Cahier des Charges de spécifications Technique Final

I. Intégration de Spring Cloud Config dans les microservices du projet G-Shop

Objectif

Permettre à tous les microservices de centraliser leurs fichiers de configurations dans le dépôt Git **https: //github.com/ManichUser/G-Shop.git**, dans le dossier **gshopconfig** , à l'aide d'un serveur Spring Cloud Config standard

Serveur de configuration commun

Tous les microservices se connecteront au serveur Spring Cloud Config configuré comme suit :

Configuration du **application.properties** du serveur config (déjà en place)

```
spring.application.name=service-config
server.port=8888

spring.cloud.config.server.git.uri=https://github.com/ManichUser/G-Shop.git
spring.cloud.config.server.git.search-paths=gshop-config
spring.cloud.config.server.git.default-label=main
spring.cloud.config.server.git.clone-on-start=true
```

Pour chaque développeur de microservice

Voici les étapes obligatoires pour intégrer le serveur de configuration :

1- **Ajouter les dépendances dans le pom.xml**

```xml
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-config</artifactId>
        </dependency>
    </dependencies>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>${spring-cloud.version}</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
```

2- **Renommer application.properties**

Dans **src\main\ressource\application.properties**

```
spring.application.name=panier-service
spring.config.import=optional:configserver:http://localhost:8888

```

Le nom (**spring.application.name**) doit exactement correspondre au nom du fichier .properties dans le dépôt de config.

Supprimer les configurations sensibles du application.properties.

Ne pas laisser
Créer ou modifier le fichier de configuration dans le dépôt Git distant

Dépôt : **https: //github. com/ManichUser/G-Shop. git**

Dossier : **gshop-config/**

Nom du fichier : **<nom-du-service>.properties**

Exemple : **gshop-config/products-service.properties**

```
spring.application.name=products-service
spring.datasource.url=jdbc:postgresql://localhost:5432/produitsdb
spring.datasource.username=postgres
spring.datasource.password=etoile

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true

spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect

server.port=8082

```

Commit et push du fichier distant

```
$ git clone https://github.com/ManichUser/G-Shop.git
cd G-Shop/gshop-config
# Créer ou modifier le fichier .properties
git add .
git commit - m " Ajout config pour products-service"
git push origin main
```

3- Redemarrer le microservice

Il doit changer sa configuration automatiquement depuis :

http: //localhost: 8888/products-service/default

Résultat attendu

- Microservice demarre sans erreur.
- Ne contient pas de configuration sensible localemnt.
- Peut modifier ses propriéets via Git uniquement(centralisation)
- Serveur Config fonctionne comme source unique de verité.

II. Spécification technique : microservice de paiement monolithique « paiement-service »

1- Contexte

Le microservice de paiement est conçu pour gérer l'ensemble des transactions financières entre plusieurs acteurs (clients, vendeurs, plateforme) dans un environnement microservices. Il agit comme un point centralisé pour initier, suivre et finaliser les paiements et remboursements, tout en intégrant les fournisseurs de paiement externes comme Orange Money, MTN Mobile Money et une passerelle de paiement par carte bancaire.

Objectifs

- Fournir des API REST réactives pour initier les transactions, vérifier leur statut et recevoir les notifications des fournisseurs externes.
- Gérer la logique métier complexe des différents types de transactions (collecte, décaissement unique, décaissement multiple).
- Assurer l'intégration fluide avec les fournisseurs de paiement via des services spécifiques.
- Garantir la traçabilité, la robustesse et la gestion des erreurs grâce à une base de données et à une couche de logging.
- Être capable de fonctionner dans une architecture réactive, non bloquante, pour une meilleure scalabilité et réactivité.

2- Architecture et Technologie

- Framework : Spring Boot avec le starter WebFlux, pour une architecture réactive et non bloquante.
- API Web : Contrôleurs exposant des endpoints REST, basés sur Mono et Flux.
- Appels externes : Utilisation de WebClient pour communiquer avec Orange Money, MTN MoMo et la passerelle bancaire, dans un modèle réactif.
- Base de données : Stockage des transactions, sous-transactions, statuts et logs. Utilisation probable de Spring Data réactif.
- Gestion des événements : Envoi d'événements vers un bus d'événements interne pour informer les autres microservices de l'évolution des transactions.
- Gestion des erreurs : Exceptions personnalisées et journalisation pour faciliter le suivi et le débogage.

3- Fonctionnalités Clés

API Endpoints

- POST /transactions/initier : Point d'entrée pour demander l'initiation d'une transaction. Cette requête inclut le montant, la devise, le type de transaction (collecte,

décaissement unique, décaissement multiple), la méthode de paiement et les informations des acteurs concernés (sources et cibles).

