Template tirée : https://www.dokchess.de/en/00_overview/

Introduction and Goals

Le but de cette application d'effectuer la gestion des ventes, de stocks, de produits, et des rapports pour les utilisateurs finaux. Pour y arriver, il faut respecter également les qualités de systèmes, les contraintes mises sur l'architecture et les besoins des parties prenantes importantes.

Requirements overview

Le but de cette exercice est de développer une système de point de vente pour une entreprise contenant des magasins, permettant également la gestion des magasins et de leurs produits.

Fonctionnalités essentielles:

- Rechercher un produit
- Acheter un produit
- Générer un rapport consolidé des ventes
- Consulter le stock central
- Déclencher un réapprovisionnement
- Visualiser les performances des magasins dans un tableau de bord

---

Quality Goals

Attributs de qualité	**Motivation / Description**
Maintenabilité	L'utilisation de tests et l'architecture 3-tier facilite la compréhension du système, permettant aux nouveaux développeurs de s'adapter au code rapidement. Les bugs sont rapidement corrigés en conséquence.
Évolutivitité	Le système est capable de s'adapter facilement à de nouveaux besoins et à une agumentation du nombre d'utilisateurs , et ce, sans changement majeurs de code.
Utilisabilité	Le système présente une interface claire et simple, et ce, sans erreurs majeurs qui pourrait perturbé l'expérience du client. Les cas d'utilisation sont clairement divisé dans le UI.
Performance	Le système doit permettre de repondre aux requetes du browser dans les délais les plus rapides, même quand la charge devient importante.
Rigidité	Le système doit pouvoir réagir adéquatement face à des pannes.

Stakeholders

Role/Name	**Contact**	**Expectations**

Role/Name	Contact	Expectations
Fabio Petrillo	fabio.petrillo@etsmtl.ca	Product Owner
Lojan Arunasalam	lojan.arunasalam.1@ens.etsmtl.ca	Responsable de l'architecture du système -- Developpeur du système

Architecture Constraints

Contrainte	Background ou motivation
Implémentation en Python	Le projet est developpé en Python et doit rester en Python, sauf à indication contra9ire.
Architecture microservices	sdf
DDD	sdf

System Scope and Context

Business Context

Communication parternaire	Input - Output
Employé magasin	Un employé peut effectuer une recherche de produit, ou acheter un produit. Également, si le stock d'un produit en local est proche de finir, il peut déclencher un réapprovisionnement. Une communication entre ces deux parties est donc requise.
Gestionnaire	Un gestionnaire peut effectuer une demande de rapport de ventes pour chaque magasin. Également, il peut visualiser les performances des magasins. Une communication entre ces deux parties est donc requise.
Base de données	Le système persiste les données effectués par les requêtes des utilisateurs

Technical Context

Mapping Input/Output to Channels

Channels	Input - Output
Browser	Reçoit en input des requêtes HTTP venant des utilisateurs et effectue le rendering des HTML en conséquence.
Web Server	Reçoit en input les requêtes HTTP du browser et exécute des requêtes vers la base de données

Channels	Input - Output
PostgreSQL	Reçoit en input les requêtes en TCP/IP et output les données

Solution Strategy

Attributs de qualité	Approche pour atteindre cette qualité
Maintenabilité	Utilisation d'une architecture 3-tier pour séparer les responsabitlités. Facilite la compréhension du système
Évolutivitité	Utilisation d'une architecture 3-tiers permet aux couches d'évoluer, et peut évoluer vers une architecture n-tiers
Utilisabilité	Interface utilisateur simple avec les cas d'utilisations sur une page différente
Simplicité	Limiter la complexité technique: ne pas overengineer

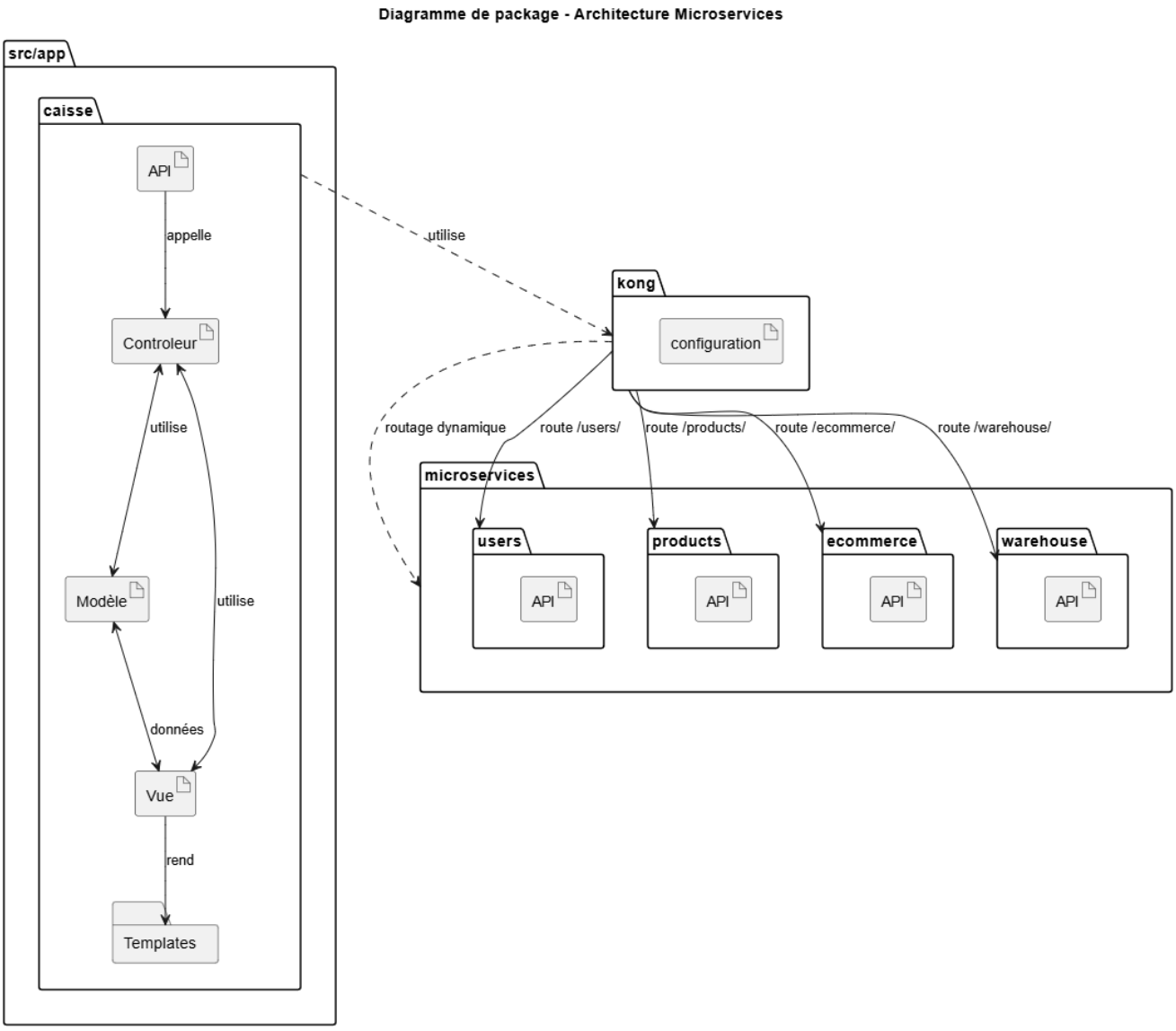
Building Block View

Level 1

Ce diagramme de composant illustre l'architecture hybride du système avec la coexistence de l'application Django et les nouveaux microservices. En effet, ce diagramme est le vue d'ensemble du système avec description des blocs de construction principaux.

