rapports.
- **Robustesse** : Isoler chaque composant pour que la défaillance d'un service n'impacte pas l'ensemble de l'application.

1.3 Parties prenantes

Rôle	Responsabilités	Enjeux principaux
Gestionnaire	Supervision globale, analyse des performances	Qualité et précision des rapports
Employé de magasin	Enregistrement des ventes, gestion des stocks	Rapidité et ergonomie de l'interface
Équipe de développement	Maintenance, tests et déploiement	Modularité et contrôle de la dette technique

---

2. Contraintes

2.1 Contraintes techniques

- **Plate-forme** : Node.js v18
- **Framework web** : Express
- **ORM** : Sequelize
- **Base de données** : PostgreSQL
- **Conteneurisation** : Docker & Docker Compose
- **Tests** : Jest (unitaires & intégration), Supertest
- **CI/CD** : GitHub Actions (build, tests, image Docker, push)

- **Déploiement** : appleboy/ssh-action pour mise à jour Docker Compose sur VM

2.2 Contraintes organisationnelles

- **Découpage en deux laboratoires** :
  - Lab 1 – application console 2-tiers
  - Lab 2 – API REST 3-tiers
- **Réutilisation** de la logique métier et des modèles `Produit` et `Vente` du Lab 1
- **Gestion centralisée** du code et de la documentation dans un dépôt Git unique
- **Documentation obligatoire** : README, arc42, UML et ADR dans `ADR/` et `UML/`
- **Livraison reproductible** : un seul `docker-compose up` suffit

2.3 Contraintes d'évolution (Lab 1 → Lab 2)

Éléments à conserver

- Calculs de ventes et agrégations métier
- Modèles de données de base : `Produit`, `Vente`
- Données d'amorçage (`seed.js`) pour tests

Éléments à modifier

- **Architecture** : passer de la console 2-tiers (app + BD) à une API REST 3-tiers (client HTTP + serveur Express + BD)
- **Modèle de stock** : remplacer le stock unique par deux entités `StockCentral` (global) et `StockLocal` (par magasin)
- **Persistance** : troquer les requêtes SQL manuelles pour des modèles Sequelize avec migrations/sync

Éléments à ajouter

- **Couche API** : routes Express et controllers pour UC1, UC2, UC3

- **Tests automatisés** : scénarios unitaires et d'intégration pour chaque endpoint
- **Déploiement** : script SSH + Docker Compose pour mise à jour sur VM distante

---

3. Contexte

3.1 Contexte Métier

- **Domaine** : Gestion de ventes et de stocks pour un réseau de magasins physiques.
- **Modèle** : Magasins autonomes reliés à une API centrale pour supervision et pilotage.
- **Enjeux** : Visibilité en temps réel des performances de chaque point de vente.

3.2 Évolution du Lab 1 au Lab 2

Lab 1 – Système de caisse console (2-tiers)

```
[ Application console ]  ⇄  [ PostgreSQL ]
       Tier 1                    Tier 2
```

- Architecture monolithique en 2 tiers (interface CLI + BD).
- Enregistrement des ventes et gestion locale du stock.
- Base de données initiale et scripts de "seed" pour les tests.

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

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

- Architecture découplée en 3 tiers (client, service, persistance).
- Endpoints pour :
  - UC1 : rapports consolidés des ventes
  - UC2 : consultation du stock central & réapprovisionnement
  - UC3 : tableau de bord des indicateurs
- Conteneurisation via Docker Compose et pipeline CI/CD pour déploiement automatisé.

---

4. Stratégie de Solution

4.1 Architecture cible

- **3-tiers** : Client HTTP ↔ API Express ↔ PostgreSQL
- **Pattern MVC Express**
- **Tests automatisés** : unitaires (Jest), intégration (Supertest)
- **CI/CD** : GitHub Actions (build → tests → lint → image Docker → push)
- **Données initiales** : `seed.js` (bulkCreate pour produits, magasins, ventes, stock)

