**RAPPORT DE PROJET DE FIN DE MODULE  
  
Projet : Reverse Logistic**Réalisé par :  
Amina ED-DAF  
Omar RIAFI  
Meriem ZIZOUANE  
  
Encadré par :  
Yamane Chemlal  
  
Date : Juin 2025

# Table des matières

Cliquez ici puis faites clic droit > Mettre à jour le champ pour actualiser la table.

**Sommaire**

1. Analyse du Domaine Métier (DDD Stratégique)

1.1. Contexte et Objectifs

1.2. Cartographie du Contexte (Context Map)

1.3. Langage Ubiquitaire

2. Conception Tactique (DDD Tactique) pour un Bounded Context Spécifique

2.1. Modèle de Domaine Détaillé : Gestion des Demandes de Retour

2.1.1. Agrégats

2.1.2. Entités

2.1.3. Objets Valeur

2.1.4. Services de Domaine

2.1.5. Événements de Domaine

2.2. Flux de Processus pour la Gestion des Demandes de Retour

2.3. Modèle de Domaine Détaillé : Réception et Traitement en Entrepôt (Warehouse Intake)

2.3.1. Agrégats

2.3.2. Entités

2.3.3. Objets Valeur

2.3.4. Services de Domaine

2.3.5. Événements de Domaine

2.4. Flux de Processus pour la Réception et le Traitement en Entrepôt

3. Proposition d’Architecture Microservices

3.1. Diagramme d’Architecture Générale

3.2. Description des Services

3.3. Communication Inter-Services

3.3.1. Communication Asynchrone (Event-Driven Architecture)

3.3.2. Communication Synchrone (API RESTful)

3.4. Service Discovery

3.5. API Gateway

3.6. Justification des Choix Architecturaux et de Conception DDD

# 1. Analyse du Domaine Métier (DDD Stratégique)

# 1.1. Contexte et Objectifs

Ce rapport détaille la conception d’un système de gestion des retours basé sur les principes du Domain-Driven Design (DDD) et une architecture de microservices. L’objectif principal est de créer un système robuste, évolutif et maintenable qui gère l’ensemble du processus de retour, de la demande initiale du client à la gestion des stocks et aux remboursements/échanges.

# 1.2. Cartographie du Contexte (Context Map)

L’analyse du domaine métier a révélé plusieurs sous-domaines clés, chacun représentant un aspect distinct du processus de gestion des retours. Ces sous-domaines sont modélisés comme des Bounded Contexts, définissant des limites claires pour le langage ubquitiaire et les modèles de domaine. Les principaux Bounded Contexts identifiés sont :

Gestion des Demandes de Retour (Return Request Management) : Ce contexte gère la création, la validation et l’approbation des demandes de retour initiées par les clients.

Réception et Traitement en Entrepôt (Warehouse Intake) : Ce contexte est responsable de la réception physique, de l’inspection et de la classification des produits retournés.

Gestion des Stocks (Inventory Management) : Ce contexte gère les niveaux de stock des produits, y compris les mises à jour suite aux retours et aux actions de remise en stock ou de mise au rebut.

Traitement des Remboursements et Échanges (Refund & Exchange Processing) : Ce contexte gère la logique de décision et l’exécution des remboursements ou des échanges pour les produits retournés.

Gestion des Utilisateurs (User Management) : Ce contexte gère les comptes des clients et du personnel, y compris l’authentification et les rôles.

Analyse des Retours (Returns Analytics) : Ce contexte est dédié à la collecte et à l’analyse des données relatives aux retours pour des rapports et des insights.

Ces Bounded Contexts interagissent via des événements de domaine, des commandes et des API, formant une carte de contexte qui illustre leurs relations et leurs dépendances. Le document Event Storming fourni a été essentiel pour identifier ces contextes et leurs interactions.

# 1.3. Langage Ubiquitaire

Un langage ubiquitaire cohérent a été établi pour chaque Bounded Context afin d’assurer une compréhension commune entre les experts du domaine et l’équipe de développement. Voici quelques termes clés et leur signification dans le contexte de la gestion des retours :

Demande de Retour (Return Request) : Une demande formelle d’un client pour retourner un ou plusieurs articles.

Article Retourné (Return Item) : Un article spécifique inclus dans une demande de retour.

Raison de Retour (Return Reason) : La justification fournie par le client pour le retour de l’article.

