École de technologie supérieure

Labo 4 – Load Balancing, Caching, Test de charge et Observabilité

Rapport de l'architecture logicielle

Cours	LOG430	
Session	E-2025	
Groupe	01	
Professeur	Fabio Petrillo	
Étudiant(s)	Maksym Pravdin	PRAM86290201
Date	18 juin 2025	

Table des matières

1. Introduction et objectifs

Lien vers le repo GitHub : https://github.com/PraMaks/LOG430_Labo_0

Lien vers les conteneurs Docker sur DockerHub :
https://hub.docker.com/repositories/pramaks

Rapport rédigé en suivant le gabarit de ARC42 : https://arc42.org/overview

Dans le cadre du cours LOG430, il faut développer une application de gestion de magasins en appliquant les notions vues durant le cours. L'architecture du système évoluera au cours des laboratoires.

Pour le « Labo 4 », des nouveaux objectifs sont ajoutés au système :

- Exécuter un test de charge réaliste sur l'application.
- Observer les 4 Golden Signals (latence, trafic, erreurs, saturation).
- Ajouter des logs structurés et des métriques applicatives.
- Mettre en œuvre un répartiteur de charge.
- Implémenter un cache pour optimiser les endpoints critiques.
- Analyser et comparer les performances avant/après les optimisations

Ainsi, ces exigences sont ajoutées aux besoins fonctionnelles et non fonctionnelles déjà existants.

Besoins fonctionnels :

- Le système doit avoir une gestion des produits
- Rechercher un produit par son nom
- Consulter l'état du stock du magasin

Le système doit avoir une gestion des ventes :

- Enregistrer une vente :
  - sélection de plusieurs produits
  - calcul automatique du total
  - réduction du stock à la suite de la vente
- Annuler une vente (gestion des retours) :
  - remettre des produits dans l'inventaire du magasin
- Le système doit avoir la persistance de données
  - Toutes les données doivent être stockées de façon persistante dans une base de données (MongoDB)

- Utilisation d'une couche d’abstraction de persistance (ORM ou ODM) pour interagir avec la base de données
- Le système doit avoir le support "multicaisse" pour supporter plusieurs utilisateurs
- Le système doit supporter les interactions de plusieurs utilisateurs connectés simultanément à la base de données
- Les opérations de vente du système doivent être transactionnelles/enregistrées pour garantir la cohérence des données

Besoins non fonctionnels :

- Le système doit pouvoir supporter plusieurs transactions simultanées (ex. : 3 caisses)
- Le système doit avoir la suppression ou l'annulation d'une vente qui ne doit pas corrompre l'état de l'inventaire
- Le code doit être clair, bien structuré et modulaire (séparation de la logique métier, de la présentation et de la persistance)
- Le système doit respecter les bonnes pratiques de développement (pipeline CI/CD, tests unitaires avec Pytest)
- Le système doit être exécutable via Docker (Dockerfile + docker-compose.yml)
- Le système doit être conçu de manière évolutive pour pouvoir être réutilisée vers les futures versions (dans les futurs laboratoires)

Exigences du labo 3 :

- Extension de l’architecture avec une couche d’API REST (et avoir des bonnes pratiques)
- Documentation des API avec Swagger
- Sécurité et accessibilité avec CORS et un service d’authentification
- Tests et validation pour les endpoints
- Avoir un déploiement fonctionnel avec Docker

Historique des versions :

- Labo 0 : **Architecture « Mainframe »** - un seul fichier Python était utilisé pour faire toutes les opérations : additionner 2 chiffres ensemble. Ce laboratoire était plus introductif pour se familiariser avec Docker et avec GitHub Actions

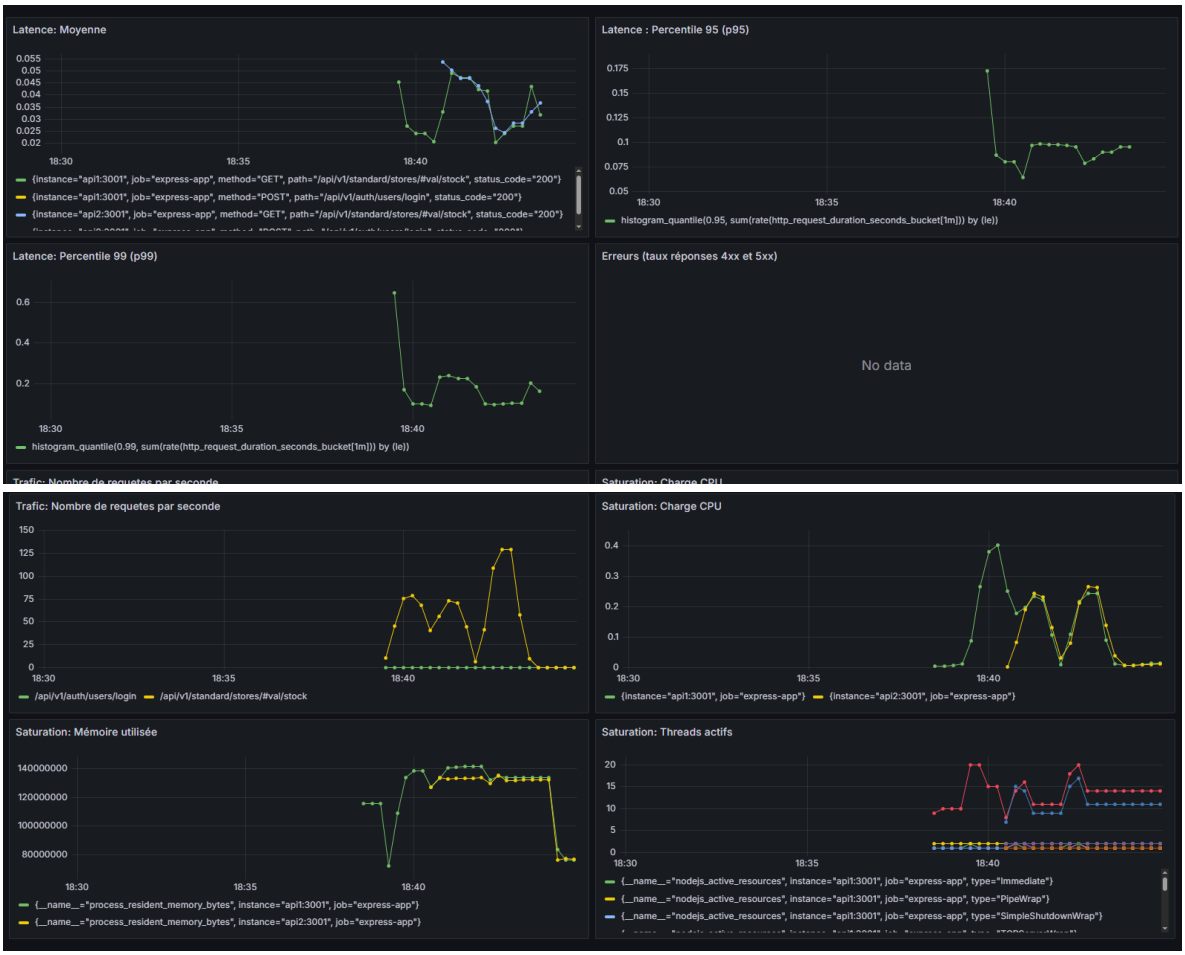
pour faire un pipeline CI/CD (4 étapes : Lint – Tests unitaires – création de l'image Docker – publication de cette image sur Docker Hub)

- Labo 1 : **Architecture « 2-tier »** - une base de données NoSQL, MongoDB est ajouté à l'architecture. Le système commence à avoir les premières fonctionnalités pour la gestion de l'inventaire d'un magasin à partir de 3 caisses : rechercher un produit, enregistrer une vente, faire un retour de vente, consulter l'inventaire du magasin. Pour assurer la persistance et la synchronisation des données, un ORM est utilisé qui s'appelle « mongoengine » et qui compatible avec le langage Python.
    - Labo 2 : **Architecture « 3-tier » et Domain Driven** – un framework backend Express.js, est ajouté au système à la suite de la recommandation par un chargé de laboratoire au lieu de FastAPI. Ainsi, une grande partie du code Python de la version précédente a dû être migrée en JavaScript. À partir de cette version, le système doit être en mesure de gérer l'inventaire de 5 magasins, d'un magasin mère et du centre de stockage central. En plus des fonctionnalités existantes, des nouvelles fonctionnalités sont ajoutés. Les magasins peuvent consulter l'état de stock du magasin mère et faire une demande de réapprovisionnement. Le magasin mère a accès à toutes les fonctionnalités que les magasins standards ont. De plus, le magasin mère peut générer un rapport consolidé des ventes, visualiser les performances des magasins et mettre à jour les données d'un produit à travers tous les magasins. Une version initiale d'un API REST est mise en place sur Express.js pour gérer les communications http entre la console Python (frontend) et Express.js (backend). Puisque Python ne communique plus directement avec la base de données, « mongoengine » est remplacé par « mongoose » une alternative pour JavaScript. Les exigences suivantes étaient ajoutées :
        - Permettre la gestion simultanée et cohérente de plusieurs magasins.
        - Offrir une consultation centralisée des stocks disponibles et des transactions réalisées par les magasins.
        - Assurer la synchronisation fiable et cohérente des données entre les différents magasins, le magasin mère et le centre de stockage.
        - Rapports consolidés générés pour l'administration depuis la maison mère.
        - Évolutivité vers une potentielle interface web ou mobile.