Composants principaux :

- Package src/app : Application Django monolithique avec architecture MVC (Contrôleur, Modèle, Vue, Templates). L'application peut utiliser Kong afin d'accéder aux microservices
- Package kong : API Gateway pour le routage et la gestion des requêtes vers les microservices. En effet, il route dynamiquement les requêtes vers les services appropriés selon les endpoints
- Package microservices : Quatre services indépendants (users, products, ecommerce, warehouse) exposant chacun leur API REST. Ils ont chacun leur propre logique métier.



Level 2 zooms into some building blocks of level 1. Thus it contains the white box description of selected building blocks of level 1, together with black box descriptions of their internal building blocks.

N/A

Level 3 zooms into selected building blocks of level 2, and so on.

N/A

Whitebox Overall System

Nom (boîtes noires)	Responsabilité
Interface Utilisateur	Afficher les templates HTML et interagir avec l'utilisateur
Views.py	Rendering de l'HTML
controller.py	Couche logique pour les besoins métiers
models.py	Représentent les entités domaines

Nom (boîtes noires)	Responsabilité
Base de données	Persistent les données

Interface Utilisateur

- But: Permet l'interaction entre le client et le système
- Interface: HTTP - Il doit être réactive et facile à utiliser

views.py

- But: Permet de rediriger les requêtes HTTP vers les rendering appropriés
- Interface: HTTP - Fonctions pythons - Il doit être mince

controller.py

- But: Effectue la logique pour atteindre les besoins métiers, les uses cases.
- Interface: Fonctions pythons - Il doit être robuste

models.py

- But: Représentent les entités métiers
- Interface: Class python - Il doit garantir l'intégrité des données

Base de données

- But: Stockent les données
- Interface: Requêtes SQL via ORM - Il doit respecter les principes ACID

Runtime View

Diagramme de cas d'utilisation

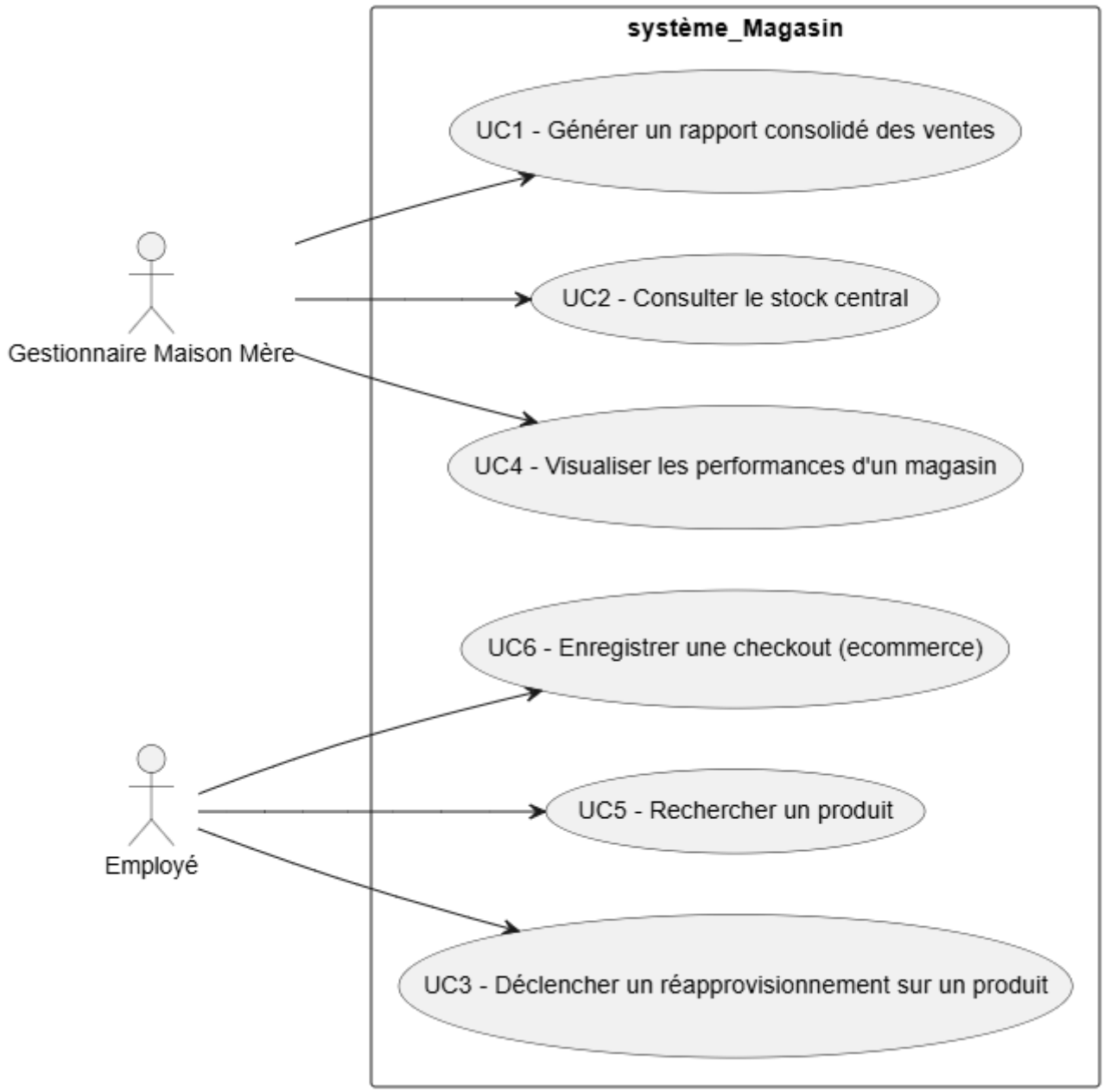
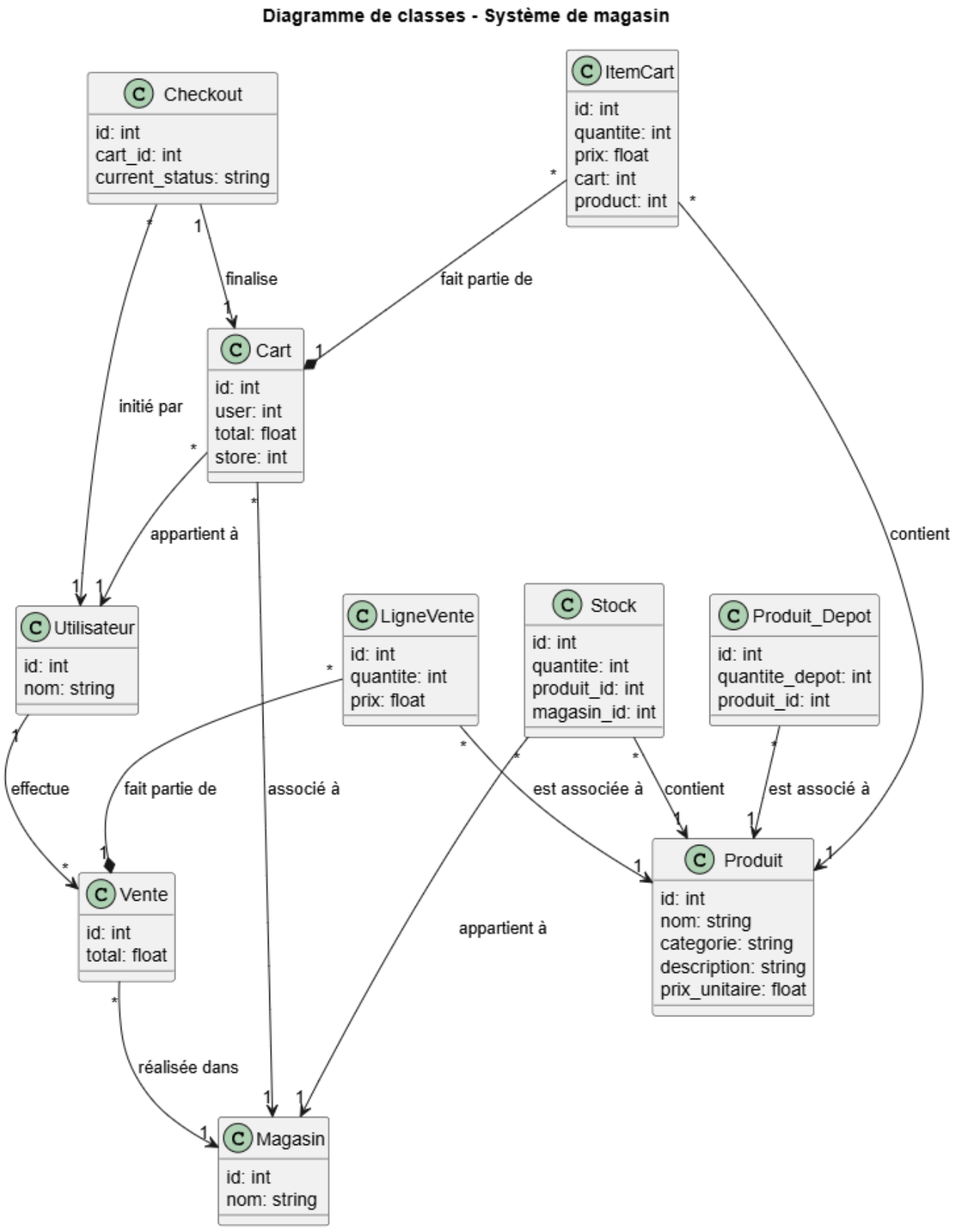