4.2 Approche Domain-Driven Design (DDD)

- **Domaine Principal (Core Domain) : Gestion des Ventes**
  - **Description** : cœur métier, agrégation et synthèse des transactions
  - **Contexte délimité** : Ventes
  - **Langage Ubiquitaire** : Vente, prixUnitaire, quantite, Produit, Magasin
  - **Composants d'architecture** :
    - Modèles : `vente.js`,`produit.js`,`magasin.js`
    - Controller : `magasinController.js`: fonction - `enregistrerVente`
- **Domaine Support (Supporting Domain) : Stock & Réapprovisionnement**
  - **Description** : garantit la disponibilité produit via stock central et local
  - **Contexte délimité** : Gestion logistique
  - **Langage Ubiquitaire** : StockCentral, inventaire.
  - **Composants d'architecture** :
    - Modèles : `stockCentral.js`,`produit.js`,`magasin.js`
    - Controller : `stockController.js`: toutes les fonctions
- **Domaine Générique (Generic Domain) : Catalogue des Produits**
  - **Description** : gestion du référentiel produit

- **Contexte délimité** : Catalogue de produits

- **Langage Ubiquitaire** : Produit, id, nom, categorie, prix,
- **Composants d'architecture** :
  * Modèle : `produit.js`

  * Controller : `magasinController.js`: fonctions - `afficherProduits` - `rechercherProduits`

---

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 2.

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

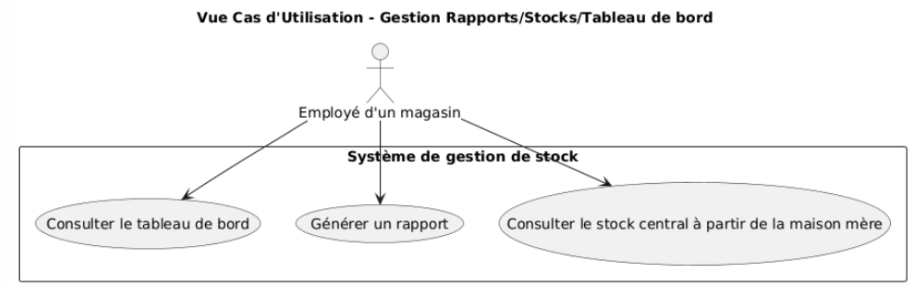

Figure 1: Cas d'utilisation

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

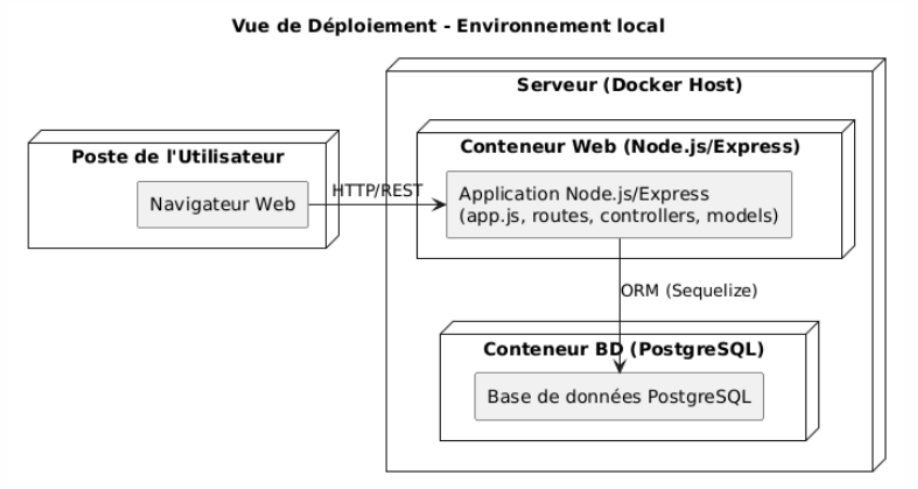

Figure 2: Déploiement

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 MVC.

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 modeles, les controlleurs, les vues ainsi que les relations entre elles.

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

5.5 Vue Processus

Voici les trois vues qui représentes les diagramme de séquences des cas d'utilisation réalisés dans le laboratoire 2.

Emplacement dans le projet: docs/UML/vue_processus_UC1_consulter_tableau_bord.puml

Emplacement dans le projet: docs/UML/docs/UML/vue_processus_UC2_consulter_rapport.puml

Emplacement dans le projet: docs/UML/vue_processus_UC3_consulter_le_stock_central.puml