- Labo 3 : **Architecture « 3-tier » Domain Driven avec REST API + frontend Django (backend monolithique) –** un framework frontend Django est ajouté pour remplacer la console Python. Cette décision permet aux utilisateurs d'accéder au système via un browser ou un appareil mobile. Pour assurer que les pages soient responsives, une extension de bootstrap5 est ajoutée au framework Django. Ceci permet aux pages de s'adapter à la résolution de la fenêtre de l'utilisateur. De plus, un service d'authentification est ajouté à l'architecture. Une collection des utilisateurs est ajoutée à la base de données. Les mots de passe des utilisateurs sont chiffrés avec l'extension bcrypt et donc ces mots de passes apparaissent chiffrés dans la base de données. Une page de connexion nécessitant un nom d'utilisateur et un mot de passe sont nécessaires pour se connecter. Une fois l'appel API du frontend est envoyée au backend, le mot de passe reçu en payload est comparé avec bcrypt avec celui qui est chiffré dans la base de données. Si c'est le cas, un token est généré et ajouté au cache d'express.js. Toutes les autres routes API nécessitent ce token pour être disponible à l'utilisateur. Django utilise l'objet session dans la base de données sqlite de django pour stocker ce token. Ce token est supprimé lorsque l'utilisateur se déconnecte ainsi le token est supprimé de l'objet session et de la cache d'express.js. Des tests backend sont ajoutés pour tester les routes endpoints de l'API REST. De plus, des décorateurs sont ajoutés au frontend Django, ceux-ci permettent de protéger les routes frontend, ainsi si l'utilisateur n'a pas de token ou n'a pas les permissions administratives, il ne peut pas accéder aux pages 'admin'.
- Labo 4 : **Ajout de load balancing NGINX, du serveur de mise en cache Redis, d'outils de monitoring Prometheus+Grafana et de tests de charges au monolithe existant** – pour faire les tests de charge sur le monolithe du Labo 3, la librairie k6 était choisie avec 3 types de tests de charges différents pour refléter les différents cas d'utilisation : obtenir les rapports, obtenir l'inventaire de 2 magasins un à la suite de l'autre, mettre à jour les informations d'un produit. Ceux-ci couvrent la grande majorité des opérations effectuées par les utilisateurs. Ensuite, une extension à Express.js était ajoutée (express-prom-bundle) pour permettre au serveur Prometheus de recolleter les métriques du backend. Ensuite, un serveur Grafana est ajouté pour représenter ces données dans un dashboard

contenant les 4 Golden Signals parmi plusieurs graphiques.
Voici les données des graphiques prises sous les tests de charge pour le monolithe :

Ensuite, un balanceur de charge NGINX était ajouté pour supporter au moins 2 instances de l'application Express.js, mais comme que Express.js utilisais une map pour stocker les tokens, il fallait ajouter Redis pour faire la mise en cache des tokens entre les instances, car on ne pouvait jamais assurer avec quelle instance l'utilisateur communiquerais.
Ainsi, les mêmes tests de charge ont été testé et il est possible de remarquer des améliorations sur les graphiques surtout au niveau des latences :



Cela indique que l'ajout de NGINX et de Redis ont bonifié la performance et la résilience de l'application.

Enfin, logging était ajouté au système avec l'extension 'pino' ce qui a facilité grandement la lecture des logs dans la console.

Ainsi, le système est testé et reste fonctionnel lorsqu'une instance d'Express.js tombe en panne avec la commande de docker compose stop « nomContainer ».

2. Analyse critique et discussion

Labo 0 : Architecture Mainframe

- Points forts :
  - Simple à développer et déployer : C'est qu'une architecture avec 1 seul composant du système.

    - Pas besoin de gérer les communications entre composants : Le système est composé d'un seul composant.
- Points Faibles :
    - Scalabilité horizontale non existante : Pas possible de créer plusieurs nœuds qui communiquent ensemble.
    - Besoin de relancer l'application quand une modification est faite : comme que c'est un mainframe, tout doit être relancé pour utiliser les nouvelles fonctionnalités.
    - Si le système tombe en panne, tout tombe en panne : le mainframe s'occupe de tout et s'il tombe en panne le système n'est plus opérationnel.
    - Difficile à maintenir à long terme si l'application grandit beaucoup : le mainframe regroupe toute la logique ce qui complexifie la maintenance.
- Docker et Docker Compose : Un seul conteneurs docker est fait pour l'application mainframe.

Labo 1 : Architecture 2-tier

- Points forts :
    - Séparation de la logique entre le client et la base de données : la console client Python interagit avec la base de données MongoDB et n'est donc plus un mainframe.
    - Utilisation d'un ORM : mongoengine est utilisée pour manipules des objets Python au lieu d'écrire manuellement des requêtes MongoDB en JSON et assure qu'on a accès aux informations à jour dans la base de données.
    - Mise en place de Docker Compose (pour les 2 conteneurs) : Permet de lancer toute l'architecture avec une seule commande pour me simplifier le développement et les tests. Techniquement la console Python ne devrait pas être conteneurisé, car elle se retrouverait sur une machine du client, cela était fait juste pour faciliter les tests avec Docker (plus d'infos dans la section Docker et Docker Compose).
- Points Faibles :
    - Console Python conteneurisée : Dans une architecture en production/dans la vraie vie, la console Python ne devrait pas être conteneurisée, car elle est exécutée sur la machine d'un client. Outre

que pour me faciliter le développement, mon apprentissage de Docker et de Docker Compose, cette manière nuit à la séparation entre client/serveur.
    - Sécurité faible : Le client peut accéder directement à la base de données.
    - Gestion de la logique et de validation : Toute la logique de traitement et de validation est dans le client et donc il est difficile de centraliser la logique métier. Préférablement, le client ne devrait pas avoir ça, cela sera réglée dans le futur laboratoire.
- Docker et Docker Compose : 2 conteneurs docker sont lancés par Docker Compose (un conteneur contient la console Python et l'autre c'est une image de MongoDB officielle). Cela est pour but de pouvoir lancer toute l'architecture avec une commande, mais ça reste quand même une architecture 2-tier et un ORM est utilisée pour la communication. Techniquement la console Python ne devrait pas être conteneurisé, car elle se retrouverait sur une machine du client, cela était fait juste pour faciliter les tests avec Docker. J'ai fait cela de cette manière aussi pour me pratiquer plus avec Docker et Docker Compose pour pouvoir lancer des conteneurs en mode interactif.

Labo 2 : Architecture 3-tier avec DDD

- Points forts :
    - Séparation plus profonde de logique métier : La console Python agit comme un client externe. Le backend Express.js encapsule la logique métier et gère les appels API. La base de données MongoDB est accédée via mongoose comme ORM.
    - Le client ne connait pas la base de données : il n'est plus connecté à la base de données directement.
    - Utilisation d'un API : permet d'être réutilisée pour d'autres types d'application client (comme pour Django par exemple)
    - Application du DDD et du MVC : le backend est organisé selon les modèles et les services/contrôleurs.
- Points Faibles :
    - Pas d'authentification : n'importe qui peut faire les appels à l'API sans être validé.
    - Client reste en ligne de commande : Pas préférable pour des vrais utilisateurs, sera remplacée par Django dans le futur laboratoire.

    - Pas de tests pour Express.js : Les tests Express.js ne sont pas écrits encore, car l'API risque d'évoluer avec le futur laboratoire, seuls les tests Python sont faits.
    - Absence de load balancing : Pas de load balancer pour équilibrer la charge entre les composantes si le trafic est élevé.
- Docker et Docker Compose : 2 conteneurs docker sont lancés par Docker Compose (un conteneur contient le backend Express.js et l'autre c'est une image de MongoDB officielle). Cela est pour but de pouvoir lancer le backend et la base de données à partir d'une seule commande via Docker Compose. L'architecture du système reste quand même une architecture 3-tier avec DDD, car la console Python peut être lancée et fera des appels API par http au backend Express.js qui lui une fois reçu la demande va passer par l'ORM mongoose pour exécuter des queries sur MongoDB et ensuite envoyer le résultat au frontend Python.

Labo 3 : Architecture « 3-tier » Domain Driven avec REST API + frontend Django (backend monolithique)

- Points forts :
    - Mêmes points forts que pour le Labo 2.
    - Ajout de service d'authentification pour la connexion des utilisateurs.
    - Ajout de mot de passes chiffrées dans la base de données pour plus de sécurité.
    - Ajout de la génération de token pour protéger les routes APIs et les routes frontend.
    - Ajout d'une interface web/mobile responsive avec Django (et avec bootstrap5).
    - Ajout de la documentation Swagger interactive.
    - Meilleure organisation des routes pour REST API (+ gestion des versions de l'API).
    - Ajout de CORS dans express.js qui vise l'adresse du serveur frontend pour permettre que les appels depuis ce client (et depuis Postman pour des raisons de tests)
- Points Faibles :
    - Pas de tests pour Django : Les tests unitaires côté frontend n'ont pas été fait dû à la limitation de temps et parce que les tests unitaires backend ont priorité.

    - Absence de load balancing : Pas de load balancer pour équilibrer la charge entre les composantes si le trafic est élevé.
    - Absence de logging sophistiqué : console.log et print sont utilisées présentement.
    - Si les serveurs Express.js ou Django tombent en panne, la fonctionnalité du système est grandement diminuée
    - Manque de paginations et de tri pour éviter des surcharges sur la bd.
- Docker et Docker Compose : 2 conteneurs docker sont lancés par Docker Compose (un conteneur contient le backend Express.js et l'autre c'est une image de MongoDB officielle). Cela est pour but de pouvoir lancer le backend et la base de données à partir d'une seule commande via Docker Compose. Ainsi, le serveur Django doit être lancé pour avoir les fonctionnalités frontend.