Produit Reçu (Received Product) : Un produit qui a été physiquement reçu à l’entrepôt suite à une demande de retour.

Classification du Produit (Product Grade) : L’évaluation de l’état d’un produit retourné (par exemple, LIKE\_NEW, NEEDS\_REFURBISHMENT, SALVAGE).

Article en Stock (Inventory Item) : Une instance d’un produit dans l’inventaire, avec son statut et son emplacement.

Remboursement (Refund) : Le processus de restitution de l’argent au client pour un article retourné.

Échange (Exchange) : Le processus de remplacement d’un article retourné par un autre article.

Événement de Domaine (Domain Event) : Un fait significatif qui s’est produit dans le domaine métier et qui est pertinent pour d’autres parties du système (par exemple, ProductGraded, ReturnRequestApproved).

Commande (Command) : Une instruction adressée au système pour effectuer une action (par exemple, UpdateInventoryWithGradedProduct, DecideRefundOrExchange).

Agrégat (Aggregate) : Un cluster d’objets de domaine traités comme une seule unité pour garantir la cohérence des données (par exemple, ReturnRequest, ReceivedProduct, InventoryItem).

Ce langage est utilisé de manière cohérente dans tous les diagrammes, le code et la documentation pour éviter toute ambiguïté.

# 2. Conception Tactique (DDD Tactique) pour un Bounded Context Spécifique

Pour illustrer la conception tactique du DDD, nous allons nous concentrer sur le Bounded Context Gestion des Demandes de Retour (Return Request Management). Ce contexte est crucial car il représente le point d’entrée de toutes les requêtes de retour client dans le système. Il est responsable de la création, de la validation et de l’approbation initiale des demandes de retour.

# 2.1. Modèle de Domaine Détaillé : Gestion des Demandes de Retour

Le modèle de domaine pour ce Bounded Context est centré autour de l’Agrégat ReturnRequest, qui encapsule la logique métier et assure la cohérence des données liées à une demande de retour spécifique. Cet agrégat est composé d’entités et d’objets valeur qui définissent collectivement l’état et le comportement d’une demande de retour.

# 2.1.1. Agrégats

ReturnRequest (Agrégat Racine):

Description: Représente une demande de retour unique initiée par un client. C’est l’agrégat racine, ce qui signifie que toutes les opérations sur les entités et objets valeur qu’il contient doivent passer par lui pour garantir la cohérence transactionnelle.

Attributs:

returnRequestId (UUID): Identifiant unique de la demande de retour.

customerId (UUID): Identifiant du client qui a initié la demande.

productId (UUID): Identifiant du produit concerné par la demande de retour.

reason: Un Objet Valeur ReturnReason décrivant la raison du retour.

requestedAt (LocalDateTime): Date et heure de la soumission de la demande.

status: Un ReturnStatus (PENDING, APPROVED, REJECTED) indiquant l’état actuel de la demande.

orderId (String): L’identifiant de la commande originale à laquelle le retour est lié.

trackingNumber (String): Numéro de suivi pour le retour (si applicable).

Comportements Métier (Méthodes):

createRequest(productId, reason, customerId): Crée une nouvelle demande de retour avec un statut PENDING.

approveRequest(approvedBy): Approuve la demande de retour, change son statut à APPROVED et enregistre qui l’a approuvée.

rejectRequest(rejectedBy): Rejette la demande de retour, change son statut à REJECTED et enregistre qui l’a rejetée.

# 2.1.2. Entités

ReturnRequest: Bien qu’étant l’agrégat racine, ReturnRequest est aussi une entité en soi, possédant une identité unique et un cycle de vie.

ReturnItem (Entité au sein de ReturnRequest):

Description: Représente un article individuel faisant partie d’une demande de retour. Une ReturnRequest peut contenir plusieurs ReturnItem.

Attributs:

id (String): Identifiant unique de l’article de retour.

productId (String): Identifiant du produit.

quantity (Int): Quantité de cet article à retourner.

purchaseDate (DateTime): Date d’achat de l’article.

condition (Condition): État de l’article (par exemple, ‘Bon’, ‘Endommagé’).

Comportements Métier: Aucun comportement complexe spécifique n’est défini pour ReturnItem en dehors de sa création et de sa gestion par l’agrégat ReturnRequest.

