

1. Introduction et buts

Le but de ce laboratoire est de concevoir un système de gestion de magasin.

Durant le laboratoire 1, une interface graphique simple a été conçu pour permettre l'utilisation des fonctionnalités minimales implémentées. Le but du laboratoire 2 précisément est d'améliorer ce qui a été fait dans le laboratoire un, en plus d'implémenter de nouveaux requis.

L'architecture monolithique 2-tiers de ce projet doit être amélioré vers un système distribué et scalable.

Les objectifs clés incluent :

- La cohérence des données entre les entités
- Génération de rapports centralisés pour le siège social
- Évolutivité vers une interface web

1.1 Aperçu des exigences

Priorisation MoSCoW

Must

- UC1 – Générer un rapport consolidée des ventes
- UC2 – Consulter le stock central et déclencher un réapprovisionnement
- UC3 – Visualiser les performances des magasins dans un tableau de bord

Should (Ne sont pas implémentés)

- UC4 – Mettre à jour les produits depuis la maison mère
- UC6 – Approvisionner un magasin depuis le centre logistique

Could (Ne sont pas implémentés)

- UC7 – Alerter automatiquement la maison mère en cas de rupture critique
- UC8 – Offrir une interface web minimale pour les gestionnaires

1.2 Objectifs de Qualité

Top 3 Qualités

1. Cohérence des données :

- Synchronisation des données au travers de la chaîne de magasins
- Mécanisme de résolution de conflits

2. Évolutivité :

- Ajout d'un magasin en <1 heure
- Support pour extensions futures

3. Transférabilité :

- Les interfaces ne doivent pas pas dépendre du système d'exploitation pour être mises en marche

1.3 Parties Prenantes

Rôle	Préoccupations Majeures
Hauts dirigeants	Exactitude des rapports financiers
Employés de magasin	Simplicité d'interface
Équipe DevOps	Facilité de déploiement
Gestionnaires	Utilité du centre logistique

Attentes Techniques

- Documentation ADR complète
- Tests automatisés >80% de couverture
- CI/CD avec validation

2. Contraintes Architecturales

- **VM imposée** : L'infrastructure doit fonctionner sur la machine virtuelle fournie
- **Docker obligatoire** : Conteneurisation requise pour tous les composants
- **Modèle 4+1 UML** : Production des vues architecturales exigée
- **ADR requis** : Documentation des décisions architecturales obligatoire
- **CI/CD imposée** : Pipeline GitLab existante à utiliser
- **Budget limité** : Solutions open-source uniquement, pas de services cloud payants
- **Conventions de code** :
 - Français pour le domaine métier
 - Anglais pour le code technique
 - JavaDoc obligatoire

3. Contexte et Périmètre

3.1 Contexte Métier


Système Central:

- Gestion des stocks multi-magasins
- Génération de rapports consolidés
- Tableaux de bord de performance

Partenaires Métier:

Acteur	Interactions
--------	--------------

Acteur	Interactions
Employés de magasin	Demandes de réapprovisionnement
Employés de magasin	Saisie des ventes
Employés de magasin	Recherch de produits
Siège Social	Consultation des rapports

Format/Protocole : Interface graphique web  Usecases

3.2 Contexte Technique

Stock:

- Synchronisation via API Rest
- Base de données SQLite centrale

Contraintes d'Intégration:

- Doit coexister avec l'infrastructure CI/CD implémentée durant le lab 1
- Doit le système doit rouler sur un docker container

 ContexteTechnique

4. Stratégie de Solution

Décisions Fondamentales

1. Architecture Controller + Services + DAO

- *Pourquoi* : Découple les différents éléments du système pour les rendre indépendents des un des autres et donc facile à refactorer
- *Implémentation* :

2. Synchronisation par Événements

- *Pourquoi* : Atteindre la cohérence des données entre magasins
- *Avantage* : Nécessite seulement de mettre à jour les informations sur la BD, pas d'avertir les autres éléments du système.