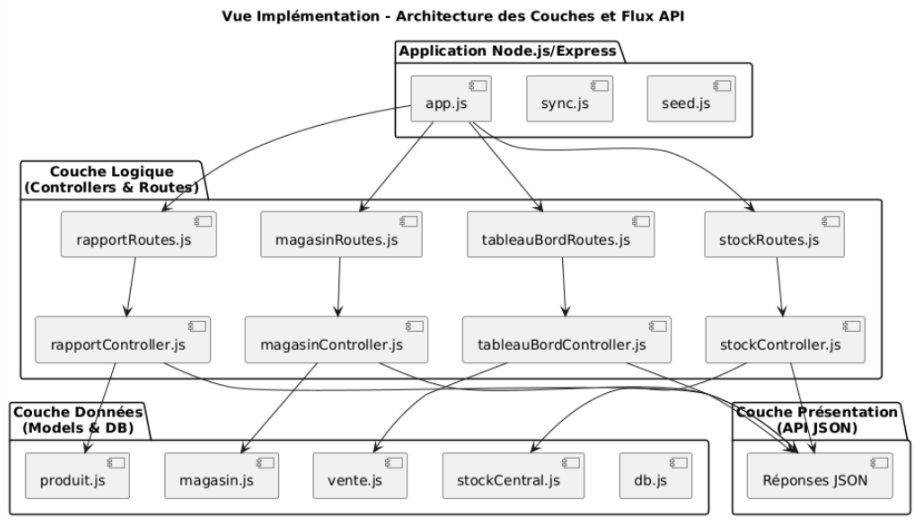

Figure 3: Implémentation

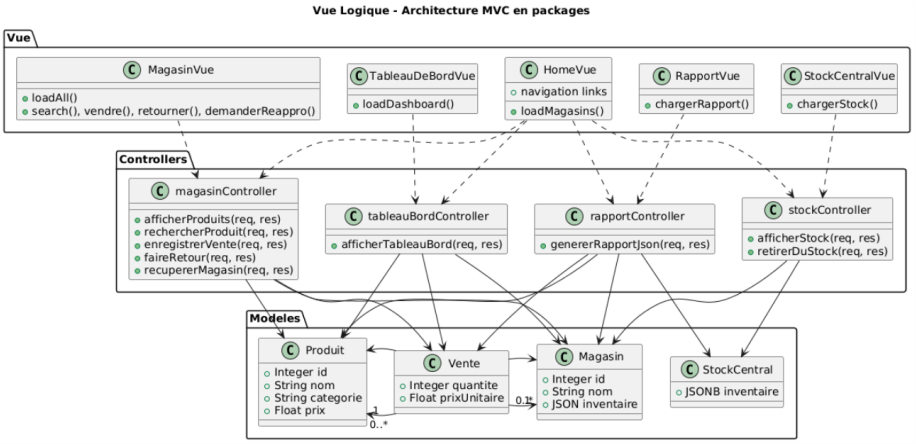

Figure 4: Logique

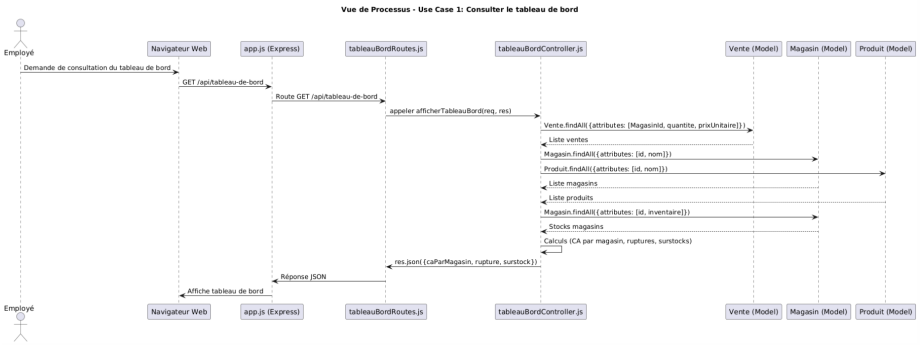

Figure 5: UC1

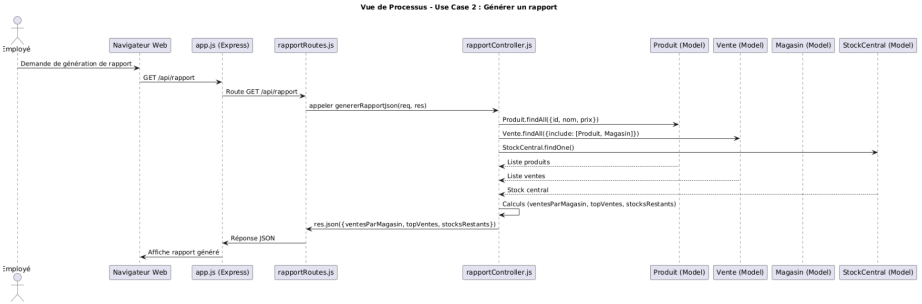

Figure 6: UC2

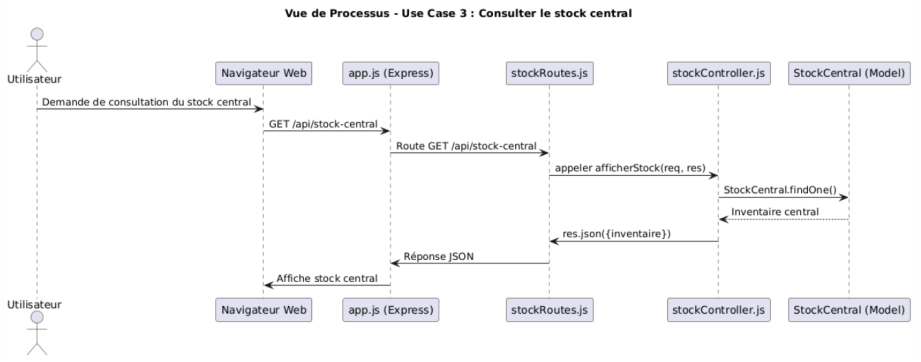

Figure 7: UC3

6. Concepts Transversaux

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

6.1 Sécurité

- **Validation des entrées**
  Chaque controller vérifie explicitement les champs du corps JSON (`produitId`, `quantite`, etc.) avant d'interagir avec la base.
- **Protection contre l'injection SQL**
  Sequelize utilise des requêtes paramétrées, éliminant les risques d'injection SQL.
- **CORS**
  Le middleware `cors()` est activé par défaut. À terme, il sera configuré pour restreindre les origines aux domaines autorisés.
- **Absence d'authentification/autorisation**
  Aucun mécanisme de contrôle d'accès n'est encore implémenté, ce qui constitue un point critique à traiter.

6.2 Performance

- **Pool de connexions**
  La configuration Sequelize inclut un pool de connexions (par défaut 5 connexions), réduisant la surcharge de connexion à PostgreSQL.
- **Requêtes optimisées**
  Les opérations de lecture intègrent les options `include` pour charger en une seule requête les entités liées (ex. joindre `Produit` lors du rapport).
- **Tests**
  Les tests unitaires s'exécutent en parallèle localement pour accélérer le feedback. En CI, on privilégie l'exécution séquentielle (`--runInBand`) pour limiter la consommation CPU.

6.3 Observabilité

- **Logging centralisé**
  Tous les événements importants (démarrage, synchronisation des modèles, erreurs critiques) sont envoyés vers la sortie standard et collectés par Docker.