# 2.1.3. Objets Valeur

ReturnReason:

Description: Représente la raison spécifique pour laquelle un produit est retourné. C’est un objet valeur car il n’a pas d’identité propre et est défini uniquement par ses attributs.

Attributs:

reasonCode (String): Code unique de la raison (par exemple, DEFECTIVE, NOT\_SATISFIED, WRONG\_ITEM).

description (String): Description textuelle de la raison.

ReturnStatus:

Description: Un énumérateur représentant l’état d’une demande de retour.

Valeurs: PENDING, APPROVED, REJECTED.

# 2.1.4. Services de Domaine

Dans le contexte de la Gestion des Demandes de Retour, les services de domaine sont généralement moins prévalents car la plupart de la logique est encapsulée dans l’agrégat ReturnRequest. Cependant, si des opérations complexes impliquant plusieurs agrégats ou des dépendances externes étaient nécessaires, un service de domaine pourrait être introduit. Pour l’instant, les comportements sont gérés directement par l’agrégat.

# 2.1.5. Événements de Domaine

Les événements de domaine sont des faits du passé qui se sont produits dans le système et qui sont significatifs pour d’autres parties du domaine. Ils sont cruciaux pour la communication entre les Bounded Contexts dans une architecture de microservices.

ReturnRequestCreated:

Description: Émis lorsque une nouvelle demande de retour est soumise par un client.

Données: returnRequestId, productId, customerId, reason, requestedAt.

ReturnRequestApproved:

Description: Émis lorsque la demande de retour est approuvée par le personnel.

Données: returnRequestId, productId, customerId, approvedAt.

ReturnRequestRejected:

Description: Émis lorsque la demande de retour est rejetée par le personnel.

Données: returnRequestId, reason.

Ces événements sont consommés par d’autres Bounded Contexts, tels que Warehouse Intake (pour ReturnRequestApproved) ou Returns Analytics, pour déclencher des actions ou mettre à jour leurs propres modèles de lecture.

# 2.2. Flux de Processus pour la Gestion des Demandes de Retour

Le processus de gestion des demandes de retour suit les étapes suivantes :

Soumission de la Demande: Un client soumet une demande de retour via une interface utilisateur. Cette action déclenche une Commande (CreateReturnRequestCommand) qui est traitée par l’Application Service ReturnRequestService.

Création de l’Agrégat: Le ReturnRequestService utilise les données de la commande pour créer une nouvelle instance de l’agrégat ReturnRequest.

Validation et Persistance: L’agrégat ReturnRequest effectue ses propres validations métier. Une fois valide, il est persisté dans le référentiel de données du Bounded Context.

Émission d’Événement: Après la persistance réussie, l’agrégat ReturnRequest émet l’événement de domaine ReturnRequestCreated. Cet événement est ensuite publié dans un bus d’événements pour être consommé par d’autres services.

Approbation/Rejet: Le personnel de l’entrepôt ou du service client examine la demande de retour. Selon la décision, une Commande (ApproveReturnRequestCommand ou RejectReturnRequestCommand) est envoyée au ReturnRequestService.

Mise à Jour de l’Agrégat et Émission d’Événement: Le ReturnRequestService charge l’agrégat ReturnRequest existant, appelle la méthode appropriée (approveRequest ou rejectRequest), persiste les changements, et émet l’événement de domaine correspondant (ReturnRequestApproved ou ReturnRequestRejected).

Ce flux garantit que la logique métier est encapsulée dans l’agrégat et que les communications entre les contextes sont basées sur des événements, favorisant ainsi le découplage et la résilience de l’architecture.

# 2.3. Modèle de Domaine Détaillé : Réception et Traitement en Entrepôt (Warehouse Intake)

Le Bounded Context Réception et Traitement en Entrepôt est responsable de la gestion physique des produits retournés, de leur réception à leur classification. Il est étroitement lié au contexte de Gestion des Demandes de Retour car il consomme les événements d’approbation de ce dernier pour anticiper l’arrivée des produits.

# 2.3.1. Agrégats

ReceivedProduct (Agrégat Racine):

Description: Représente un produit spécifique qui a été reçu ou est attendu à l’entrepôt dans le cadre d’une demande de retour. Cet agrégat gère le cycle de vie du produit au sein de l’entrepôt, de sa réception à sa classification.