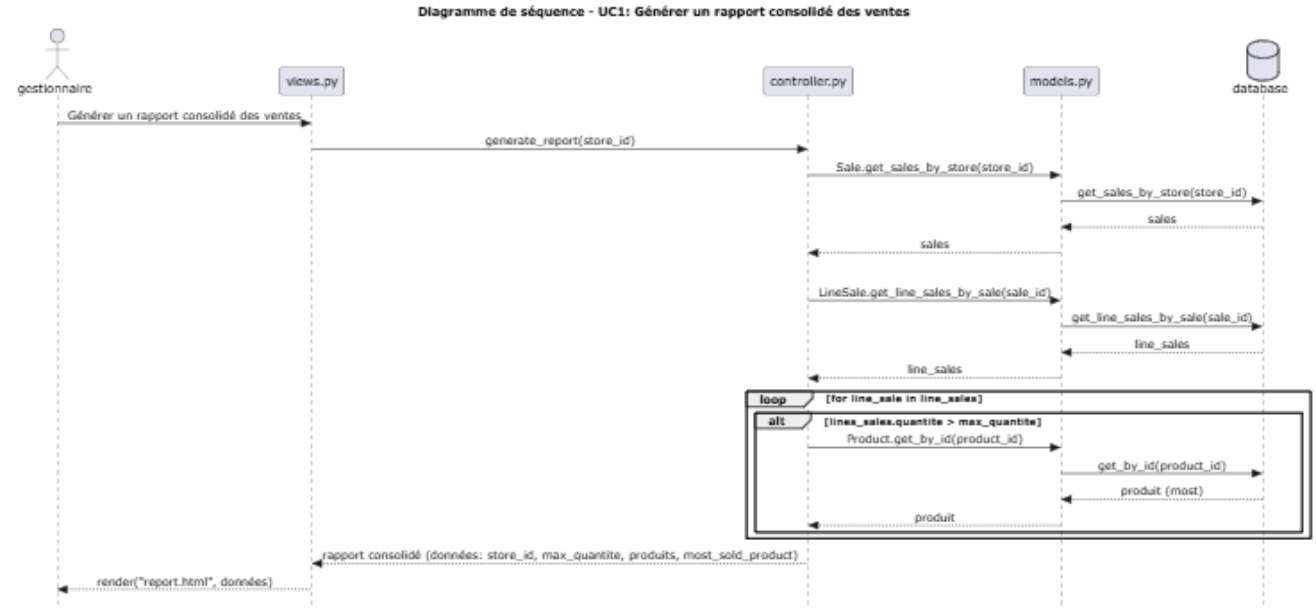
Diagramme de classe



Diagrammes de sequences

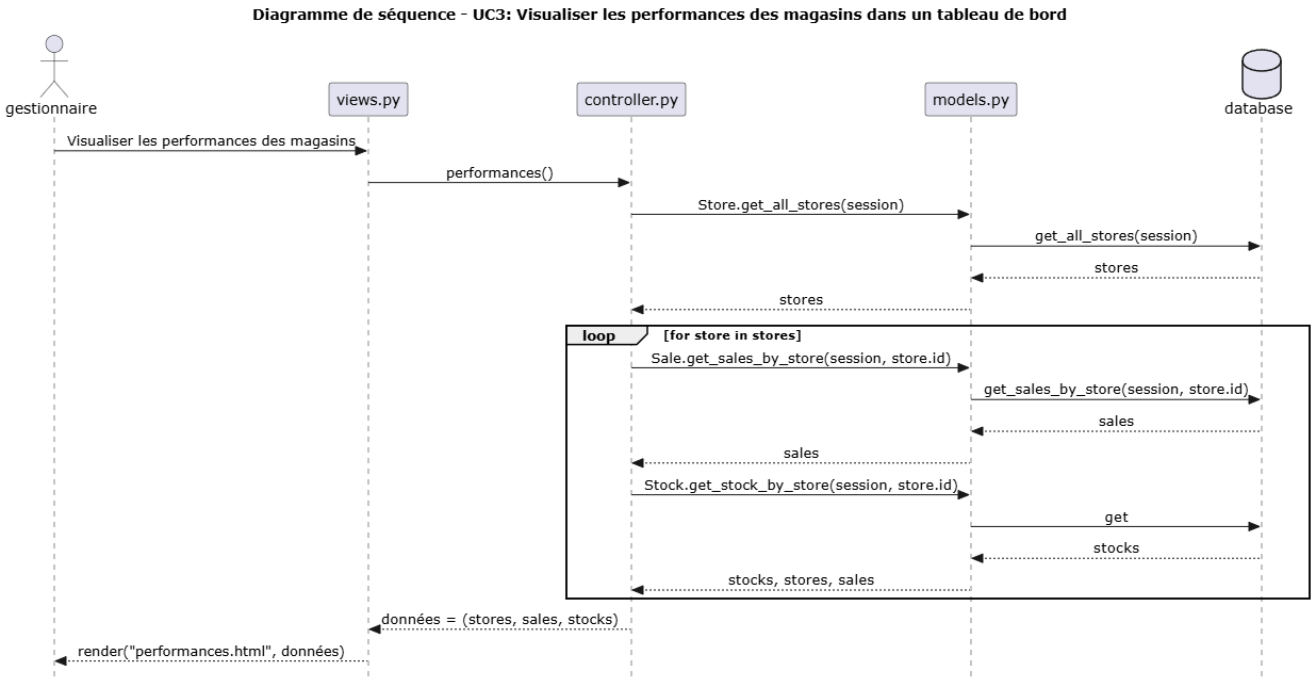
Voici quelques diagrammes de séquences importantes:

<Runtime Scénario 1 - UC1 >



<Runtime Scénario 2 - UC2>