Labo 4 : Ajout de load balancing NGINX, du serveur de mise en cache Redis, d'outils de monitoring Prometheus+Grafana et de tests de charges au monolithe existant

- Points forts :
    - Mêmes points forts que pour le Labo 2 et 3.
    - Ajout de load balancing.
    - Ajout de continuité de support si une instance d'Express.js tombe en panne.
    - Ajout de mise en cache des tokens dans un Redis Server.
    - Ajout de monitoring des performances avec Prometheus+Grafana
    - Ajout de logging sophistique avec 'pino'
- Points Faibles :
    - Pas de tests pour Django : Les tests unitaires côté frontend n'ont pas été fait dû à la limitation de temps et parce que les tests unitaires backend ont priorité.
    - Manque de paginations et de tri pour éviter des surcharges sur la bd.
    - La base de données est le « bottleneck » du système ce qui pourrait être corrigé avec l'introduction de services.
- Docker et Docker Compose : Une suite des containers est lancée avec les containers suivants avec la commande « docker compose up --build » :
    - Un container NGINX pour le load balancing
    - Un container avec le serveur Redis pour la mise en cache
    - Un container pour la base de données mongoDB

    - Un container pour le serveur Prometheus
    - Un container pour le serveur Grafana
    - Deux containers (par défaut) pour les instances de l'appli Express.js (api1 et api2)

3. Contraintes

Voici la liste des contraintes pour l'architecture logicielle :

- L'application est développée et hébergée sur une machine virtuelle de l'ÉTS, il est nécessaire d'être dans le réseau de l'université (ou utiliser un VPN) pour se connecter par SSH à la VM.
- MongoDB est déjà mis en place comme la base de données principale depuis le « Labo 1 », il faut continuer de l'utiliser pour éviter de migrer la logique avec la base de données.
- Un pipeline CI/CD doit être maintenu (mis en place dans le « Labo 0 ») avec GitHub Actions et la version finale de code pour chaque laboratoire doit passer toutes les étapes :
    - Ne pas avoir d'avertissements de formattage de fichiers avec PyLint
    - Passer tous les tests automatisés avec Pytest (et avec Jest pour Express.js)
    - Avoir un dockerfile fonctionnel pour créer une image de l'application
    - Être capable de publier cette image sur Docker Hub en gardant l'image générée à l'étape précédente comme un artéfact.
- Le frontend de l'application doit avoir la possibilité d'évolution vers une interface web/mobile. Cela est fait avec Django + Bootstrap5.

Notes additionnelles sur les contraintes :

- Avec l'ajout de Django comme framework frontend, les tests unitaires ne sont pas faits, mais les tests unitaires backend pour tester les endpoints sont faits.
- Avec l'ajout de load balancing NGINX, il faut changer les anciens appels vers le NGINX au lieu de l'appli Express.js

4. Contexte & portée

Depuis le « Labo 2 », le système doit être en mesure de pouvoir gérer les inventaires et d'autres fonctionnalités de plusieurs magasins. Ainsi, des collections de magasins et

de demandes de réapprovisionnement sont ajoutés à la base de données MongoDB pour supporter ces nouvelles demandes. Il existe également deux types d'utilisateurs dans le système : utilisateur d'un magasin standard et l'utilisateur du magasin mère (utilisateur admin).

Depuis le « Labo 3 », la manière dont on accède est via localhost:8000 qui est le serveur frontend Django. Lorsque l'utilisateur accède au lien, il est obligé de se connecter pour accéder au reste de l'application, car toutes les routes frontend sont protégés par des décorateurs qui vérifient l'utilisateur qui est connecté.

Ainsi, l'utilisateur du magasin standard peut :

- Rechercher un produit dans son magasin
- Enregistrer une vente dans son magasin
- Faire un retour de vente dans son magasin
- Consulter l'inventaire de son magasin
- Consulter l'inventaire du centre de stockage
- Faire une demande de réapprovisionnement

Tandis qu'un utilisateur du magasin mère peut :

- Rechercher un produit dans un magasin standard de son choix
- Enregistrer une vente dans un magasin standard de son choix
- Faire un retour de vente dans un magasin standard de son choix
- Consulter l'inventaire dans un magasin standard de son choix
- Consulter l'inventaire du centre de stockage
- Faire une demande de réapprovisionnement dans un magasin de son choix
- Générer un rapport consolidé des ventes
- Visualiser les performances de magasins
- Mettre à jour les informations d'un produit dans tous les magasins

5. Exigences

Pour éviter la répétition des informations, les exigences ont été mentionnées dans les sections précédentes de ce rapport. Cependant, les exigences fonctionnelles par MoSCoW ont été ajoutés dans le « Labo 2 » :

- Essentielles (Must have) :

- UC1 – Générer un rapport consolidé des ventes : Un gestionnaire à la maison mère génère un rapport détaillé contenant les ventes par magasin, les produits les plus vendus, et les stocks restants. Ce rapport est utilisé pour la planification et les décisions stratégiques.
  Celle-ci est ajouté au système du « Labo 2 », car c'est une exigence essentielle.
- UC2 – Consulter le stock central et déclencher un réapprovisionnement : Un employé d'un magasin consulte le stock disponible dans le centre logistique. Si un produit est insuffisant localement, il peut initier une demande d'approvisionnement depuis son interface.
  Celle-ci est ajouté au système du « Labo 2 », car c'est une exigence essentielle.
- UC3 – Visualiser les performances des magasins dans un tableau de bord : Un gestionnaire de la maison mère accède à un tableau de bord synthétique affichant les indicateurs clés : chiffre d'affaires par magasin, alertes de rupture de stock, produits en surstock, tendances hebdomadaires.
  Celle-ci est ajouté au système du « Labo 2 », car c'est une exigence essentielle.

- Souhaitables (Should have) :
  - UC4 – Mettre à jour les produits depuis la maison mère : Un responsable modifie les informations d'un produit (nom, prix, description). Les changements sont synchronisés automatiquement dans tous les magasins afin d'assurer une cohérence dans les points de vente.
    Celle-ci est ajouté au système du « Labo 2 », car elle était jugée nécessaire pour pouvoir mettre à jour les informations des produits lorsque le système aura plus de fonctionnalités.
  - UC5 – Approvisionner un magasin depuis le centre logistique : Le responsable logistique valide une commande de réapprovisionnement pour un magasin donné. L'opération déclenche le transfert du stock et met à jour les niveaux de stock dans les deux entités.
    Celle-ci n'est pas ajouté au système du « Labo 2 », mais possède déjà une bonne base de code dans le cas que cette exigence

deviendra essentielle, car les demandes de réapprovisionnement sont déjà sauvegardées dans la base de données avec toutes les informations nécessaires pour pouvoir faire cette exigence.

- Facultatives (Could have) :
    - UC6 – Alerter automatiquement la maison mère en cas de rupture critique : Lorsqu'un produit atteint un seuil critique de stock dans un ou plusieurs magasins, une alerte automatique est envoyée à la maison mère afin de permettre une action rapide (commande urgente, redistribution).
      Celle-ci n'est pas ajouté au système du « Labo 2 », à cause du manque de temps pour implémenter cette exigence et aussi, à cause du fait que la partie frontend de l'application est encore en ligne de commandes.
    - UC7 – Offrir une interface web minimale pour les gestionnaires : Une interface web légère permet aux gestionnaires d'accéder à distance aux indicateurs clés du système : ventes, stocks, alertes. Elle offre une visibilité rapide sans devoir accéder directement au système interne.
      Celle-ci n'est pas ajouté au système du « Labo 2 », à cause du manque de temps pour implémenter cette exigence et aussi, à cause du fait que la partie frontend de l'application est encore en ligne de commandes.

À la suite du Labo 2 , pour le « Labo 3 », quatre cas d'utilisation doivent être présents dans le système :

- UC1 : Générer un rapport consolidé des ventes - Permet d'obtenir un résumé agrégé des ventes réalisées dans tous les magasins pour une période donnée.
- UC2 : Consulter le stock d'un magasin spécifique - Permet d'interroger le niveau de stock d'un magasin identifié par son ID.
- UC3 : Visualiser les performances globales des magasins - Fournit un tableau de bord regroupant des indicateurs clés de performance (ventes, fréquentation, disponibilité des stocks).
- UC4 : Mettre à jour les informations d'un produit - Permet de modifier les attributs d'un produit existant (nom, prix, stock, etc.) dans la base de données

L’entièreté de ces quatre cas d’utilisation étaient déjà implémentés dans la version précédente du logiciel. Ainsi, il suffisait qu’à les intégrer avec Django.

Le diagramme de la vue cas d’utilisation sur la *Figure #1* ci-dessous illustre les acteurs principaux du système ainsi que les fonctionnalités dont ils ont accès.