Attributs:

intakeId (UUID): Identifiant unique de l’entrée en entrepôt.

returnRequestId (UUID): Référence à la demande de retour associée.

productId (UUID): Identifiant du produit.

receivedAt (LocalDateTime, nullable): Date et heure de la réception physique du produit.

grade: Un Objet Valeur ProductGrade indiquant la classification du produit après inspection.

inspectionReport: Un Objet Valeur ProductInspectionReport contenant les détails de l’inspection.

status: Un IntakeStatus (EXPECTED, RECEIVED, GRADED, REVIEWED, SENT\_TO\_SUPPLIER) indiquant l’état actuel du produit dans le processus d’intake.

Comportements Métier (Méthodes):

expectProduct(expectedAt): Marque un produit comme attendu à une certaine date, généralement déclenché par l’événement ReturnRequestApproved.

receiveProduct(receivedAt): Enregistre la réception physique du produit à l’entrepôt.

gradeProduct(inspectionReport): Attribue une classification (ProductGrade) au produit après inspection, basée sur le rapport d’inspection.

reviewProduct(): Marque le produit comme ayant été examiné, prêt pour la prochaine étape (par exemple, envoi au fournisseur ou mise en stock).

sendToSupplier(): Marque le produit comme envoyé à un fournisseur pour réparation ou autre traitement.

# 2.3.2. Entités

ReceivedProduct: Comme ReturnRequest, ReceivedProduct est à la fois l’agrégat racine et une entité.

# 2.3.3. Objets Valeur

ProductGrade:

Description: Un énumérateur représentant la classification de l’état d’un produit retourné après inspection.

Valeurs: LIKE\_NEW, NEEDS\_REFURBISHMENT, SALVAGE.

ProductInspectionReport:

Description: Contient les détails de l’inspection d’un produit retourné.

Attributs:

damageLevel (int): Niveau de dommage (par exemple, échelle de 0 à 10).

functionalIssues (List): Liste des problèmes fonctionnels identifiés.

IntakeStatus:

Description: Un énumérateur représentant l’état d’un produit dans le processus de réception en entrepôt.

Valeurs: EXPECTED, RECEIVED, GRADED, REVIEWED, SENT\_TO\_SUPPLIER.

# 2.3.4. Services de Domaine

Dans ce contexte, les services de domaine sont principalement utilisés pour orchestrer des actions complexes qui ne peuvent pas être encapsulées dans un seul agrégat ou qui impliquent des interactions avec des systèmes externes. Pour l’instant, la logique principale est gérée par l’agrégat ReceivedProduct.

# 2.3.5. Événements de Domaine

Les événements de domaine émis par ce Bounded Context sont cruciaux pour informer les autres services de l’avancement du traitement des produits retournés.

ProductExpected:

Description: Émis lorsqu’un produit est attendu à l’entrepôt, généralement après l’approbation d’une demande de retour.

Données: intakeId, returnRequestId, productId, expectedAt.

ProductReceived:

Description: Émis lorsque le produit est physiquement reçu à l’entrepôt.

Données: intakeId, returnRequestId, productId, receivedAt.

ProductGraded:

Description: Émis après l’inspection et la classification du produit.

Données: intakeId, returnRequestId, productId, grade.

ProductReviewed:

Description: Émis lorsque le produit a été examiné et est prêt pour la prochaine étape.

Données: intakeId.

ProductSentToSupplier:

Description: Émis lorsque le produit est envoyé à un fournisseur externe.

Données: intakeId.

Ces événements sont consommés par des services comme Inventory Management (pour ProductGraded) ou Returns Analytics pour mettre à jour leurs propres modèles ou déclencher des actions.

# 2.4. Flux de Processus pour la Réception et le Traitement en Entrepôt

Le processus de réception et de traitement en entrepôt suit les étapes suivantes :

Anticipation de la Réception: Lorsque l’événement ReturnRequestApproved est émis par le contexte Gestion des Demandes de Retour, le WarehouseService (Application Service) dans ce contexte consomme cet événement et crée une instance ReceivedProduct avec le statut EXPECTED.

Réception Physique: Le personnel de l’entrepôt reçoit le produit et enregistre sa réception. Une Commande (ReceiveProductCommand) est envoyée au WarehouseService.

Mise à Jour de l’Agrégat et Émission d’Événement: Le WarehouseService charge l’agrégat ReceivedProduct, appelle la méthode receiveProduct(), persiste les changements, et émet l’événement ProductReceived.