- GET /transactions/{transaction_id}/statut : Permet de récupérer le statut en temps réel d'une transaction, avec les détails pertinents (montant, type, statut).
- POST /callbacks/orange-money : Endpoint dédié pour recevoir les notifications d'Orange Money, garantissant la mise à jour du statut des transactions liées.
- POST /callbacks/mtn-momo : Endpoint pour les notifications de MTN Mobile Money.
- POST /callbacks/carte : Endpoint pour recevoir les retours de la passerelle de paiement par carte.

Logique Métier des Transactions

- Collecte vers plateforme : Plusieurs clients effectuent des paiements vers la plateforme. Chaque sous-transaction est gérée individuellement selon la méthode de paiement choisie. La plateforme attend les callbacks pour valider chaque paiement.
- Décaissement vers acteur unique : La plateforme verse des fonds à un unique bénéficiaire. La méthode de paiement est choisie selon le type (Orange Money, MTN MoMo, virement bancaire).
- Décaissement vers acteurs multiples : La plateforme rembourse plusieurs clients simultanément, chaque remboursement étant traité avec la méthode adéquate. Les statuts sont mis à jour après réception des callbacks.

Gestion des Intégrations Externes

Les services OrangeMoneyService et MTNMoMoService utilisent le client HTTP réactif WebClient pour envoyer les demandes de paiement et recevoir les réponses. Chaque intégration respecte le protocole et les spécificités de l'API externe.

Base de données et suivi

Chaque transaction et sous-transaction est enregistrée avec un identifiant unique, un statut (EN_ATTENTE, SUCCES, ECHEC, REMBOURSE), un horodatage et les détails de paiement. Les mises à jour de statut se font à partir des callbacks ou des réponses des fournisseurs.

Gestion des erreurs et résilience

Les erreurs lors des appels externes sont capturées et des exceptions personnalisées sont levées pour assurer une gestion claire des échecs. La journalisation détaillée permet de tracer les opérations et facilite le diagnostic.

Le microservice de paiement sera une application réactive, robuste et modulaire, capable de gérer les différents flux financiers en intégrant des fournisseurs externes de manière asynchrone et non bloquante. Cette conception garantit une bonne scalabilité et une meilleure expérience utilisateur grâce à la rapidité de traitement et au suivi précis des transactions.

III. Spécification techniques du microservice : « vinted-vente »

Plateforme de vente de produits d'occasion entre particuliers (type Vinted)

Date : 02/06/2025

1- Objectif du microservice

Le microservice `vinted-vente` a pour objectif de gérer les produits mis en vente par les utilisateurs. Il permet :

- L'ajout d'un produit par un utilisateur
- La consultation des produits (par statut, par utilisateur, par catégorie, etc.)
- La recherche textuelle partielle
- La gestion des statuts (`DISPONIBLE`, `RÉSERVÉ`, `VENDU`, etc.)

2- Technologies utilisées

Java 17+, Spring Boot 3.x, Data MongoDB 6.x, Maven, Lombok, Docker(optionnel)

3- Architecture

Le microservice « vinted-vente » suit une architecture 3 couches : Controller, Service, Repository.

4- Structure des packages

com.bytemasterming.vinted vente

```
├── controller
├── model
├── repository
├── service
├── dto
└── config
```

5- Modèle de données principal

ProductVinted contient: id, idUser, productName, description, imageUrl, category, price, status
Enum ProductStatus : DISPONIBLE, RESERVE, VENDU

6- Points d'entrée (API REST)

- POST /api/vinted : Ajouter un produit
- GET /api/vinted/disponible : Produits disponibles
- GET /api/vinted/utilisateur/{idUser}
- GET /api/vinted/utilisateur/{idUser}/status/{status}
- GET /api/vinted/categorie/{categorie}
- GET /api/vinted/recherche?nomProduit=xxx

- GET /api/vinted

7- Flux metier

Ajout d'un produit : POST → statut DISPONIBLE → sauvegarde
Consultation : requêtes avec filtres

8- Sécurité

Vérification de l'utilisateur via JWT avec auth-service.

9- Tests

Tests unitaires avec JUnit et Mockito, tests d'intégration avec @SpringBootTest

10- Déploiement

Port : 8084, MongoDB local ou cloud, configuration via application.properties

11- Améliorations futures

- Upload réel d'images
- Pagination / Tri
- Microservice de réservation
- Notifications en temps réel via Kafka
- Swagger pour documentation automatique

IV. Spécification techniques du microservice : « authentification»

1- Objectif principal du projet

- C'est un système complet d'authentification et de gestion des rôles pour une plateforme de vente locale, où des utilisateurs peuvent :
- S'inscrire et se connecter via JWT (JSON Web Token),
- Avoir un ou plusieurs rôles (USER, PRODUCER, ADMIN),
- Passer du rôle d'utilisateur à celui de fournisseur en remplissant un formulaire détaillé,
- Être géré (activé, désactivé, supprimé) par un administrateur.
- Le backend est développé avec Spring Boot et respecte les standards modernes de sécurité et d'architecture REST.