3. Découpage Microservices

- *Services* :
 - Gestion-Stock (Spring Boot)
 - Reporting (Quarkus pour performance)
- *Avantage* : Évolutivité indépendante

Qualités Clés Adressées

Qualité	Scénario	Solution
Cohérence	Sync stocks instantané	Toutes transactions reflétées directement sur la BD
Évolutivité	Ajout magasin en moins d'un jour	API versionnée + contrat Swagger

Choix Technologiques

- **Backend :**
 - Python, API sur serveur Flask
 - Base de données : SQLite
- **Frontend :**
 - LitElements (découplée)
- **DevOps :**
 - CI/CD : GitLab Pipelines

5. Vue des Blocs de Construction

5.1 Vue Niveau 1 (Système Global)



Composants principaux :

Bloc	Responsabilités	Interfaces
Site Web	Point d'accès, Communiquer avec l'API	javascript fetch
Serveur Flask	Gestion de la logique système	API REST
Base de données	Persistence des données	SQLite

Relations :

- Tout communication externe au Docker est gérée par le serveur Flask avant d'accéder à la base de données
- L'interface orchestre les interactions utilisateur et la communication au serveur

5.2 Vue Niveau 2 (Détail Docker container)



Structure interne :

1. **Serveur Flask :**
 - Expose la logique métier (Recherche de produits, Gestion de commandes)
 - Contrôle les transactions

- API sur le port 8080

2. Base de données SQLite et ORM :

- Abstraction de l'accès aux données
- Gère le mapping objet-relationnel

Principe clé :

Séparation stricte entre logique métier (Service) et persistance (ORM).

6. Vue de runtime

UC1 et UC3 - Génération et visualisation de rapport



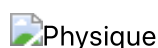
UC2 – Consulter le stock central et déclencher un réapprovisionnement



Sauvegarder une commande



7. Vue de déploiement



Node

Node	Description
Caisse	Poste utilisé par le caissier pour interagir avec le système via l'interface Web.
Ordinateur	Utilisé par les gestionnaires et dirigeants pour accéder aux fonctions de gestion et de consultation.
Site Web	Interface Web client accessible via HTTPS. Sert de point d'entrée principal.
Docker	Conteneur Docker déployé sur un serveur distant. Héberge les composants de backend.
Serveur Flask	Roule à partir du Docker container. Fournit une API au backend.
API	Fournit une interface REST entre le site Web et la logique métier.
Contrôleur	Gère la logique de traitement des requêtes.
Services	Contient la logique métier, appelle la base de données via un ORM.
Base de données	Persistance des données métier du système.

Les terminaux clients (caisses et ordinateurs) communiquent avec le site web via HTTPS. Le site Web appelle ensuite l'API REST exposée par le backend.

Prérequis

- **Client** : Navigateur Web moderne, connectivité Internet via HTTPS.
- **Serveur** : Environnement Docker avec les services applicatifs, base de données relationnelle, et connectivité réseau sécurisée.

8. Concepts transversaux

Cette section décrit les concepts transversaux appliqués dans tout le système. Ils garantissent la cohérence, la maintenabilité et la qualité interne de l'architecture. Ces concepts influencent plusieurs blocs fonctionnels et techniques.

8.1 Sécurité

Le système applique une séparation stricte des rôles (caissier, gestionnaire, dirigeant). Toutes les communications client-serveur sont protégées par HTTPS. Un mécanisme d'authentification est requis pour accéder aux interfaces sensibles, et les actions sont contrôlées selon les rôles attribués aux utilisateurs.

8.2 Architecture REST uniforme

Toutes les interactions entre le site Web et le backend utilisent des API REST. Les conventions suivantes sont respectées :

- URLs structurées par ressource (ex. `/products`, `/orders/report`)
- Utilisation cohérente des méthodes HTTP (GET, POST, PUT, DELETE)
- Réponses en JSON, avec codes d'état HTTP standardisés pour indiquer les succès et les erreurs.

8.3 Validation des données

Les entrées des utilisateurs (depuis la caisse ou l'interface de gestion) sont systématiquement validées avant traitement :

- Au niveau de l'API : vérification des formats (ex. types, champs obligatoires)
- Au niveau métier : règles spécifiques (ex. stock suffisant avant une vente) Cette approche renforce la robustesse du système et évite la propagation d'erreurs jusqu'à la base de données.