Inspection et Classification: Le produit est inspecté. Une Commande (GradeProductCommand) est envoyée au WarehouseService avec les détails de l’inspection.

Mise à Jour de l’Agrégat et Émission d’Événement: Le WarehouseService appelle la méthode gradeProduct() sur l’agrégat ReceivedProduct, persiste les changements, et émet l’événement ProductGraded. Cet événement est particulièrement important car il déclenche des actions dans d’autres contextes, comme la mise à jour de l’inventaire ou la décision de remboursement/échange.

Examen et Actions Suivantes: Le produit peut être examiné (reviewProduct()) avant d’être envoyé au fournisseur (sendToSupplier()) ou de passer à la gestion des stocks. Chaque action déclenche des mises à jour de statut et l’émission d’événements correspondants.

Ce flux illustre comment les événements de domaine facilitent la communication et la coordination entre les différents Bounded Contexts, permettant une architecture de microservices découplée et réactive.

# 3. Proposition d’Architecture Microservices

L’adoption d’une architecture microservices pour le système de gestion des retours est une démarche stratégique, dictée par la complexité du domaine métier et le besoin inhérent de flexibilité, de scalabilité et de résilience. Cette approche permet de décomposer le système monolithique traditionnel en un ensemble de services autonomes, chacun responsable d’un Bounded Context spécifique, favorisant ainsi un développement, un déploiement et une maintenance indépendants.

# 3.1. Diagramme d’Architecture Générale

L’architecture proposée s’articule autour des Bounded Contexts identifiés précédemment, chacun étant implémenté comme un microservice distinct. Cette séparation est illustrée par le diagramme d’architecture conceptuel ci-dessous, qui met en évidence les principaux services et leurs interactions.

[Diagramme d’architecture conceptuel - image\_png(2).png]

Figure 1: Vue d’ensemble de l’architecture microservices, délimitant les Bounded Contexts en services autonomes.

Comme le montre la Figure 1, chaque boîte représente un microservice, correspondant à un Bounded Context. Les flèches indiquent les flux de communication et les dépendances entre ces services. Cette structure favorise un couplage lâche et une forte cohésion interne à chaque service, principes fondamentaux des microservices.

# 3.2. Description des Services

Chaque microservice est conçu pour être autonome, possédant sa propre base de données (si nécessaire) et encapsulant la logique métier de son Bounded Context. Voici une description détaillée des microservices clés :

Microservice de Gestion des Demandes de Retour (Return Request Management Service):

Responsabilités: Gère le cycle de vie complet des demandes de retour, de la soumission à l’approbation ou au rejet. Il est le point d’entrée pour les clients souhaitant initier un retour.

API: Expose des points de terminaison RESTful pour la création, la consultation, l’approbation et le rejet des demandes de retour.

Communication: Émet des événements de domaine (ReturnRequestCreated, ReturnRequestApproved, ReturnRequestRejected) pour informer les autres services des changements d’état des demandes.

Microservice de Réception et Traitement en Entrepôt (Warehouse Intake Service):

Responsabilités: Prend en charge la gestion physique des produits retournés, y compris la réception, l’inspection et la classification. Il est le gardien de l’état physique des articles retournés.

API: Fournit des points de terminaison pour enregistrer la réception des produits, leur classification et les rapports d’inspection.

Communication: Consomme l’événement ReturnRequestApproved pour anticiper l’arrivée des produits. Émet des événements comme ProductReceived, ProductGraded pour informer de l’avancement du traitement en entrepôt.

Microservice de Gestion des Stocks (Inventory Management Service):

Responsabilités: Maintient l’état de l’inventaire des produits. Il est mis à jour lorsque des produits sont remis en stock ou mis au rebut suite à un retour.

API: Offre des points de terminaison pour ajuster les quantités en stock, marquer des articles comme mis au rebut, ou consulter l’état de l’inventaire.

Communication: Consomme l’événement ProductGraded (en particulier pour les produits LIKE\_NEW ou SALVAGE) pour mettre à jour l’inventaire. Émet des événements comme ItemRestocked, ItemMarkedAsSalvaged.

Microservice de Traitement des Remboursements et Échanges (Refund & Exchange Processing Service):

Responsabilités: Contient la logique métier complexe pour décider et exécuter les remboursements ou les échanges. C’est le centre de décision financière et logistique post-inspection.