2- Modules et fonctionnalités

a) Authentification

- Inscriptions via /api/auth/register.
- Connexion via /api/auth/login.
- Génération d'un token JWT à chaque connexion.
- Vérification du token à chaque requête protégée via un filtre JWT.

b) Gestion des rôles

- Rôle par défaut à l'inscription : USER.
- Passage au rôle PRODUCER via /api/user/devenir-fournisseur avec un formulaire complet.
- Possibilité de switcher de rôle actif via /api/auth/switch-role ou /api/user/switch-role.
- Vérification des rôles avec @PreAuthorize("hasRole('X')").

c) Rôle de Producteur (fournisseur)

- Informations d'entreprise enregistrées (nom, RC, adresse, contacts, banque mobile, produits, zones de livraison...).
- Accès aux endpoints réservés aux producteurs (/api/producer/mes-produits).

d) Administration

L'admin peut :

- Voir les logs d'audit (/api/admin/audit-logs),
- Activer/désactiver/supprimer des comptes utilisateurs,
- Toutes les actions sont loggées dans une entité AuditLog.

e) Audit logging

Toutes les actions administratives sont journalisées avec :qui a fait l'action, sur quel utilisateur, date/heure, détail de l'action.

f) Sécurité Spring Security + JWT

- Authentification sans session (stateless),
- Filtrage des requêtes via JwtFilter injecté dans le SecurityFilterChain,
- Gestion personnalisée des erreurs de token (401 Unauthorized).

g) Validation et sécurité métier

- Numéros MTN/ORANGE vérifiés via PhoneValidator,
- Zones de livraison valides restreintes à une liste (VALID_AREAS),
- Vérification des rôles disponibles avant changement.

h) Configuration CORS et Swagger

- Accès autorisé depuis un frontend React (localhost:3000),
- Documentation de l'API générée avec Swagger (annotations + SwaggerConfig).

i) Structure technique

- entities/: User, ProducerDetails, AuditLog, Role (enum)
- dto/ : Objets pour recevoir les requêtes (RegisterRequest, LoginRequest, ProducerRegisterRequest)
- security/ : Gestion du JWT (JwtUtil, JwtFilter)
- service/ : AuthService = cœur de la logique métier (inscription, login, changement de rôle, conversion en producteur)
- controller/ : AuthController : login/register/switch
- UserController : actions utilisateur
- AdminController : actions admin
- ProducerController : actions producteur

Ajouter les entités de produits : la route /api/producer/mes-produits est un placeholder. Il faut gérer des produits réels liés à un producteur.

Ajouter la validation avec @Valid et @NotBlank dans les DTO pour une meilleure sécurité.

Créer le frontend (React ou autre) pour :

- L'inscription et la connexion,
- L'interface admin (gestion des utilisateurs),
- L'espace fournisseur (produits, zones…).
- Ajouter un système de notification ou d'email,
- Mettre en place un système de mot de passe oublié,
- Gérer la pagination et la recherche dans la liste des utilisateurs/produits.

V. Spécifications techniques du microservice : « Commande »

1- Objectif du microservice

Le microservice commande gère la participation d'un utilisateur à un achat groupé d'un produit unique, ce qui correspond à un "panier" pour un seul produit. Chaque commande est représentée par un document dans MongoDB avec les informations : userId , produitId , quantite , etat .

2- Architecture logicielle

Le microservice suit les bonnes pratiques DDD (Domain Driven Design) et Clean Architecture, avec des couches bien séparées :

```
commande-service/
├── src/
│   └── main/
│       ├── java/
│       │   └── com/example/commande/
│       │       ├── CommandeServiceApplication.java       ← Classe principale Spring Boot
│       │
```

```
│       │       ├── controller/                           ← Couche API REST
│       │       │   └── CommandeController.java
│       │
│       │       ├── dto/                                  ← Objets d'échange (API)
│       │       │   ├── CommandeRequestDTO.java
│       │       │   ├── CommandeResponseDTO.java
│       │       │   └── ProduitDTO.java
│       │
│       │       ├── model/                                ← Documents MongoDB
│       │       │   └── Commande.java
│       │
│       │       ├── repository/                           ← Accès base MongoDB
│       │       │   └── CommandeRepository.java
│       │
│       │       ├── service/                              ← Logique métier
│       │       │   └── CommandeService.java
│       │
│       │       ├── mapper/                               ← Conversion DTO ↔ modèle
│       │       │   └── CommandeMapper.java
│       │
│       │       └── client/                               ← Communication Feign
│       │           └── ProduitClient.java
│       │
│       └── resources/
│           ├── application.yml                           ← Config globale
│           └── application-dev.yml                       ← Profil dev (MongoDB, Eureka)
│
├── Dockerfile                                             ← Pour conteneurisation
├── pom.xml                                                ← Dépendances Maven
```