<Runtime Scénario 3 - UC3>



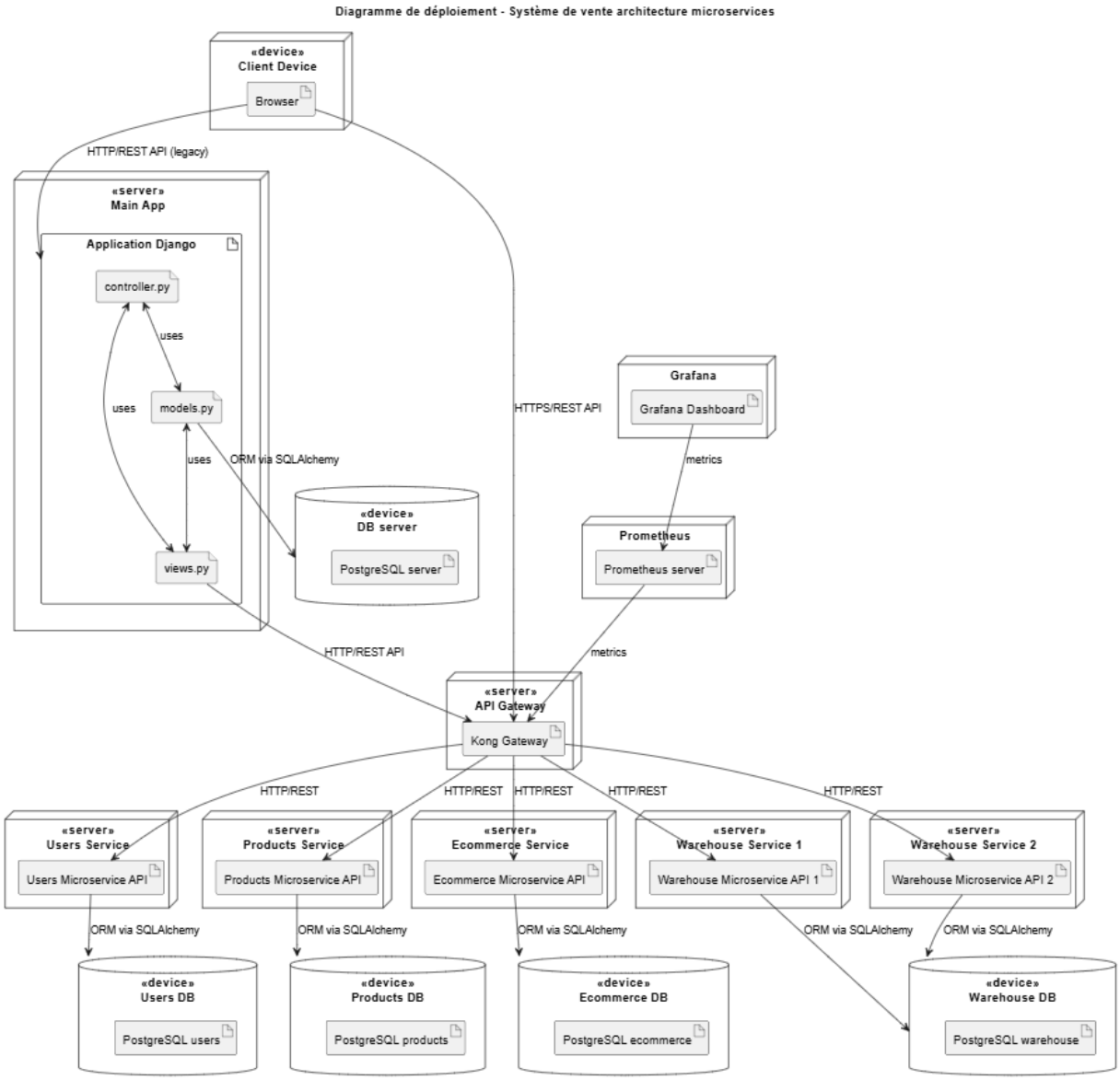
Deployment View

Ce diagramme représente l'architecture de déploiement hybride du système, montrant la relation entre l'application Django et les nouvelles microservices.