8.4 Conteneurisation avec Docker

Le backend est entièrement déployé dans un conteneur Docker, ce qui permet une homogénéité de l'environnement entre développement, tests et production. Tous les composants (API, services, base de données) sont configurés pour fonctionner ensemble dans un environnement isolé et reproductible.

9. Décision d'architecture

ADR 1 Choix de Flask pour l'API

Statut

Implémenté

Contexte

Lors du laboratoire 1, la communication entre l'interface utilisateur et la base de données était limitée et directe, avec peu de logique métier côté serveur. Avec l'évolution vers le laboratoire 2, le système doit gérer davantage de règles métier, de validations et d'interactions complexes entre les entités (commandes, produits, magasins, utilisateurs).

Il fallait un moyen simple, flexible et rapide à mettre en œuvre pour exposer ces fonctionnalités sous forme d'API REST, tout en gardant la maintenance facile et la possibilité d'évolution.

Décision

Nous avons décidé d'utiliser Flask comme framework backend pour l'API. Flask est léger, simple à prendre en main, compatible avec SQLAlchemy (notre ORM), et permet de construire des routes REST rapidement et efficacement.

Cela nous offre un contrôle granulaire sur les requêtes, les réponses, la gestion des erreurs et la sécurité, tout en gardant la possibilité d'étendre ou d'ajouter des fonctionnalités facilement.

Conséquences

- L'API est découplée de la couche frontend, facilitant les développements parallèles et la maintenance.
- Le backend peut évoluer indépendamment, par exemple en ajoutant des authentifications, middlewares, ou en intégrant d'autres services.
- Flask, grâce à sa simplicité, réduit la complexité de mise en place initiale, ce qui accélère le développement.
- Le choix impose de gérer manuellement certains aspects (gestion des erreurs, sécurité) mais offre aussi plus de flexibilité.
- Le projet nécessite un serveur dédié pour héberger cette API, ce qui est pris en compte dans le déploiement via Docker.

ADR 2 Choix de SQLite pour la persistance des données

Statut

Implémenté

Contexte

Pour le laboratoire 1, il fallait une solution simple et légère pour stocker les données localement pendant le développement initial. Les exigences n'étaient pas encore très complexes et le système devait rester facile à configurer et à déployer.

Une base de données relationnelle intégrée, sans serveur à gérer, était idéale pour ce contexte.

Décision

Nous avons choisi d'utiliser SQLite comme solution de persistance. SQLite est une base de données relationnelle embarquée, qui stocke les données dans un simple fichier local.

Ce choix facilite le déploiement, évite la complexité d'un serveur de base de données et offre un bon support SQL pour les besoins actuels.

Conséquences

- Le système est facile à configurer et à démarrer, notamment pour les tests et le développement local.
- La gestion des données reste performante tant que la charge est modérée et les données de taille raisonnable.
- Ce choix limite la scalabilité et la gestion de connexions concurrentes par rapport à un serveur SQL dédié (ex : MySQL, PostgreSQL).
- Lors du passage à un contexte plus complexe (lab 2), un changement vers une base plus robuste pourra être envisagé.
- Le fichier de base de données peut être facilement sauvegardé, copié ou déplacé.

10. Exigences de qualité

Cette section rassemble les exigences de qualité applicables au système. Les exigences critiques sont détaillées à la section [1.2 Objectifs de Qualité](#) et sont simplement référencées ici.

Les scénarios ci-dessous décrivent des exigences de qualité supplémentaires, jugées utiles mais non critiques. Leur atteinte améliore la robustesse, la maintenabilité ou l'expérience utilisateur, sans être bloquante en cas d'échec partiel.

10.1 Référence aux objectifs de qualité principaux

Voir section [1.2 Objectifs de Qualité](#) pour les trois qualités prioritaires :

- **Cohérence des données**
 - Synchronisation des stocks entre magasins
 - Résolution des conflits entre écritures concurrentes
- **Évolutivité**
 - Intégration d'un nouveau magasin en moins d'une heure
 - Support anticipé pour nouvelles fonctionnalités
- **Transférabilité**
 - Interfaces compatibles multiplateformes
 - Déploiement standardisé via Docker

10.2 Scénarios de qualité secondaires

ID	Description
10.2.1	Le site Web doit rester utilisable sur une connexion mobile ou instable (latence élevée, débit réduit).
10.2.2	Toute opération de vente doit être confirmée à l'écran en moins de 2 secondes.