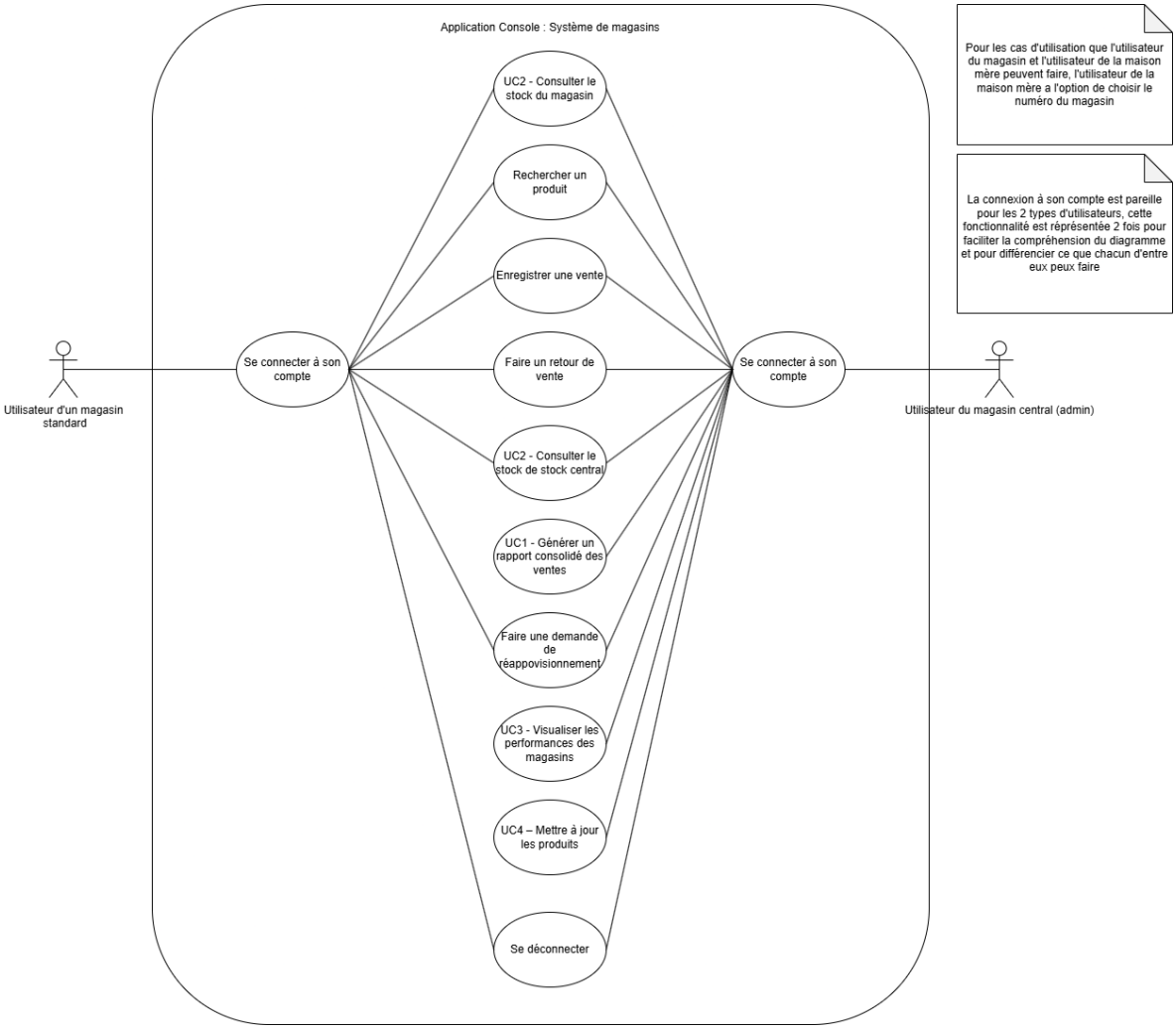


Figure #1 : Diagramme de la vue cas d’utilisation

6. Vue logique

Avant le « Labo 2 », les modèles des classes étaient stockés en Python et étaient basés sur les données dans les collections MongoDB.

Depuis le « Labo 2 », Express.js stocke ces modèles de classes dans le dossier « models » et leurs relations avec les autres objets sont décrites dans le diagramme ci-dessous sur la *Figure #2*.

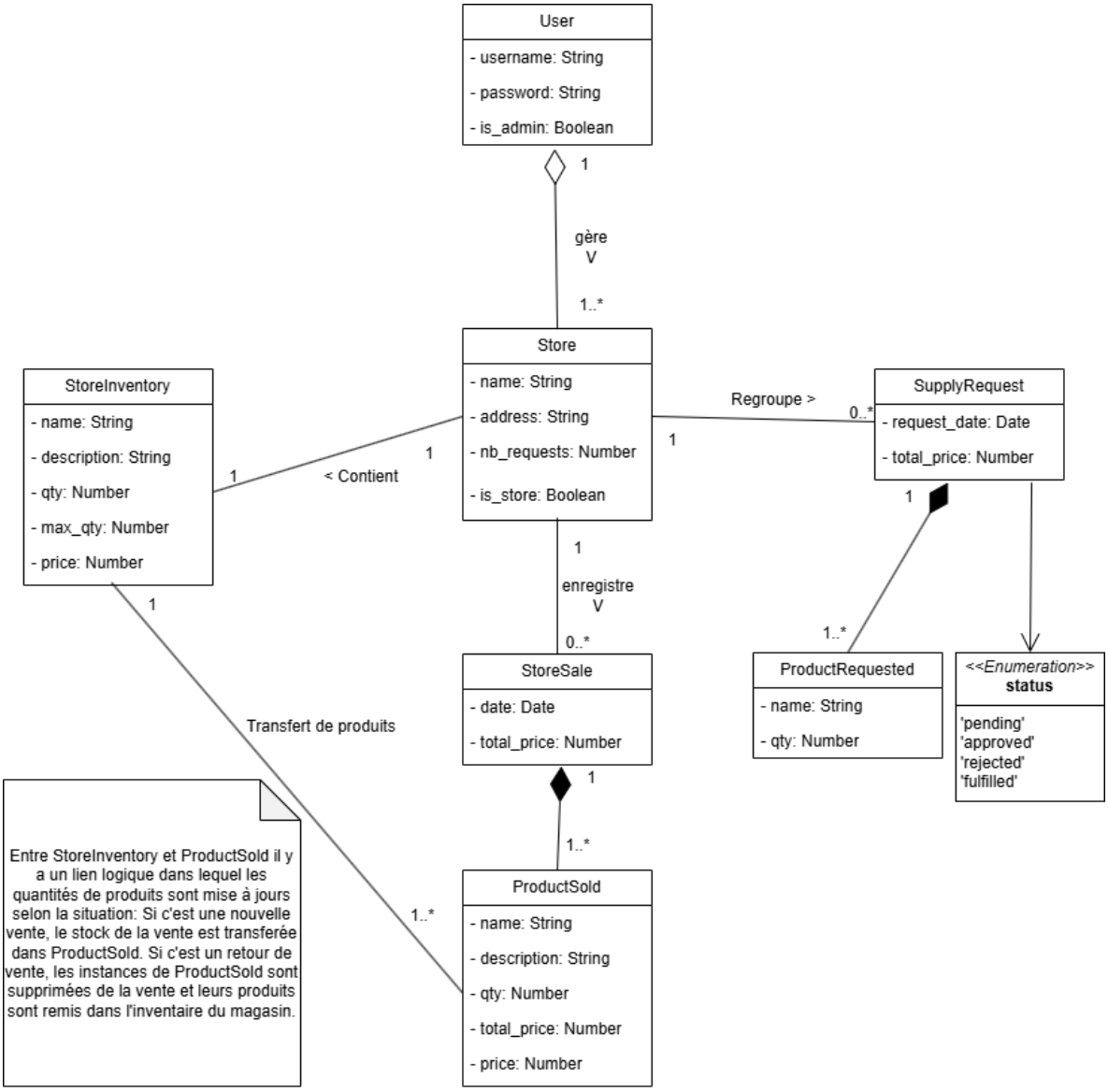


Figure #2 : Diagramme de la vue logique

Store.js : Représente une instance d'un magasin standard, magasin mère ou si un centre de stockage. De plus, ce sont les ventes et les demandes d'approvisionnement qui font la référence vers le magasin et pas l'inverse. Cela ne causerait pas de problème pour le centre de stockage, car il n'est pas possible pour celui-ci de faire de ventes dans le système.

StoreInventory.js : Représente une instance d'inventaire d'un magasin, celle-ci possède un lien relationnel vers l'instance du magasin auquel il appartient. Ainsi, l'inventaire contient les informations sur ces produits et est mis à jour aussi selon les ventes qui sont fait dans le magasin (via les produits).

StoreSale.js : Représente une instance de vente d'un magasin, celle-ci possède un lien relationnel vers l'instance du magasin auquel il appartient. De plus, elle possède une liste de « EmbeddedDocument » qui s'appelle « ProductSold » qui contient les informations nécessaires pour un produit appartenant à cette vente. Ainsi, cette liste cesse d'exister une fois que la vente est supprimée de la base de données.

SupplyRequest.js : Représente instance de demande de réapprovisionnement d'un magasin, celle-ci possède un lien relationnel vers l'instance du magasin auquel il appartient. De plus, elle possède une liste de « EmbeddedDocument » pour les produits demandés dans la demande. Ainsi, cette liste cesse d'exister une fois que la vente est supprimée de la base de données. Aussi, il existe une énumération pour représenter le statut de la demande. Lors de la création de l'instance, le statut par défaut est en attente ('pending').

User.js : Représente un utilisateur du système. Regroupe les utilisateurs des magasins standards et les utilisateurs administratifs. Il peut gérer les magasins dont il a accès : les utilisateurs standard ont accès à leur magasin uniquement et peuvent consulter le stock central. Un utilisateur admin à accès à tous les magasins et à toutes les fonctionnalités du système. Lorsqu'un utilisateur est ajouté à la base de données son mot de passe est automatiquement chiffré avec bcrypt.

7. Vues processus

Cette section du rapport couvre le fonctionnement de toutes les fonctionnalités présentes dans le système. Voici quelques précisions sur les diagrammes à venir :

- L'acteur est l'utilisateur d'un magasin standard ou d'un magasin mère. Si l'utilisateur du magasin mère veut exécuter une fonction que l'utilisateur du magasin standard peut faire, il est sous-entendu que l'utilisateur du magasin mère choisit aussi le numéro du magasin standard ou lui-même sur lequel il veut exécuter son opération.
- Frontend est un Django Server lancé sur localhost:8000. Il est sous-entendu que pour toutes les opérations l'utilisateur envoie son token dans un header

« authorization » pour toutes les opérations outre que celle pour se connecter. Ainsi, le frontend reçoit des réponses JSON à partir du backend pour faire ces opérations.