L'infrastructure comprend :

- API Gateway Kong comme point d'entrée unique pour les microservices
- Quatre microservices indépendants (Users, Products, Ecommerce, Warehouse) avec leurs bases de données dédiées
- Ancienne Application Django qui effectue des requêtes vers les nouvelles services.
- Stack de monitoring avec Prometheus et Grafana pour l'observabilité

- Scalabilité horizontale illustrée par les instances multiples du service Warehouse



Cross-cutting Concepts

UI/UX

L'interface utilisateur suit un principe de simplicité et de clarté pour faciliter l'utilisation par les employés de magasin et les gestionnaires.

Principes de design :

- Navigation intuitive : Chaque cas d'utilisation (recherche produit, achat, gestion stock) est accessible via des sections distinctes
- Consistance visuelle : Même palette de couleurs et composants UI à travers toute l'application

DDD (Domain Driven Design)

L'architecture microservices reflète une approche Domain Driven Design avec une séparation claire des domaines métiers.

Voici la séparation des domains métiers:

- Users Domain : Gestion des création des utilisateurs et authentification
- Products Domain : Catalogue produits et informations associées
- Ecommerce Domain : Panier d'achat avec items et processus de vente avec checkout
- Warehouse Domain : Gestion des stocks, réapprovisionnement et logistique pour le stock central

Design Decisions

Voici une liste compréhensives des ADRs prises lors de la conception de ce système:

01 - Architectural decision record: Choix de l'architecture

Status

Accepté

Contexte

Pour rendre notre application plus scalable, il faut évoluer à partir de notre application de base. Cette évolution nous permet d'accepter plus de clients et de rendre notre application plus robuste et maintenable.

Des architectures possible sont les suivantes: 3-tier, n-tier, architecture hexagonale...
Notre application doit permettre une séparation claire des différentes couches (métier, base de données, interface utilisateur)

Décision

Architecture 3-tiers a été choisi comme architecture principale.

Justification

La raison pour laquelle une architecture 3-tiers a été choisi est parce que la séparation des responsabilités permet au systeme d'être plus flexible et maintenable. De plus, cette architecture permet une évolution vers n-tiers plus facile.

Conséquences

- Système plus modulaire et maintenable
- L'ajout de nouvelles fonctionnalités est facile

02 - Architectural decision record: Choix du cadriciel MVC

Status

Accepté

Contexte

Le système de magasin doit maintenant répondre aux besoins d'une entreprise qui possèdent cinq magasins, un centre de logistique et un maison mère. En conséquence, le système doit évoluer d'une architecture client-serveur vers une architecture 3-tiers. Pour y arriver, une cadriciel web doit être choisi pour implémenter une couche application puis présenter les cas d'utilisations. De nombreux cadriciels se présentent comme solution: FastAPI, Flask, Django

Décision

Django a été choisi comme cadriciel web pour faire évoluer l'application vers une architecture 3-tiers.

Justification

L'architecture 2-tiers présente de nombreux limites. Il existe un couplage enorme entre la base de donnees et puis le client. Pour effectuer un découplage, on peut se tourner vers une architecture 3-tiers. Django est un cadriciel MVT qui permet de faciliter l'architecture 3-tiers, en decouplant la couche présentation avec les templates, la couche business logic avec leur views et la couche métier avec les models. De plus, Django offre son propre cadriciel pour créer des APIs, ce qui serait idéal si l'on veut évoluer notre application vers cette direction ex. GraphQL ou API REST.

Conséquences

- Restreint à l'écosystème Django.
- Très lourd pour des petits projets
- Facilité vers un grand projet.

03 - Architectural decision record: Transition vers une architecture microservices

Status

Accepté

Contexte

L'architecture monolithique actuelle présente des limitations de performance sous charge élevée et des difficultés de maintenance avec l'augmentation de la complexité du système. La croissance en fonctionnalités deviendrait alors un problème.

Des architectures comme microservices, architecture orientée services, ou même une architecture orientée évènements.

Décision

Accepté pour microservices. Migrer vers une architecture microservices avec séparation des domaines métiers : Users, Products, Ecommerce, et Warehouse.

Justification

- Amélioration des performances sous charge (démontrée par les tests Locust)
- Scalabilité horizontale par service
- Séparation des responsabilités selon les domaines métiers
- Facilite le développement parallèle par équipes

Conséquences

Positives :

- Meilleure performance et résilience
- Évolutivité indépendante des services
- Isolation des pannes

Négatives :

- Complexité opérationnelle accrue (configuration de api gateway)
- Besoin d'outils de monitoring distribué

04 - Architectural decision record: Choix du cadriciel API pour les services indépendants

Status

Accepté

Contexte

Chaque microservice nécessite un framework web léger et performant pour exposer ses APIs REST. Le choix du framework impacte les performances, la maintenabilité et la facilité de développement.

Des options se présentent, comme Django REST Framework, Flask, ou même FastAPI.

Décision

Utiliser FastAPI comme framework web pour tous les microservices (Users, Products, Ecommerce, Warehouse).

Justification

- Performance élevée
- Documentation automatique des APIs (Swagger/OpenAPI)

- Syntaxe moderne et intuitive
- Développement rapide
- Compatible avec l'écosystème Python existant

Conséquences

Positives :

- APIs documentées automatiquement (Documentation OpenAPI déjà existante via url/docs)
- Développement rapide avec validation intégrée
- Excellentes performances
- Facilite les tests d'intégration

Négatives :

- Dépendance à un framework relativement récent
- Courbe d'apprentissage pour l'équipe (si non familière avec FastAPI)