3- Explications des répertoires

Dossier	Rôle principal
`controller`	Gère les routes HTTP (REST API). Utilise des DTO pour échanger les données avec le front/API
`dto`	Contient les objets d'entrée (`RequestDTO`) et de sortie (`ResponseDTO`) pour chaque commande
`model`	Définit les entités métiers persistées dans MongoDB (`Commande`)
`repository`	Interfaces qui étendent `MongoRepository` pour requêter la base MongoDB
`service`	Contient la logique métier. Gère les traitements de commandes, calculs, validations, etc.
`mapper`	Convertit les entités (`Commande`) en DTO (`CommandeResponseDTO`) et inversement
`client`	Interfaces Feign pour appeler les microservices externes (ex: `produit-service`)
`resources`	Fichiers de configuration Spring (application.yml, profils, port, nom de service, MongoDB, etc)

4- Technologies utilisées

Élément	Outil / Framework
Langage	Java 17
Framework principal	Spring Boot
ORM NoSQL	Spring Data MongoDB
Communication inter-service	Spring Cloud OpenFeign
Découverte de service	Spring Cloud Eureka
Transfert de données	DTO + Mapper (manuels)
Conteneurisation	Docker
Build & dépendances	Maven

5- Sécurité (futur)

- Authentification via utilisateur-service
- Vérification du token dans les requêtes

VI. Spécifications techniques du microservice : « products-service »

1- Objectif du microservice

Le microservice products-service est responsable de la gestion des produits proposés à l'achat groupé par les grossistes. Il doit permettre la création, modification, consultation, suppression et surveillance des produits, tout en respectant des seuils et dates limites de groupage.

2- Technologies et outils

Élément	Choix technique
Langage	Java 17
Framework	Spring Boot 3.5.0
ORM	Spring Data JPA
Base de données	PostgreSQL
Build tool	Maven

3- Structure des packages

com.monapp.products_service

- controller // Contrôleurs REST
- dto // DTOs pour les échanges API
- model // Entités JPA
- repository // Interfaces JpaRepository
- service // Logique métier
- scheduler // Tâches planifiées
- application.properties // Configuration

4- Modèle de données

Attribut	**Type**	**Contraintes**
id	Long	PK, auto-généré
nom	String	Non null
description	String	Optionnel
prixUnitaire	BigDecimal	précision : 10, scale : 2
seuilMinimum	Integer	Obligatoire pour les achats groupés
quantiteDisponible	Integer	>= 0
dateLimiteGroupage	LocalDateTime	date de fin de l'achat groupé

image	String	URL ou nom de fichier (optionnel)
dateCreation	LocalDateTime	Auto-généré à la création
idGrossiste	Long	FK vers user-service (lecture seule ici)

5- DTOs (Data Transfer Objects)

ProduitRequestDTO (requêtes de création/mise à jour)

- nom, description, prixUnitaire, seuilMinimum, quantiteDisponible, dateLimiteGroupage, image, idGrossiste

ProduitResponseDTO (réponses client)

- Tous les champs + seuilAtteint (boolean), dateLimiteDepassee (boolean)

6- Routes REST exposées

Base : /api/produits

Méthode	**Route**	**Description**
GET	/api/produits	Liste tous les produits
GET	/api/produits/{id}	Obtenir un produit par son ID
POST	/api/produits	Créer un nouveau produit
PUT	/api/produits/{id}	Modifier un produit existant
DELETE	/api/produits/{id}	Supprimer un produit
GET	/api/produits/grossiste/{idGrossiste}	Liste les produits d'un grossiste

7- Fonctionnalité métier : planification(scheduler)

ProduitScheduler

- Tâche planifiée : toutes les heures (cron 0 0 * * * *)
- Vérifie si un seuil est atteint (quantiteDisponible <= seuilMinimum)
- Vérifie si la date limite de groupage est dépassée
- Prépare la notification au grossiste (à implémenter)

8- Règles métier spécifiques

- Un produit appartient à un seul grossiste (idGrossiste)
- La création ou la modification d'un produit impose que idGrossiste soit fourni.
- Si la date limite est passée et le seuil est atteint, le produit est prêt à être validé par les commandes.

9- Test à implémenter

- Tests unitaires : création de produit, conversion DTO ↔ Entity, détection seuil/date
- Tests d'intégration : endpoints REST, base H2

10- Evolutions prévus

- Intégration avec notification-service pour les alertes
- Sécurisation complète avec JWT + rôles
- Ajout de statistiques produits
- Upload de fichiers image via REST