- **Healthchecks et endpoints**
  PostgreSQL dispose d'un healthcheck défini dans Docker Compose. Une future amélioration pourrait être d'exposer un endpoint `/health` dans l'API pour surveiller son état.

- **Couverture de tests**
  Jest génère un rapport de couverture (`coverage/`) à chaque build. Ces données servent à vérifier que chaque domaine métier est suffisamment testé.
- **Monitoring minimal**
  Actuellement, on surveille l'état des conteneurs via les outils Docker natifs. Aucune solution APM n'est déployée pour l'instant.

7. Tests

- **Fichiers** :
  `UC1_generer_rapport.test.js`, `UC2_stock_central.test.js`, `UC3_tableau_bord.test.js`

- **Exécution** :
  `npm test` (→ `npx jest --ci --runInBand`)

- **Types** :
  unitaires (Jest) & intégration (Supertest)

- **Couverture** :
  `npx jest --coverage`

8. Architectural Decision Records (ADR)

ADR-001 — Architecture centralisée multi-magasins

Statut

Accepté

Contexte

L'évolution du système impose la gestion simultanée de plusieurs magasins, la centralisation des stocks et la génération de rapports consolidés pour la maison mère. Il faut garantir la cohérence des données, la consultation centralisée et la possibilité d'évolution vers d'autres interfaces (web, mobile).

Décision

Le système adopte une architecture centralisée : chaque magasin possède son propre inventaire, mais un stock centralisé (StockCentral) fait référence pour l'ensemble des magasins. Toutes les opérations critiques (consultation, réapprovisionnement, rapports, tableau

de bord) s'appuient sur cette base centralisée, accessible via des routes Node.js dédiées.