- Backend Express.js représente le serveur lancé en localhost:3001 qui contient l'API pour gérer les demandes du frontend et les traités sur la base de données. Ainsi, il reçoit des réponses JSON à partir de la base de données une fois que les queries ont été exécutées**, mais avec l'ajout de balancement de charge avec NGINX, on appelle localhost:80 à l'extérieur de l'application Express.js**.
- StoreInventory, StoreSale, StoreRequest représentent les schémas définis avec « mongoose » qui sont nécessaires pour faire les queries MongoDB entre le backend et la base de données, c'est de cette manière que le ODM est implémenté dans le système.
- Base de données MongoDB représente le service de la base de données qui roule dans le système, celui-ci s'occupe d'exécuter les queries reçues à partir de « mongoose » et à envoyer une réponse sous format JSON au backend qui traite la réponse s'il le faut puis l'envoyer au frontend par la suite.
- **Il est sous-entendu que NGINX fait la redirection vers l'instance d'Express.js qui est libre de traiter le trafic dans les diagrammes qui s'en viennent**

Les figures suivantes illustrent une vue processus d'un cas d'utilisation dans le système :

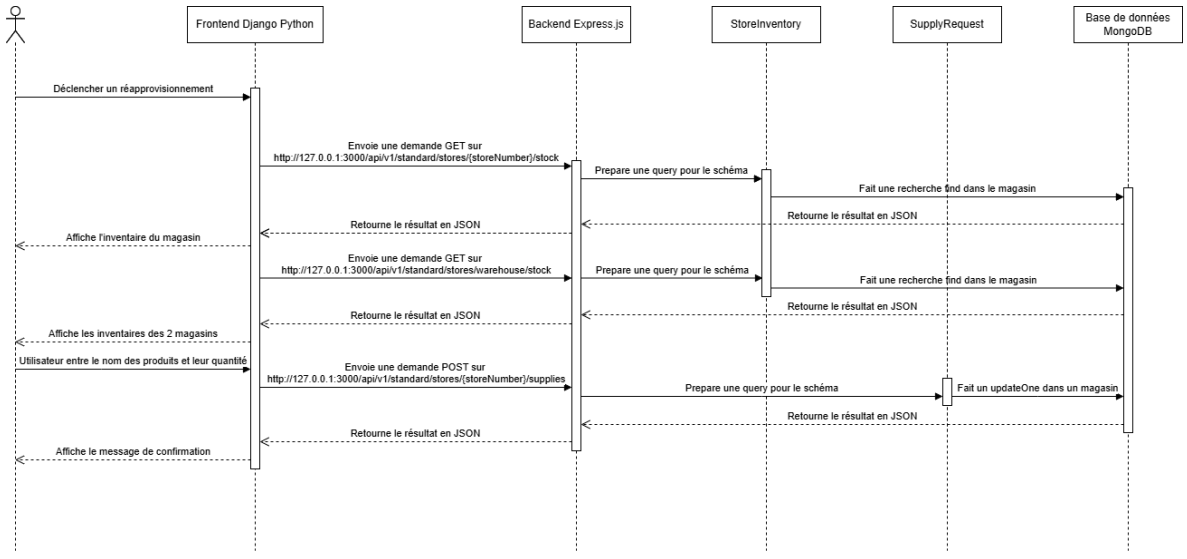


Figure #3 : Vue processus pour enregistrer une vente

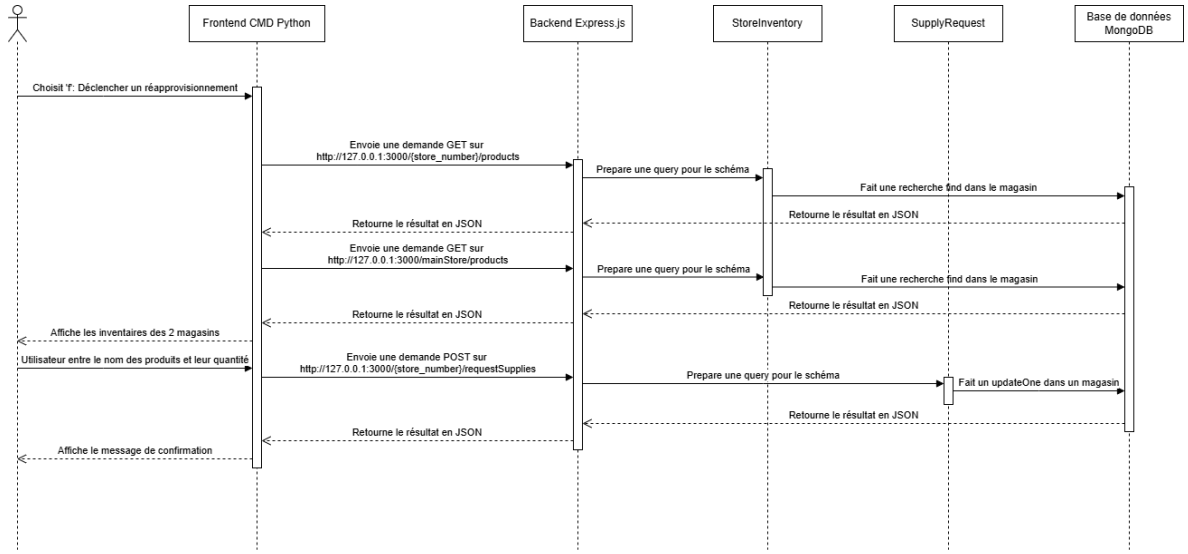


Figure #4 : Vue processus pour faire une demande de réapprovisionnement

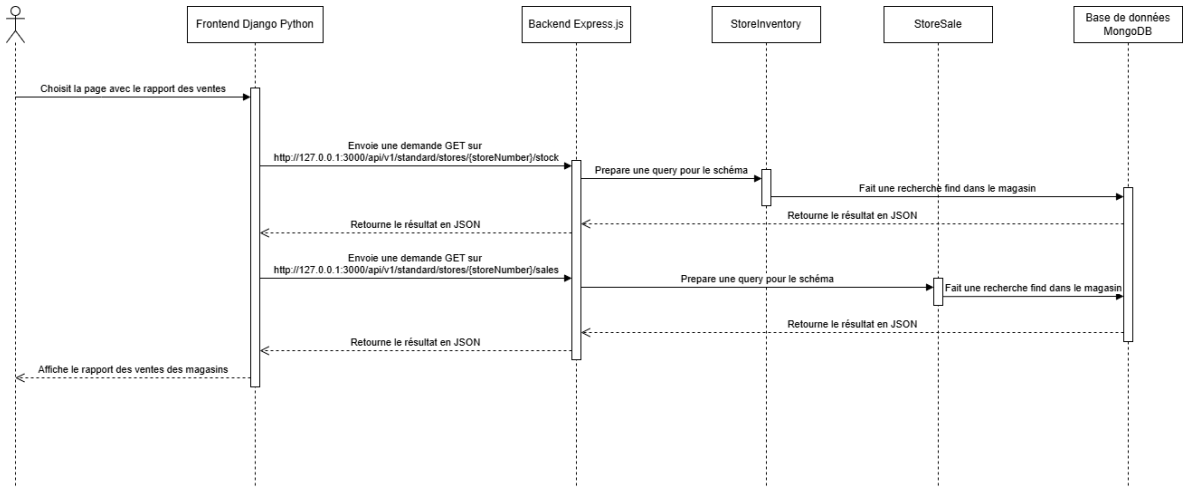


Figure #5 : Vue processus pour générer un rapport consolidé de ventes des magasins