Quality Requirements

N/A

Quality Scenarios

N/A

Risks and Technical Debts

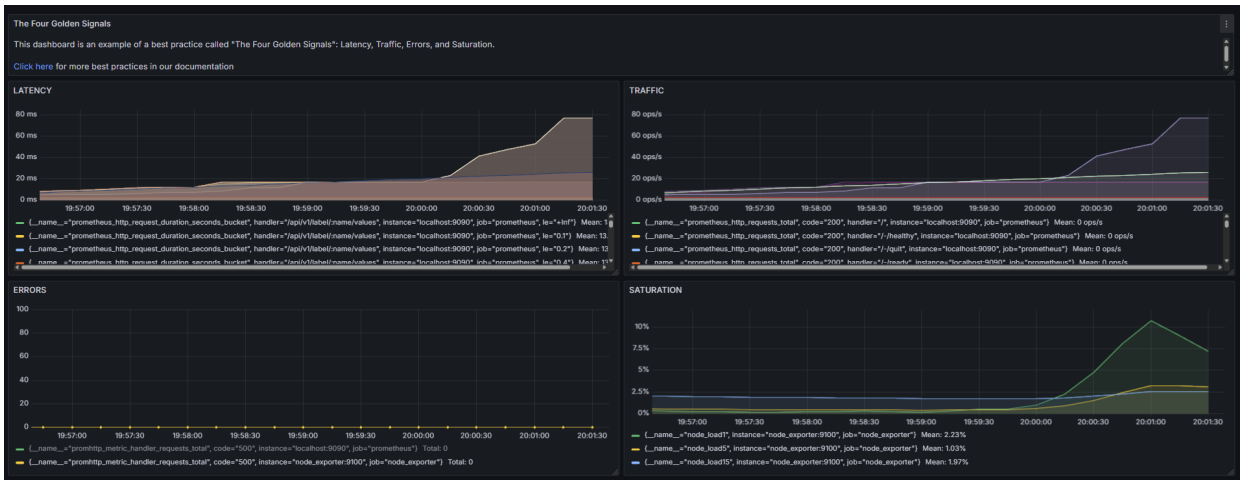
Risques	Description
Pas d'authentification	Le système ne requis pas d'authentification pour y accéder
Sécurité négligeable	Le système ne contient pas de mesures sécuritaire pour contrer les attaques
Pas de tests API	Absence de tests automatisés pour valider le fonctionnement des APIs des microservices, ce qui augmente le risque de régression lors des déploiements et complique la maintenance.
Pipeline CI/CD incomplet	Manque d'automatisation pour l'intégration continue et le déploiement continu
Pas d'authentification/d'autorisation sur les APIs	Les APIs des microservices ne requièrent pas de tokens d'authentification, créant une vulnérabilité de sécurité majeure où n'importe qui peut accéder aux services.
Pas de logging centralisé	Du logging locale sont dispersés entre les différents microservices sans centralisation (ex: Loki), rendant le debugging et la surveillance du système très difficiles.

Comparison between old architecture and new architecture

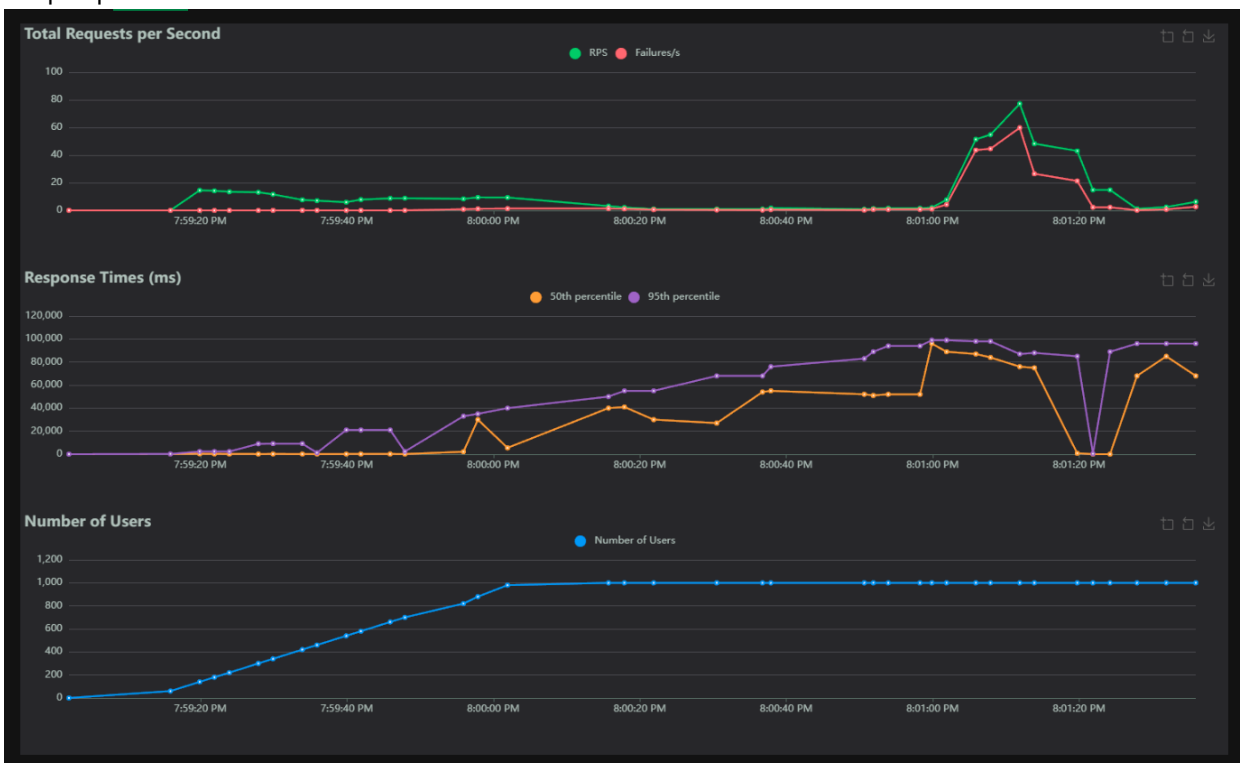
Pour savoir en quoi l'evolution d'architecture a eu un impact sur le système, un test de charge a été effectué à l'aide du cadriciel Locust et le fichier `locust.py`. Il a été effectuer à deux reprises: lors du laboratoire 4 pour son architecutre monolithique, et lors du laboratoire 5 pour son architecture microservices. Pour y arriver, `locust.py` crée une interface dans laquelle tu peux paramétrer le test de charge. Dans ce cas, 1000 users ont été crées avec un spawn-rate de 20, et ce, qui effectue tous des requêtes vers les API de produits et de stocks après près de 5 minutes. Voici les résultats:

Architecture monolithique

- Dashboard Grafana:



- Graphiques Locust:



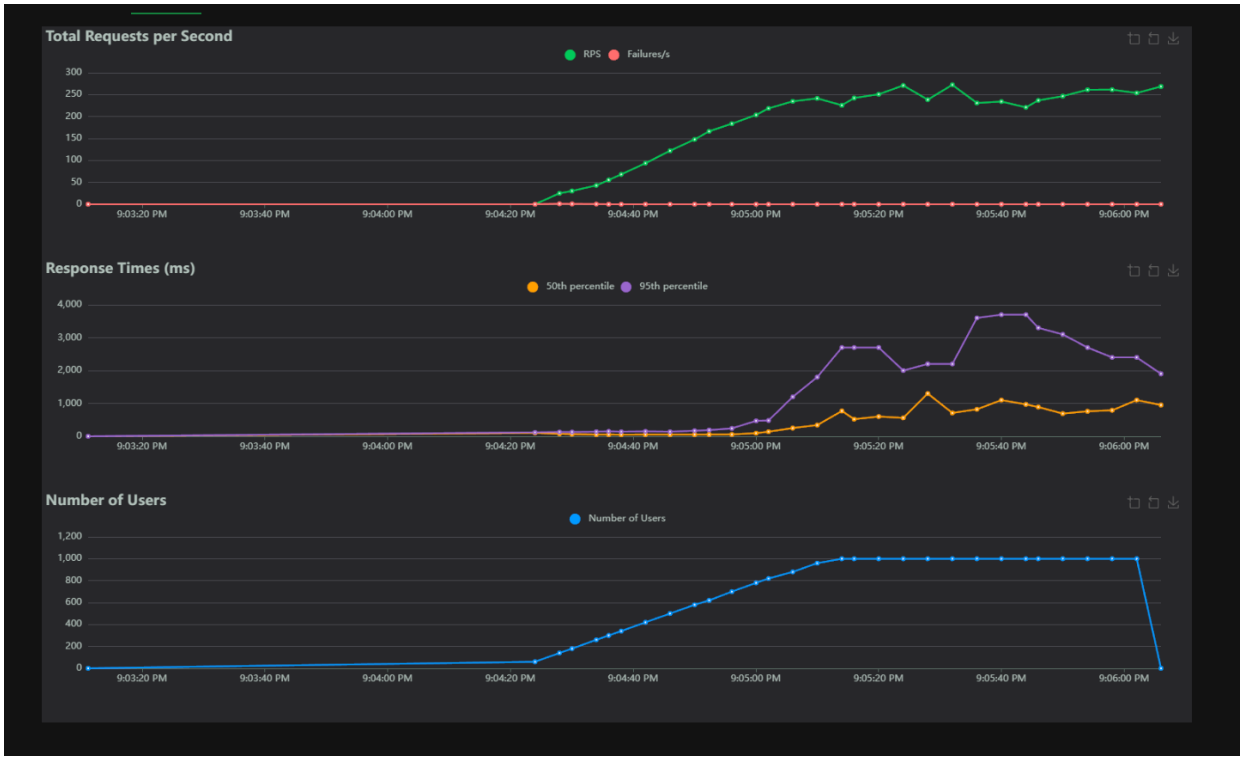
- Statistiques Locust:

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/	594	6	220	49000	69000	5154.28	3	72465	1292.81	0.6	0
GET	/api/v1/products	314	238	82000	98000	100000	72253.75	2112	101173	117181.17	1	0.4
GET	/api/v1/products/1	77	62	85000	99000	101000	73129.29	2253	100701	123962.53	0.2	0.1
GET	/api/v1/products/2	70	58	83000	97000	98000	74608.27	2232	97758	131804.26	0	0
GET	/api/v1/products/3	69	57	83000	98000	99000	73928.89	2245	99443	126797.55	0	0
GET	/api/v1/products/4	57	46	84000	99000	102000	73897.65	2244	101536	128377.28	0.2	0.1
GET	/api/v1/products/5	50	40	81000	96000	100000	74656.42	2062	99623	127261.8	0	0
GET	/api/v1/stocks/store/1	57	39	83000	96000	98000	71036.52	173	97713	109166.74	1	0.2
GET	/api/v1/stocks/store/2	77	54	80000	96000	97000	66902.1	77	96763	111888.18	1.1	0.8
GET	/api/v1/stocks/store/3	61	41	78000	91000	93000	66121.23	250	93236	107245.05	0.6	0.2
GET	/api/v1/stocks/store/4	66	53	82000	97000	99000	70166.38	189	98685	128086.48	0.9	0.7
GET	/api/v1/stocks/store/5	66	46	79000	94000	98000	64059.81	176	98454	108784.24	0.6	0.1
	Aggregated	1558	740	50000	96000	99000	46047.37	3	101536	74311.4	6.2	2.6

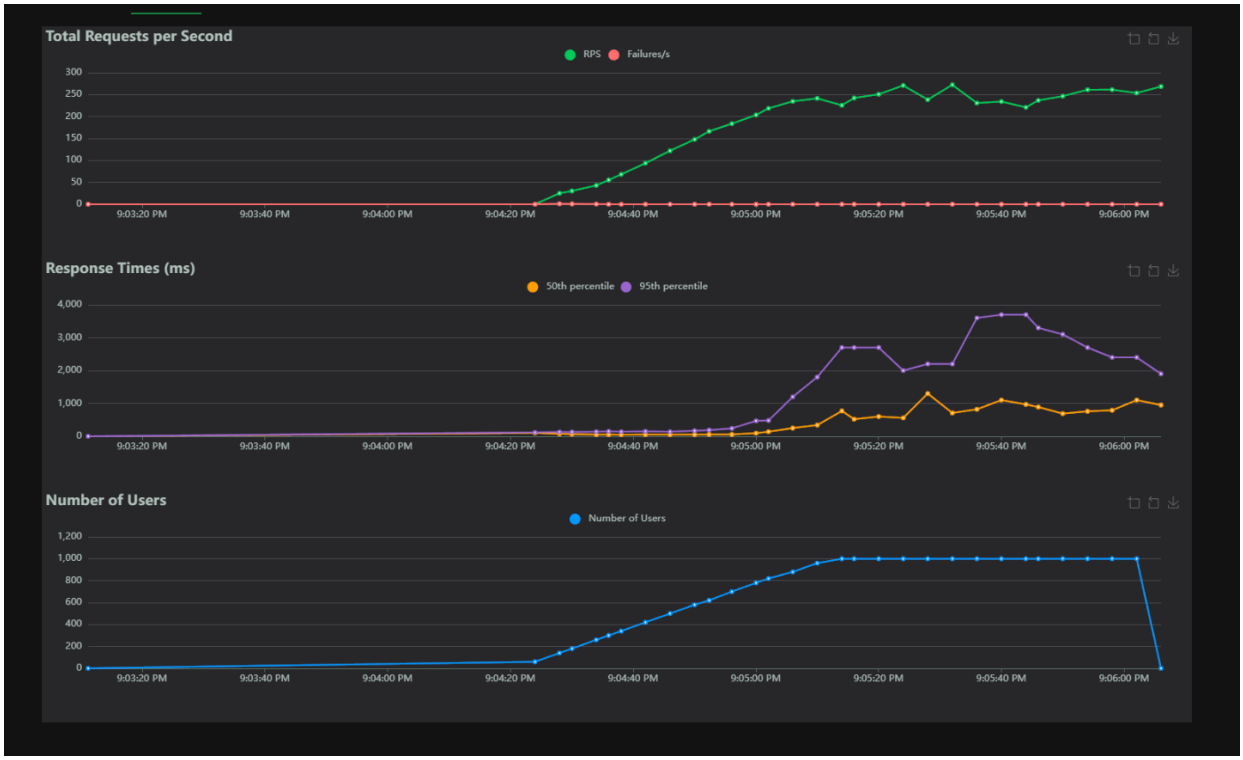
On peut voir que les requêtes vers l'API centrale de notre système monolithique est très limités. D'abord, on peut voir une augmentation majeure sur la latence des réponses. En effet, on voit que le CPU devient surchargé également avec le `node_load1`. De plus, nous voyons clairement le nombre de failures, car l'API n'arrive pas à répondre à toutes les requêtes, ce qui indiquent que l'application éprouve d'une difficulté lorsque la charge est élevée.