Raisons

- Permet une gestion cohérente et synchronisée des stocks entre magasins et maison mère.
- Simplifie la génération de rapports consolidés et la visualisation des performances.
- Facilite l'évolution vers d'autres interfaces (API, web, mobile) grâce à une logique centralisée.
- Réduit la complexité de synchronisation par rapport à une architecture totalement distribuée.

Conséquences

- Toute modification d'inventaire passe par la base centralisée, ce qui garantit la cohérence.
- Les performances dépendent de la disponibilité de la base centrale.
- L'architecture reste simple à maintenir et à faire évoluer pour les besoins futurs (ajout d'API, interface web, etc.).

ADR-002 — Gestion de la persistance et synchronisation des données

Statut

Accepté

Contexte

Le système doit gérer les inventaires de plusieurs magasins, un stock centralisé, les ventes et la génération de rapports consolidés. Il est essentiel d'assurer la cohérence des données et de permettre des opérations atomiques (réapprovisionnement, transfert de stock, etc.) tout en gardant la solution simple et évolutive.

Décision

Le projet utilise **Sequelize** comme ORM pour interagir avec une base de données relationnelle (PostgreSQL). Les inventaires des magasins et le stock central sont stockés sous forme de champs JSON/JSONB, ce qui permet de modéliser dynamiquement les quantités par produit.

Toutes les opérations critiques sont réalisées via des transactions Sequelize pour garantir la cohérence.

Raisons

- Sequelize simplifie la gestion des modèles, des migrations et des transactions.
- PostgreSQL offre robustesse, performance et support natif du type JSONB pour stocker des structures dynamiques.
- Les champs JSON/JSONB permettent de gérer facilement l'inventaire de chaque magasin sans multiplier les tables ou les jointures complexes.
- Les transactions Sequelize assurent l'intégrité lors des opérations critiques (réapprovisionnement, vente, etc.).

Conséquences

- La structure des inventaires est flexible et facilement extensible.
- Les opérations atomiques sont garanties par l'utilisation de transactions.
- La persistance centralisée facilite la génération de rapports et la synchronisation des données entre magasins et maison mère.
- L'évolution vers d'autres types de stockage (NoSQL, microservices) reste possible si les besoins changent à l'avenir.

9. Scénarios de Qualité

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

Scénario 1 : Maintenabilité – Ajout d'une nouvelle API "Détail Produit"

- **Source** : Développeur de l'équipe
- **Stimulus** : Le chef de projet demande un endpoint `GET /api/produits/:id` pour détailler un produit
- **Artefact** : Code source (`controllers`, `routes`, `models`)
- **Environnement** : Poste de dev local dans un conteneur Docker
- **Réponse** :

1. Ajouter la méthode `getProduitById` dans `produitController.js`.

2. Déclarer la route correspondante dans `produitRoutes.js` et `app.js`.

3. Écrire un test unitaire Jest et un test d'intégration Supertest.

- **Mesure de la réponse** : L'implémentation, les tests et la validation CI doivent prendre **moins de 2 heures**.

Scénario 2 : Déployabilité – Mise en place d'un nouvel environnement

- **Source** : Nouveau membre de l'équipe
- **Stimulus** : Besoin de lancer l'application complète sur sa machine
- **Artefact** : Répertoire Git du projet
- **Environnement** : Machine avec Docker, Node.js installés
- **Réponse** :
  1. Cloner le dépôt,
  2. Exécuter `docker-compose up --build`,
  3. Vérifier que la base est synchronisée (`sync.js`) et que `seed.js` charge les données,
  4. Tester un appel `GET /api/rapport` pour confirmer le bon fonctionnement.
- **Mesure de la réponse** : Le système doit être opérationnel en **moins de 10 minutes** après le clonage, sans configuration manuelle supplémentaire.

10. Risques et Dette Technique (complément)

10.1 Nouveaux Risques Techniques

- **RISK-001 : Incohérences en cas d'accès concurrent**
  - **Description** : Plusieurs requêtes simultanées de réapprovisionnement ou de vente peuvent entraîner des conflits de

mise à jour sur les mêmes enregistrements.

  - **Probabilité** : Moyenne

  - **Impact** : Critique (stocks erronés, ventes comptabilisées en doublon)

  - **Stratégie de mitigation** : Utiliser les transactions Sequelize avec verrouillage (`transaction.lock`) ou adopter une file d'attente (RabbitMQ/Kafka) pour sérialiser les opérations.
- **RISK-002 : Saturation du pool de connexions**
  - **Description** : En cas de pic d'activité (nombre élevé de requêtes simultanées), le pool par défaut (5 connexions) peut être saturé, provoquant des erreurs 5xx.