API: Expose des points de terminaison pour initier des remboursements ou des échanges, et pour consulter leur statut.

Communication: Consomme les événements ProductGraded et InventoryUpdated pour prendre des décisions éclairées. Émet des événements comme RefundIssued, ExchangeShipped.

Microservice de Gestion des Utilisateurs (User Management Service):

Responsabilités: Gère les informations des utilisateurs (clients et personnel), l’authentification et l’autorisation. Il est le point de vérité pour l’identité des acteurs du système.

API: Fournit des points de terminaison pour l’enregistrement, la connexion, la gestion des rôles et la récupération des profils utilisateurs.

Communication: Peut émettre des événements comme UserRegistered, UserLoggedIn.

Microservice d’Analyse des Retours (Returns Analytics Service):

Responsabilités: Collecte et agrège les données de tous les autres services pour générer des rapports et des tableaux de bord analytiques. C’est un service de lecture pure, optimisé pour la performance des requêtes.

API: Offre des points de terminaison pour accéder aux données agrégées et aux rapports d’analyse.

Communication: S’abonne à tous les événements de domaine pertinents (ReturnRequestSubmitted, ProductGraded, RefundIssued, etc.) pour construire ses modèles de lecture.

# 3.3. Communication Inter-Services

La communication entre les microservices est un aspect fondamental de cette architecture. Nous privilégions une approche hybride, combinant la communication asynchrone basée sur les événements et la communication synchrone via des API RESTful, en fonction des besoins spécifiques de chaque interaction.

# 3.3.1. Communication Asynchrone (Event-Driven Architecture)

La communication asynchrone est le pilier de notre architecture, favorisant le découplage et la résilience. Elle est principalement mise en œuvre via un Broker de Messages (par exemple, Apache Kafka, RabbitMQ). Lorsqu’un microservice émet un événement de domaine, il le publie sur un topic ou une file d’attente dédiée. Les microservices intéressés s’abonnent à ces topics et consomment les événements pour réagir en conséquence.

[Diagramme de flux d’événements - image\_png(3).png]

Figure 2: Illustration du flux d’événements entre les microservices, démontrant la communication asynchrone.

La Figure 2 illustre ce mécanisme : un événement comme ProductGraded est émis par le Warehouse Intake et consommé par Inventory Management et Auto Action Request (qui fait partie de Refund & Exchange Processing). Cela permet aux services de réagir aux changements sans avoir de dépendance directe et synchrone les uns envers les autres.

Avantages:

Découplage: Les services n’ont pas besoin de connaître l’existence ou l’emplacement des autres services. Ils publient et consomment des événements de manière indépendante.

Résilience: Si un service consommateur est temporairement indisponible, les événements sont mis en file d’attente et traités une fois le service rétabli.

Scalabilité: Facilite l’ajout de nouveaux consommateurs d’événements sans affecter les services existants.

Auditabilité: Le journal des événements fournit un historique complet des actions du système.

# 3.3.2. Communication Synchrone (API RESTful)

Pour les interactions nécessitant une réponse immédiate ou une requête directe d’informations, les microservices exposent des API RESTful. Ces API sont utilisées pour des requêtes de données spécifiques ou pour déclencher des actions qui ne nécessitent pas une propagation asynchrone.

Exemples d’utilisation:

Un service de frontend interrogeant le Return Request Management Service pour afficher le statut d’une demande de retour.

Le Refund & Exchange Processing Service interrogeant le Inventory Management Service pour vérifier la disponibilité d’un produit avant un échange.

# 3.4. Service Discovery

Dans un environnement microservices, les services sont dynamiques : ils peuvent être démarrés, arrêtés ou mis à l’échelle. Le Service Discovery est essentiel pour permettre aux services de se trouver et de communiquer entre eux sans avoir de connaissances préalables de leurs adresses réseau. Nous proposons l’utilisation d’un registre de services (par exemple, Eureka, Consul, Kubernetes Service Discovery).

Mécanisme: Chaque microservice s’enregistre auprès du registre de services au démarrage. Lorsqu’un service A a besoin de communiquer avec un service B, il interroge le registre pour obtenir l’adresse réseau de B. Un équilibreur de charge côté client (Client-Side Load Balancer) peut être utilisé pour distribuer les requêtes entre les instances disponibles d’un service.