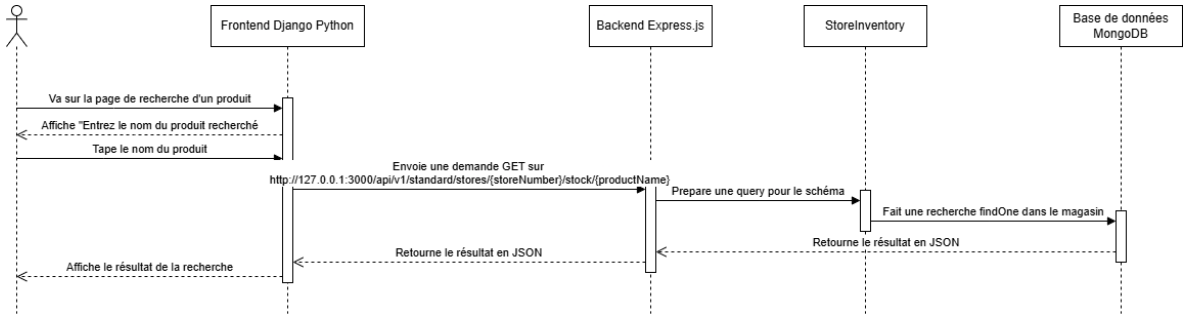


Figure #6 : Vue processus pour rechercher un produit dans un magasin

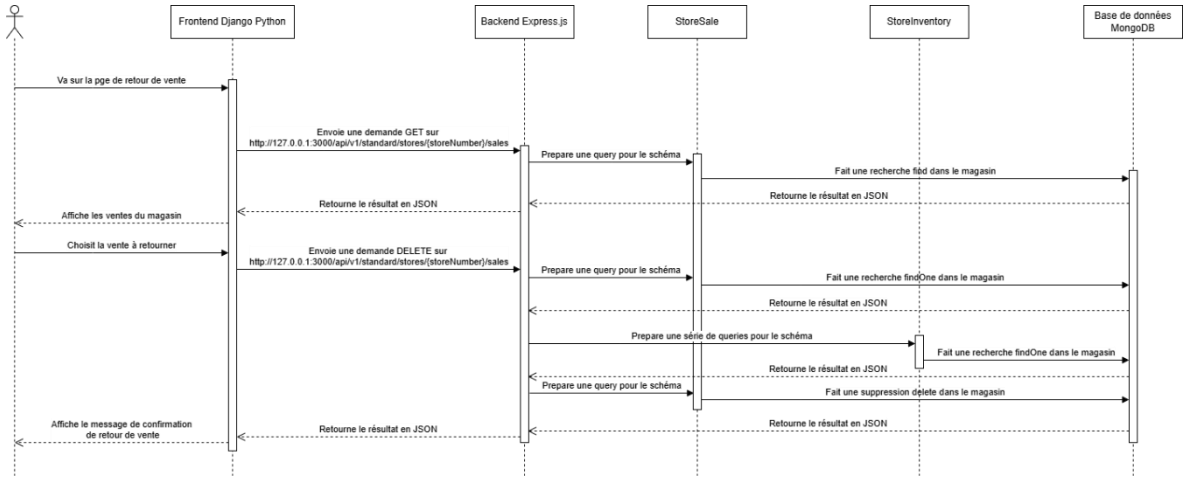


Figure #7 : Vue processus pour faire un retour de ventes

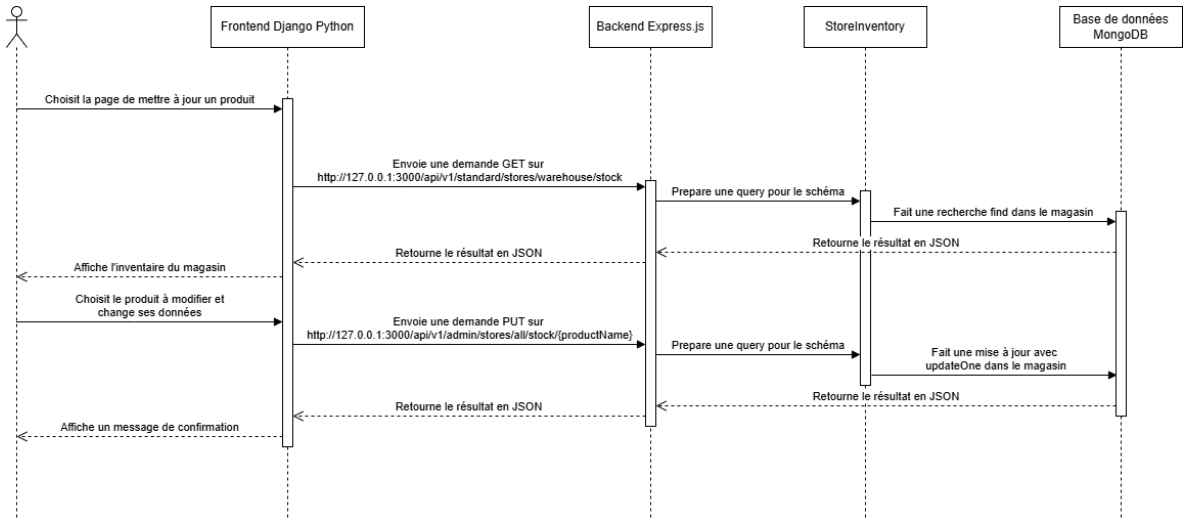


Figure #8 : Vue processus pour faire la mise à jour des informations d'un produit

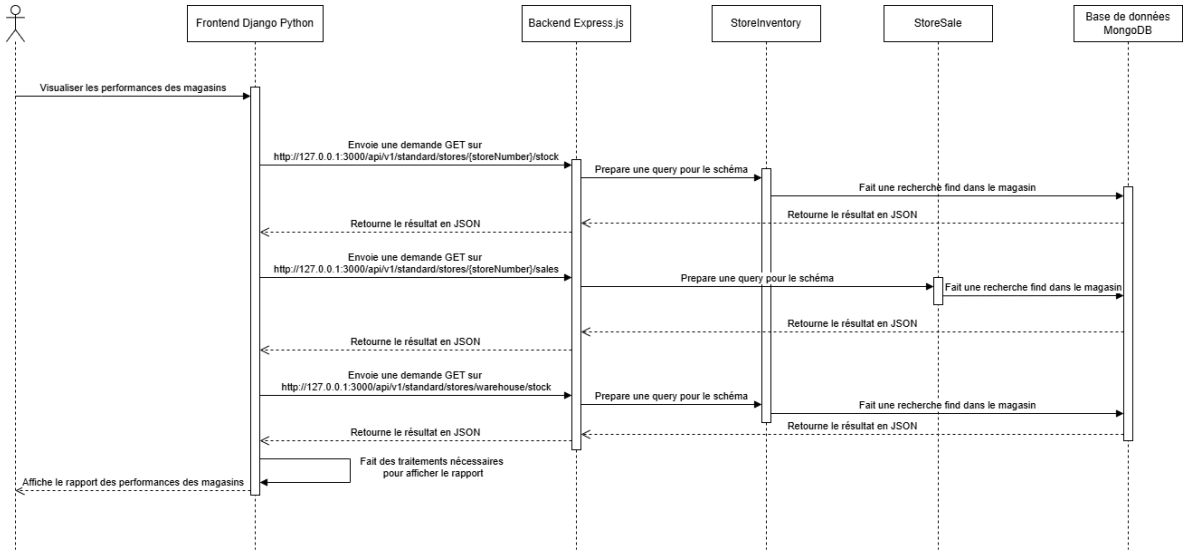


Figure #9 : Vue processus pour visualiser les performances des magasins

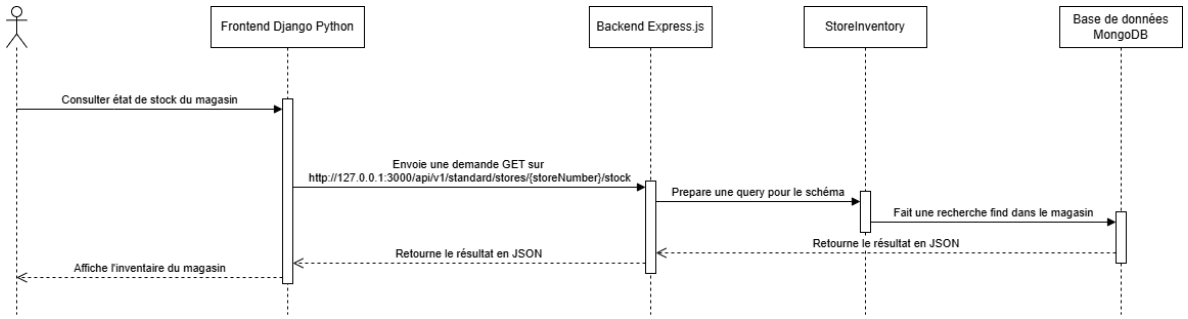


Figure #10 : Vue processus pour consulter l'inventaire d'un magasin standard

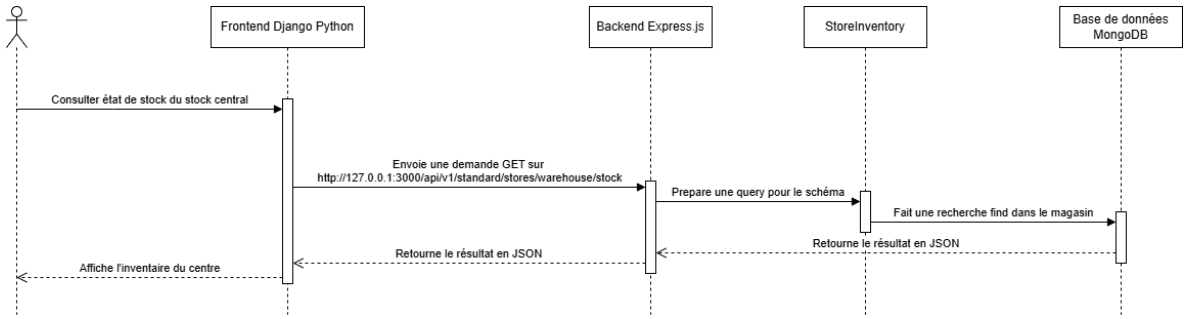


Figure #11 : Vue processus pour consulter l'inventaire du centre de stock

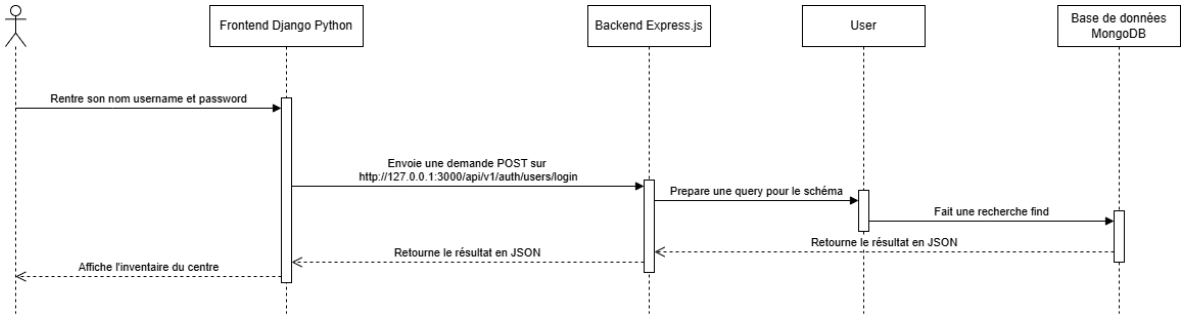


Figure #12 : Vue processus pour se connecter

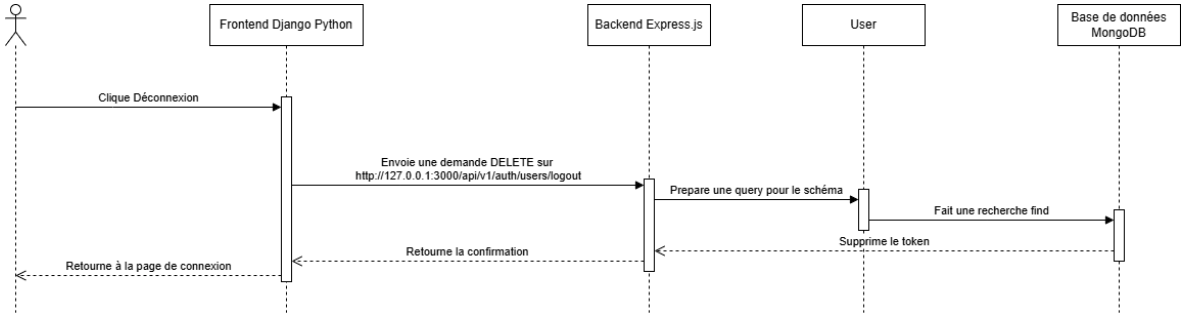


Figure #13 : Vue processus pour se déconnecter

8. Vue de déploiement

La vue de déploiement du système est représentée sur la *Figure #12* ci-dessous.