Architecture microservices

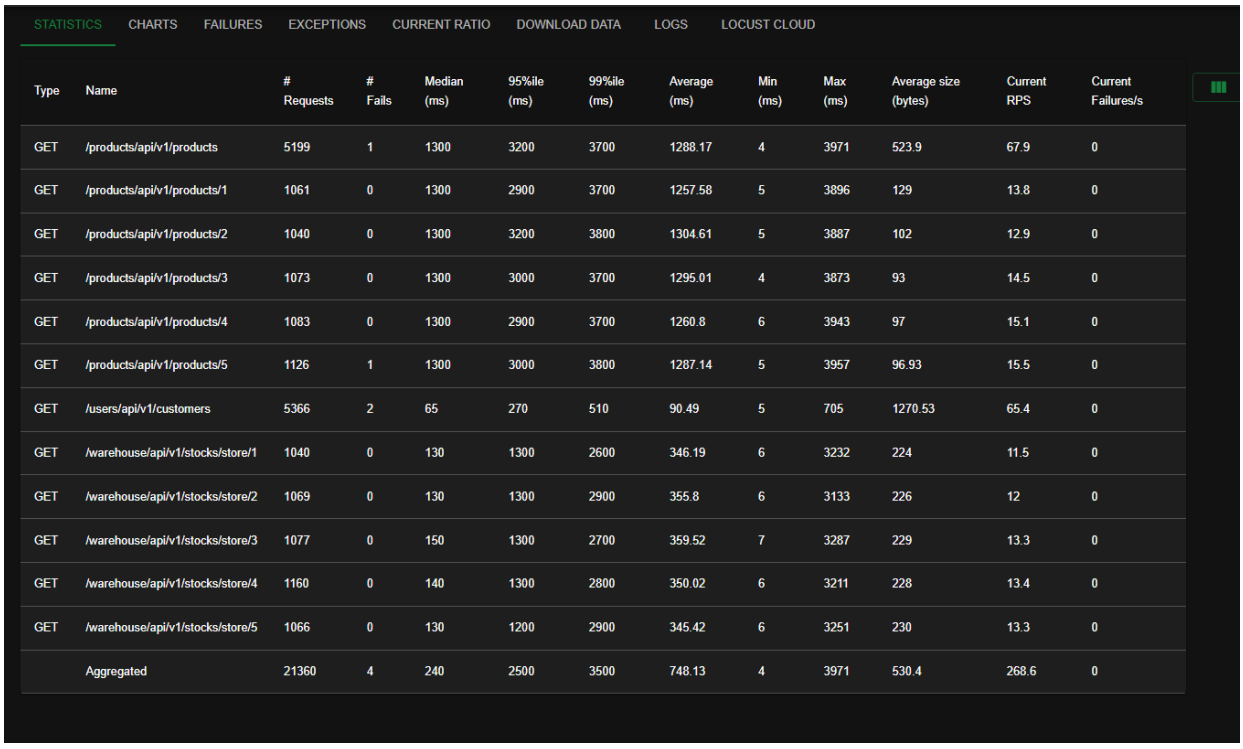
- Dashboard Grafana:



- Graphiques Locust:



- Statistiques Locust:



STATISTICS | CHARTS | FAILURES | EXCEPTIONS | CURRENT RATIO | DOWNLOAD DATA | LOGS | LOCUST CLOUD

Type	Name	# Requests	# Fails	Median (ms)	95%ile (ms)	99%ile (ms)	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	Current RPS	Current Failures/s
GET	/products/api/v1/products	5199	1	1300	3200	3700	1288.17	4	3971	523.9	67.9	0
GET	/products/api/v1/products/1	1061	0	1300	2900	3700	1257.58	5	3896	129	13.8	0
GET	/products/api/v1/products/2	1040	0	1300	3200	3800	1304.61	5	3887	102	12.9	0
GET	/products/api/v1/products/3	1073	0	1300	3000	3700	1295.01	4	3873	93	14.5	0
GET	/products/api/v1/products/4	1083	0	1300	2900	3700	1260.8	6	3943	97	15.1	0
GET	/products/api/v1/products/5	1126	1	1300	3000	3800	1287.14	5	3957	96.93	15.5	0
GET	/users/api/v1/customers	5366	2	65	270	510	90.49	5	705	1270.53	65.4	0
GET	/warehouse/api/v1/stocks/store/1	1040	0	130	1300	2600	346.19	6	3232	224	11.5	0
GET	/warehouse/api/v1/stocks/store/2	1069	0	130	1300	2900	355.8	6	3133	226	12	0
GET	/warehouse/api/v1/stocks/store/3	1077	0	150	1300	2700	359.52	7	3287	229	13.3	0
GET	/warehouse/api/v1/stocks/store/4	1160	0	140	1300	2800	350.02	6	3211	228	13.4	0
GET	/warehouse/api/v1/stocks/store/5	1066	0	130	1200	2900	345.42	6	3251	230	13.3	0
	Aggregated	21360	4	240	2500	3500	748.13	4	3971	530.4	268.6	0

On peut voir que les résultats du test de charge sur l'architecture microservices révèlent une amélioration substantielle des performances par rapport à l'architecture monolithique précédente. Le graphes sur le dashboard Grafana montrent que les métriques restent relativement stables (latence et saturation), ce qui démontre une stabilité lors d'une charge élevée. Le graphique dans Locust permet aussi de voir que le taux de réussite a drastiquement augmenté, car la distribution des requêtes vers les bonnes services permettent une meilleure traitement de requêtes.
On peut alors conclure que cette transition est une amélioration.

Glossary

N/A

Lien vers les autres labs

- Lien vers lab0: https://github.com/LojanArunasalam/LOG430-Lab0
- Lien vers lab1: https://github.com/LojanArunasalam/LOG430-Lab1
- Lien vers lab2: https://github.com/LojanArunasalam/LOG430-Lab2
- Lien vers lab3: https://github.com/LojanArunasalam/LOG430-Lab3
- Lien vers lab4: https://github.com/LojanArunasalam/LOG430-Lab4
- Lien vers lab5: https://github.com/LojanArunasalam/LOG430-Lab5