# 3.5. API Gateway

L’API Gateway agit comme un point d’entrée unique pour toutes les requêtes externes (clients web, applications mobiles, autres systèmes). Elle offre une façade unifiée pour les microservices sous-jacents, simplifiant ainsi l’interaction pour les consommateurs et fournissant des fonctionnalités transversales.

Fonctionnalités Clés:

Routage des Requêtes: Dirige les requêtes entrantes vers le microservice approprié.

Agrégation de Requêtes: Peut agréger les réponses de plusieurs microservices pour une seule requête client.

Authentification et Autorisation: Gère la sécurité en validant les jetons d’authentification et en appliquant les politiques d’autorisation avant de router les requêtes.

Limitation de Débit (Rate Limiting): Protège les microservices contre les surcharges en limitant le nombre de requêtes.

Mise en Cache: Peut mettre en cache les réponses pour améliorer les performances.

Transformation de Protocole: Peut traduire les requêtes entre différents protocoles si nécessaire.

# 3.6. Justification des Choix Architecturaux et de Conception DDD

Les choix architecturaux et de conception adoptés sont profondément enracinés dans les principes du DDD et les meilleures pratiques des microservices, visant à construire un système qui n’est pas seulement fonctionnel, mais aussi durable et adaptable.

# 3.6.1. Alignement avec le DDD

Bounded Contexts comme Microservices: Chaque microservice est une implémentation directe d’un Bounded Context. Cette correspondance garantit que chaque service possède un modèle de domaine clair et cohérent, avec un langage ubiquitaire bien défini. Cela réduit la complexité et les ambiguïtés qui surviennent souvent dans les systèmes monolithiques où différents concepts peuvent être mélangés.

Agrégats comme Unités de Cohérence: L’utilisation d’agrégats (comme ReturnRequest, ReceivedProduct, InventoryItem) assure la cohérence transactionnelle et l’intégrité des données au sein de chaque microservice. Toutes les modifications d’un agrégat passent par sa racine, ce qui simplifie la gestion des invariants métier.

Événements de Domaine pour l’Intégration: Les événements de domaine sont le mécanisme privilégié pour la communication inter-services. Ils reflètent les faits métier significatifs et permettent un découplage temporel et spatial entre les services. Cette approche événementielle est naturelle pour le DDD, car elle modélise la façon dont les événements se produisent dans le monde réel et permet aux services de réagir de manière autonome.

Services de Domaine et Services d’Application: La distinction claire entre les services de domaine (logique métier sans état) et les services d’application (orchestration des cas d’utilisation) permet de maintenir une architecture propre et de séparer les préoccupations. Les services de domaine encapsulent des opérations métier complexes qui ne s’intègrent pas naturellement dans un agrégat, tandis que les services d’application coordonnent les interactions entre les agrégats et les services de domaine pour accomplir une tâche spécifique.

# 3.6.2. Bénéfices de l’Architecture Microservices

Scalabilité Indépendante: Chaque microservice peut être mis à l’échelle indépendamment en fonction de sa charge. Par exemple, le Warehouse Intake Service pourrait nécessiter plus de ressources pendant les périodes de forte activité de retour, sans affecter la scalabilité du User Management Service.

Résilience Accrue: La défaillance d’un microservice n’entraîne pas nécessairement la défaillance de l’ensemble du système. Les mécanismes de communication asynchrone et les circuits de coupure (circuit breakers) peuvent isoler les pannes, permettant au reste du système de continuer à fonctionner.

Développement et Déploiement Indépendants: Les équipes peuvent développer, tester et déployer des microservices de manière autonome, ce qui accélère le cycle de développement et réduit les dépendances entre les équipes. Cela favorise également l’adoption de pipelines CI/CD robustes pour chaque service.

Flexibilité Technologique: Chaque microservice peut être développé en utilisant la technologie la plus appropriée à ses besoins spécifiques (langage de programmation, base de données, frameworks). Cela permet aux équipes de choisir les meilleurs outils pour le travail, sans être contraintes par un stack technologique monolithique.

Maintenance Simplifiée: La taille réduite et la portée limitée de chaque microservice facilitent la compréhension, la modification et la maintenance du code. Les bugs sont plus faciles à isoler et à corriger.

En combinant les principes du DDD avec une architecture microservices, nous créons un système qui non seulement modélise fidèlement le domaine métier, mais est également agile, robuste et prêt à évoluer avec les besoins futurs de l’entreprise.