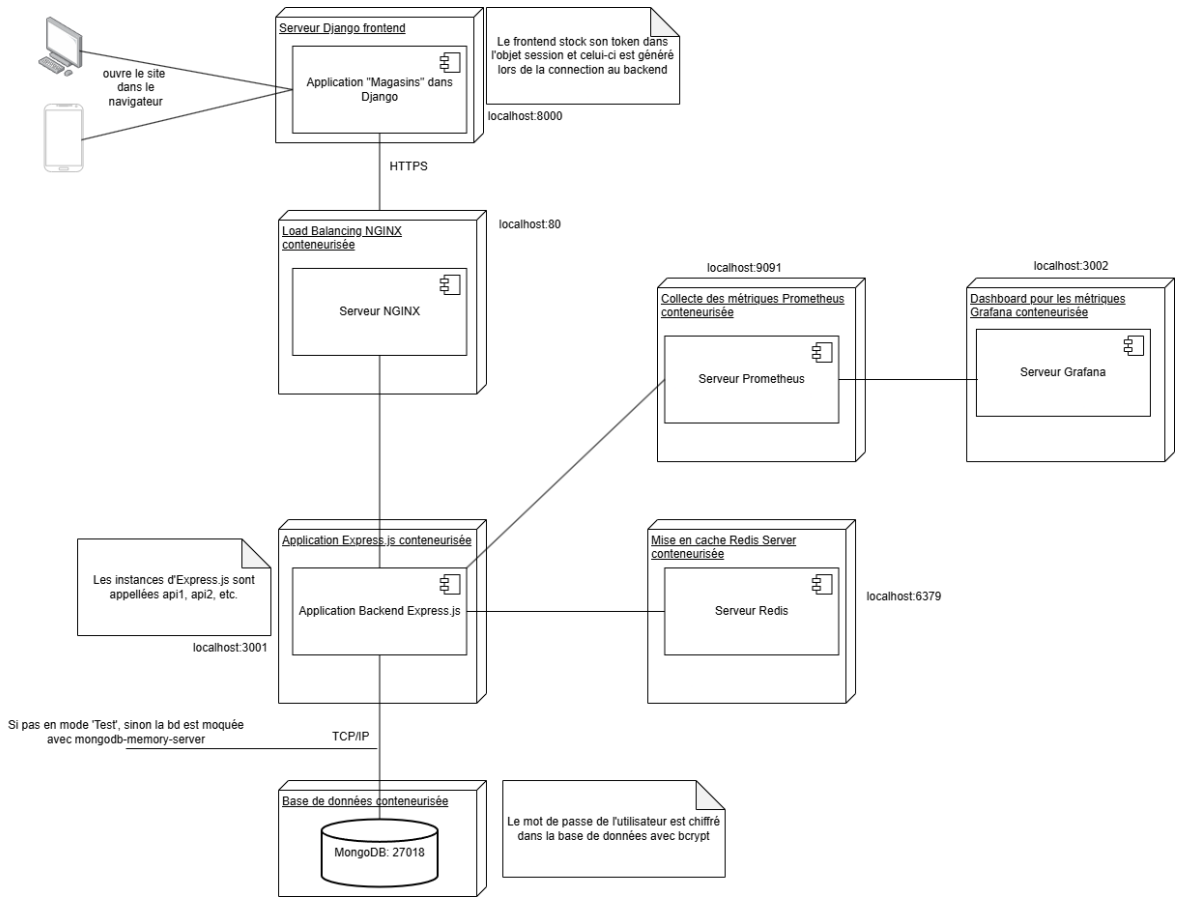


Figure #14 : Diagramme de la vue de déploiement

La vue courante de déploiement a évolué de manière significative comparée aux versions antérieures du système :

- Labo 0 – Seule l'application Python console était conteneurisée et contenait toutes les fonctionnalités du système.
- Labo 1 – Le système avait une architecture à 2 couches :
    - L'application Python console conteneurisée qui communiquait directement à la base de données.
    - La base de données est MongoDB qui elle aussi est conteneurisé, mais utilise une image officielle de Docker pour MongoDB.

- Labo 2 – Le système avait une architecture à 3 couches :
    - Console Python comme frontend
    - Pas de service d'authentification ni de CORS
- Labo 3 – Le système possède une architecture à 3 couches :
    - Le frontend est un serveur Django qui fait des appels API au backend via HTTP. Se trouve sur localhost:8000
    - Le backend est un serveur Express.js qui est conteneurisé grâce au Dockerfile, roule sur localhost:3001 et qui communique avec la base de données via la connexion TCP/IP
    - La base de données est MongoDB qui elle aussi est conteneurisé, mais utilise une image officielle de Docker pour MongoDB.
    - Pour les tests express.js la base de données est moquée avec mongodb-memory-server pour ne pas manipuler la vraie base de données.
    - L'utilisateur doit se rendre sur localhost:8000 pour se connecter au système par son navigateur.
- Pour le Labo 4 – Le système supporte NGINX, Redis et collecte des métriques Proimetheus+Grafana :
    - L'utilisateur doit se rendre sur localhost:8000 pour se connecter au système par son navigateur.
    - Le frontend est un serveur Django qui fait des appels API au backend via HTTP. Se trouve sur localhost:8000
    - Entre le frontend et les instances d'Express.js, il y a un serveur NGINX conteneurisé sur localhost:80 qui fait le load balancing pour les rediriger aux instances disponibles du backend.
    - Le backend est composé d'au moins 2 instances d'Express.js qui est conteneurisé grâce au Dockerfile, roule sur localhost:3001 et qui communique avec la base de données via la connexion TCP/IP
    - Un serveur Redis est utilisée pour gérer la mise en cache des tokens des utilisateurs et roule sur localhost:6379
    - Un serveur Prometheus ramasse les métriques des instances Express et roule sur localhost:9091
    - Un serveur Grafana roule sur locahost:3002 et ramasse les métriques de Prometheus pour les afficher dans un dashboard personnalisée et qui suite les 4 Golden Signals.

- La base de données est MongoDB qui elle aussi est conteneurisé, mais utilise une image officielle de Docker pour MongoDB.
- Pour les tests express.js la base de données est moquée avec mongodb-memory-server pour ne pas manipuler la vraie base de données.

9. Vue d'implémentation

Pour la version courante du laboratoire, l'organisation des fichiers ainsi que leurs associations sont représentées sur la *Figure #13* ci-dessous.

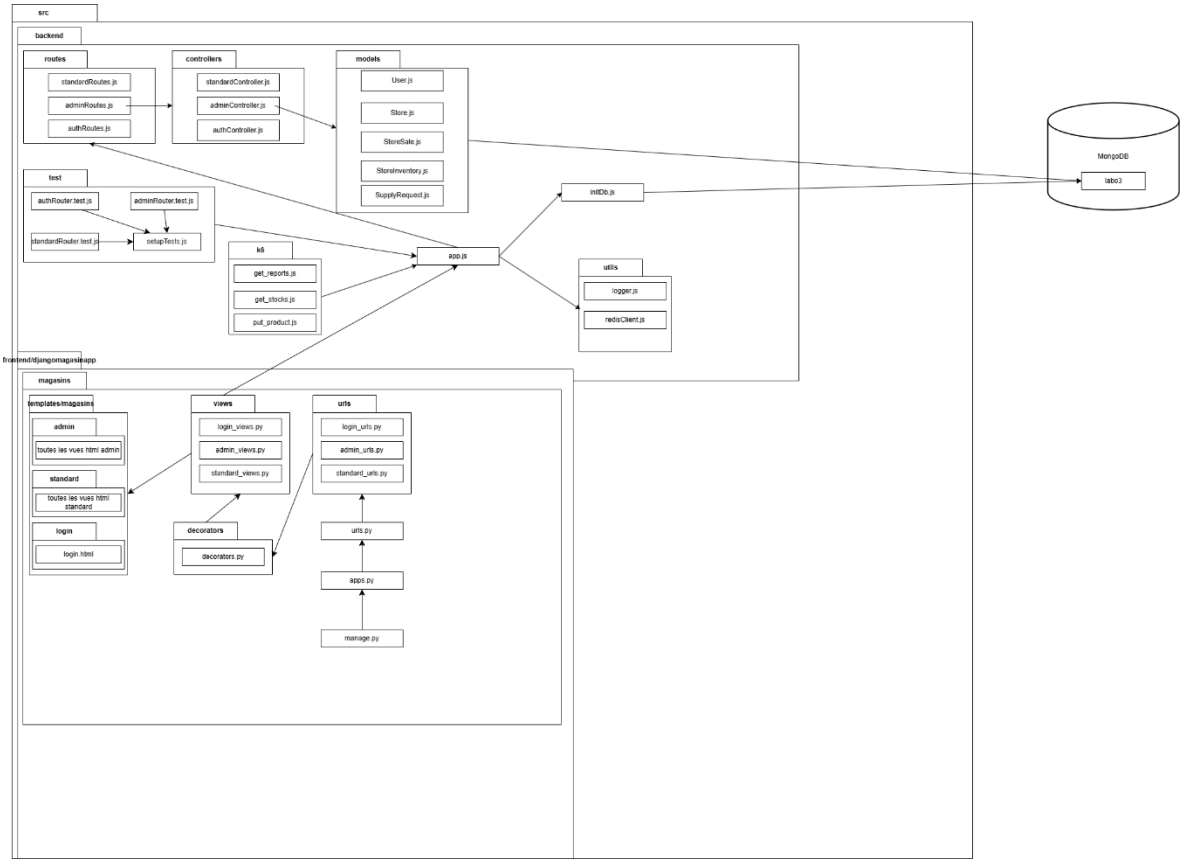


Figure #15 : Diagramme de la vue d'implémentation

Ainsi, le frontend qui est un serveur Django est lancé à partir de manage.py. Ce dernier envoie les requêtes API au backend pour afficher le contenu sous forme de site Web avec Bootstrap ce qui le rend responsive à la résolution. Le backend est un serveur Express.js lancé à partir de app.js. Une fois que ce dernier est démarré, celui-ci se connecte à la base de données à l'aide du fichier initDb.js ou créé les

collections nécessaires si la base de données est vide. Par la suite, app.js importe les trois routeurs (authRouter.js ,standardRouter.js et adminRouter.js) avec leurs propres routes API. authRouter.js est accessible à tous les utilisateurs tandis que les deux autres routeurs fonctionnent que pour leur type d'utilisateur respectif. Les routeurs utilisent un contrôleur associé pour exécuter les opérations. Ceux-ci importent les modèles de schémas mongoose pour faire des queries dans la base de données MongoDB. Pour configurer la connexion à redis le fichier redisClient.js est utilisé. Pour lancer les tests de charge, les fichiers correspondants sont dans le dossier k6. Pour le logging sophistiqué, la configuration est dans le fichier logger.js.