ID	Description
10.2.3	L'ajout d'une fonctionnalité (ex. gestion des retours) ne doit nécessiter aucune modification du service Stock .
10.2.4	La base de données doit supporter 10 utilisateurs simultanés sans ralentissement perceptible.
10.2.5	Le code backend doit atteindre 80 % de couverture par des tests unitaires automatisés.
10.2.6	Le système doit fonctionner sans dépendre d'un service externe tiers (hors infrastructure fournie).
10.2.7	Une modification mineure dans la configuration Docker ne doit pas invalider le déploiement (robustesse à l'infrastructure).

11. Risques et Dette Technique

11.1 Risques Techniques

Priorité	Risque	Impact	Solution Proposée	Statut
1	Monolithisation progressive	Réduction de la scalabilité	Découper en microservices dès que le budget le permet	Surveillé
2	Performance des requêtes SQL	Ralentissement avec l'ajout de magasins	Ajout d'index et optimisation des requêtes	En cours
3	Dépendance à la VM imposée	Difficulté de migration cloud	Containerisation complète avec Docker	Mitigé

11.2 Dette Technique

Composant	Dette	Conséquence	Plan de Remédiation
Interface terminal	Code legacy du Lab 1	Maintenance difficile	Migration vers Vue.js
Synchronisation	Approche polling plutôt qu'events	Latence élevée	Implémentation RabbitMQ
Tests	Couverture à 65%	Risque de régression	Ajout de tests d'intégration

11.3 Risques Métier

Probabilité	Risque	Impact	Atténuation
Élevée	Changement des besoins magasins	Refonte partielle	Architecture modulaire

Probabilité	Risque	Impact	Atténuation
Moyenne	Concurrence solutions SaaS	Obsolescence	Focus sur intégration locale
Faible	Évolution réglementaire	Mises à jour urgentes	Veille active

Stratégie de gestion :

- Revue trimestrielle des risques
- Budget dédié pour la dette critique
- Priorisation via matrice impact/coût

12. Glossaire

Termes Métiers

Terme	Définition
Magasin	Point de vente physique géré par le système
Siège Social	Centre administratif supervisant l'ensemble des magasins
Réapprovisionnement	Processus de commande de nouveaux stocks depuis le centre logistique
UC (Cas d'Usage)	Scénario fonctionnel clé du système (ex: UC1 = Rapport consolidé)
KPI	Indicateur de performance suivi dans les tableaux de bord

Termes Techniques

Terme	Définition
VM	Machine virtuelle hébergeant l'application
CI/CD	Pipeline d'intégration/déploiement continu (GitLab)
JWT	JSON Web Token utilisé pour l'authentification
ORM	Couche de mapping objet-relationnel
WebSockets	Protocole de communication temps réel entre magasins
ADR	Architectural Decision Record - Document de décision architecturale

Acronymes

Acronyme	Signification
API	Application Programming Interface
DTO	Data Transfer Object

Acronyme	Signification
DDD	Domain-Driven Design
MVC	Modèle-Vue-Contrôleur
ACID	Propriétés des transactions SQL (Atomicité, Cohérence, Isolation, Durabilité)

Technologies

Technologie	Usage
MySQL	Base de données relationnelle principale
Python	Langage backend principal
Vue.js	Framework pour l'interface web
Docker	Plateforme de conteneurisation
RabbitMQ	Système de messagerie pour les événements
GitLab CI	Pipeline d'intégration continue

Composants Système

Composant	Description
Service Stock	Module central de gestion des inventaires
DAO	Couche d'accès aux données
Tableau de bord	Interface de visualisation des KPI
API REST	Interface de programmation pour les intégrations
Conteneur Docker	Unité de déploiement isolée


12. Diagrammes restants

BlockViewLevel1

BlockViewLevel2

ContexteTechnique


Logique


Physique

Process

SequenceRestock

SequenceSaveOrder

 SequenceSeeStats

 Usecases