  - **Probabilité** : Moyenne

  - **Impact** : Moyen à élevé (dénis de service partiel)

  - **Stratégie de mitigation** : Ajuster la taille du pool (`pool.max` dans Sequelize), surveiller l'utilisation et mettre en place un circuit breaker (opossum).

10.2 Nouvelles Dettes Techniques

- **DEBT-001 : Manque de migrations versionnées**
  - **Description** : On utilise `sequelize.sync({ alter: true })` sans migrations formelles, ce qui peut écraser ou corrompre la structure en prod.

  - **Urgence** : Moyenne

  - **Effort estimé** : 2 jours-homme
  - **Impact** : Risque de perte de données et difficultés lors du déploiement d'évolution de schéma.
- **DEBT-002 : Logs non structurés**
  - **Description** : Les `console.log` simplistes ne fournissent pas de niveau (info, warn, error) ni de format JSON exploitable par un agrégateur de logs.

  - **Urgence** : Faible

  - **Effort estimé** : 1 jour-homme
  - **Impact** : Difficulté à tracer et corréler les événements en

production.

11. Glossaire

Termes Métiers

Terme	Signification
Magasin	Point de vente physique disposant de son propre inventaire local.
Maison-Mère	Entité centrale qui administre le catalogue produit et supervise l'ensemble des magasins.
Produit	Article référencé dans le catalogue, avec ses attributs (nom, catégorie, prix).
Vente	Transaction enregistrée lorsqu'un client achète un ou plusieurs produits dans un magasin.
Réapprovisionnement	Processus de transfert de stock du dépôt central vers un magasin à la demande de celui-ci.
Stock Central	Inventaire global de tous les produits, géré par la maison-mère.
Stock Local	Quantités de produits disponibles dans un magasin spécifique, dérivées du stock central.
Inventaire	Ensemble des quantités disponibles pour chaque produit, soit au niveau central, soit local.
Rapport consolidé	Document agrégé regroupant les ventes de tous les magasins (UC1).
Tableau de bord	Interface ou réponse JSON présentant les indicateurs clés (CA, ruptures, surstocks) (UC3).
UC (Use Case)	Cas d'utilisation métier (ex : UC1 – génération du rapport consolidé des ventes).

Termes Techniques

Terme	Signification
Node.js	Runtime JavaScript côté serveur utilisé pour exécuter l'API.
Express	Framework web minimaliste sur Node.js, gérant routes et middlewares.
Sequelize	ORM JavaScript pour PostgreSQL, gérant modèles, migrations et transactions.
Docker Compose	Outil d'orchestration de conteneurs Docker via un fichier `docker-compose.yml`.
Jest	Framework de tests unitaires en JavaScript, utilisé pour la logique métier et la persistance.

Terme	Signification
Supertest	Bibliothèque de tests d'intégration pour API Express, simulant des requêtes HTTP.
API REST	Interface de service web suivant les principes REST (endpoints, verbes HTTP, JSON).
CRUD	Acronyme de Create / Read / Update / Delete : opérations fondamentales de gestion des données.
JSON	Format d'échange de données structuré (utilisé pour les requêtes et réponses HTTP).
Healthcheck	Mécanisme automatique de vérification de l'état d'un service (ex : `pg_isready` pour PostgreSQL).
ADR	*Architectural Decision Record* : document formalisant un choix d'architecture.
CI/CD	*Continuous Integration / Continuous Deployment* : pipelines automatisés de tests et de déploiement.
Conteneurisation	Embarquement de l'API et de la base dans des conteneurs Docker pour portabilité et isolation.
DDD	*Domain-Driven Design* : approche de conception centrée sur le domaine métier et ses concepts.
MVC	*Model-View-Controller* : séparation du code en modèles, contrôleurs et routes/vues.
ORM	*Object-Relational Mapping* : abstraction des tables PostgreSQL via Sequelize en objets JavaScript.
3-Tiers	Architecture répartie en trois niveaux : client HTTP ↔ service API ↔ base de données.

12. Conclusion

En somme, ce document retrace comment nous sommes passés d'une application console monolithique à une API REST moderne, modulaire et entièrement conteneurisée. Grâce à une architecture en 3-tiers, à un découpage clair en MVC, à une approche DDD et à un pipeline CI/CD accompagné de tests automatisés.