Les routes REST API sont les suivantes :

'/api/v1/api-docs' – pour la docum Swagger

'/api/v1/standard' – pour les routes standard

'/api/v1/admin' – pour les routes administratives

'/api/v1/auth' – pour les routes d'authentification

10. Décisions d'architecture (ADR)

Cette section couvre les décisions d'architecture considérés au cours des laboratoires :

- Labo 0 : Pas de ADR (ce n'était pas demandé)
- Labo 1 : ADR #1 et ADR #2
- Labo 2 : ADR #3 et ADR #4

ADR #1 : Choix de la plateforme (langage de programmation)

Titre : Langage de programmation principal - **Python**

Statut : Acceptée

Contexte

Dans le cadre d'un système de caisse d'un petit magasin, l'application développée doit être facilement maintenable, rapide à développer et évolutive. Le langage Python permet la conception d'une telle application.

Décision

Le langage de programmation principal retenu pour l'application est **Python**. Ce choix repose sur les raisons suivantes :

- Approfondir les connaissances en Python.

- Les librairies disponibles pour Python permettent aussi d'avoir des fonctionnalités sans avoir le besoin de les coder.

- La simplicité et la lisibilité du langage facilitant la maintenance.

Conséquences - **Avantages:**

- **Maintenabilité :** Améliorée grâce à un code plus lisible et au support de tests avec Pytest.

- **Scalabilité :** Dans ce laboratoire, l'application risque d'avoir une architecture évolutive et les librairies de Python permettront d'avoir les outils pour s'adapter.

- **Rapidité de développement :** Élevée, car la syntaxe de Python est simple à comprendre et offre déjà des fonctions "built-in" pour éviter de les coder à partir de zéro.

Conséquences - **Inconvénients:**

- **Performance :** Python est généralement plus lent que les autres langages (ex: langages compliés comme C++, Java), mais c'est acceptable pour le cadre de cours LOG430.

ADR #2 : Choix de la base de données

Titre : Base de données choisie - **MongoDB**

Statut : Acceptée

Contexte

Dans le cadre d'un système de caisse d'un petit magasin, l'application développée doit être facilement maintenable et rapide à développer. Puisque MongoDB est une base de données NoSQL, celle-ci peut être facilement modifiée et adaptée, et de l'évolution potentielle du magasin (dépendamment du laboratoire).

Décision

La base de données principale choisie pour l'application est **MongoDB**. Ce choix repose sur les raisons suivantes :

- Approfondir mes connaissances avec MongoDB.

- MongoDB est flexible et sans schéma strict ce qui convient bien aux données évolutives qui risquent de changer au cours des laboratoires.

- Une bonne intégration avec Python via Mongoengine et Mongoose pour Express.js, qui fournit une approche orientée objet et offrant une couche de persistance avec ODM.

- MongoDB peut être visionné et manipulé à l'aide du shell officiel (mongosh)

Conséquences - **Avantages:**

- Flexibilité du schéma : Permet d'ajouter ou de modifier les structures de données avoir le besoin de modifier les collections puisque MongoBD est une base de données NoSQL.

- Rapidité de développement : Moins de contraintes sur les schémas initiaux puisque MongoDB est une base de données NoSSQL. Cela permet d'avancer plus rapidement l'écriture de code.

- Intégration avec Python et Express.js : Grâce à Mongoengine et Mongoose, l'interaction avec la base de données reste simple et suit l'écriture des classes en Python et Express.js.

Conséquences - **Inconvénients:**

- Moins de contrôle sur la structure des données : L'absence de schéma rigide n'impose pas de règles de structure et peut avoir des incohérences si la structure n'est pas respectée.

ADR #3 : Choix de framework backend pour la communication entre frontend et la base de données

Titre : Framework backend - **Express.js**

Statut : Acceptée

Contexte

L’application est un système de gestion d’inventaire pour un un réseau de magasins. Elle comporte une interface utilisateur en ligne de commande (CLI) développée en Python qui agit comme frontend ainsi qu’une base de données MongoDB pour stocker les données relatives aux produits et ventes.

Ceci était l'architecture du dernier laboratoire.

Il faut évoluer cette architecture pour supporter une architecture 3-tier avec un framework backend qui agit comme intermédiaire entre le frontend et la base de données qui gère les communications via un API.

Ce serveur doit exposer une API simple, efficace, et facilement maintenable pour recevoir les requêtes du frontend et interagir avec la base de données.

Décision

Le framework backend pour la communication entre frontend et la base de données retenu est **Express.js**. Ce choix repose sur les raisons suivantes :

- Express.js est un framework léger et minimaliste pour Node.js, permettant de créer rapidement des APIs REST.

- Large communauté et nombreux modules complémentaires disponibles grâce à npm.

- Facilité d’intégration avec MongoDB via des bibliothèques comme Mongoose.

- Bonne performance et scalabilité pour gérer les requêtes entre le frontend et la base de données.

Conséquences - **Avantages:**

- Développement rapide grâce à la simplicité d’Express.js.

- Flexibilité dans la définition des routes et gestion des requêtes HTTP.

- Large écosystème et support communautaire grâce à npm.

- Facilité de déploiement et d’intégration avec d’autres services.

- Bonne gestion des middlewares pour la sécurité, le logging et la validation des données.

Conséquences - Inconvénients:

- Dans la dernière version de cette application, le "backend" qui était dans la CMD était en Python, il faudra migrer ce code en JavaScript.

- Il faut remplacer Mongoengine en Python par un autre ORM comme Mongoose.

- Nécessite une connaissance de JavaScript/Node.js en plus de Python pour la CLI.

- Gestion manuelle de certains aspects comme les erreurs et la structure du projet.

ADR #4 : Choix de l'ORM pour supporter MongoDB avec Express.js

Titre : ORM choisi - **Mongoose**

Statut : Acceptée

Contexte

L'application est un système de gestion d'inventaire pour un réseau de magasins. Elle comporte une interface utilisateur en ligne de commande (CLI) développée en Python qui agit comme frontend ainsi qu'une base de données MongoDB pour stocker les données relatives aux produits et ventes. Mongoengine était utilisé comme ORM entre Python et MongoDB

Ceci était l'architecture du dernier laboratoire.

Avec l'implémentation de Express.js, Mongoengine ne peut plus être utilisé, car le CMD de Python ne communique plus avec la base de données de manière directe. Il faut donc un ORM pour MongoDB entre Express.js et la base de données.

Décision

Le choix de l'ORM (équivalent d'un ODM pour les bases SQL relationnelles) s'est porté sur Mongoose. Ce module pour Node.js permet de définir des schémas de données, de valider les données en entrée et d'interagir avec MongoDB de manière structurée. Ainsi, Mongoose gère les mêmes responsabilités que Mongoengine.

Conséquences - **Avantages:**

- Intègre une validation des données avant l'insertion dans la base de données.

- Facilite les opérations CRUD grâce à des méthodes de haut niveau.

- Possède une large communauté et une bonne documentation.

Conséquences - Inconvénients:

- Courbe d'apprentissage initiale pour bien comprendre les fonctionnalités de Mongoose après avoir utilisé Mongoengine.

11. Demandes de qualité

Scalabilité horizontale :

Description : Le système doit pouvoir gérer un nombre croissant de magasins et, donc, d'utilisateurs sans dégrader sa performance.
Solutions mis en place :

- Requêtes HTTP vers un backend (Express.js) pour appeler l'API

Cohérence des données :

Description : Plusieurs utilisateurs peuvent enregistrer des ventes simultanément, sans corrompre les stocks ni les ventes.
Solutions mis en place :

- Implémentation d'un ODM (mongoengine et ensuite mongoose) pour assurer la cohérence des données

Évolutivité vers une interface web/mobile

Description : L'architecture doit permettre l'ajout futur d'une interface web ou mobile, sans réécriture majeure (pour éviter des migrations de code comme pour Express.js).
Solutions mis en place :

- API REST exposée via Express.js
- Architecture par couches (frontend – backend – base de données)

Maintenabilité et clarté du code

Description : Le code doit être facile à lire, à tester et à modifier.
Solutions mis en place :

- Organisation par modules (MVC = routes, contrôleurs, modèles)
- Séparation des couches (frontend - backend)
- Pylint pour la clarté et cohérence des fichiers
- Tests unitaires avec Pytest

- Pipeline CI/CD mis en place après chaque push sur la branche principale

Portabilité et déploiement

Description : Le système doit pouvoir être déployé facilement sur d'autres machines via des conteneurs Docker.
Solutions mis en place :

- Dockerfile (pour Docker) et Docker Compose pour la facilité de gestion des conteneurs
- Documentation d'exécution dans README.md

12. Risques & dette technique

Voici une liste des risques et des dettes techniques présents dans la version courante du système :

- Faire les tests côté frontend pour Django

- Finaliser les tests backend pour Express pour certaines routes

13. Glossaire

- « Magasin mère » : réfère au magasin principal qui peut faire les mêmes actions que les magasins standards, mais avec des fonctionnalités additionnelles.
- « Magasin standard » : réfère à un magasin qui n'est pas un magasin mère.
- « ORM » : réfère à Object-Relationnal Mapping utilisé pour les bases de données relationnelles.
- « ODM » : réfère à Object-Document Mapping utilisée pour les bases de données NoSQL.