SEVELLEC Maxime
FERDOEL Titoutan
LANDON Cyril

Groupe 32A

Étude de cas

Qualité de développement

IUT Orléans, département informatique

2024/2025

Github : https://github.com/MaximeSevellec/qualite_dev.git

TP Exercice 1: Analysez l'application

Tâche 1 :

1. Quels sont les principaux domaines métiers de l'application Order flow ?

L'application **Order Flow** est centrée sur le domaine principal du shopping en ligne, qui est structuré en différents sous-domaines.

Les **domaines principaux** incluent le panier d'achat, qui gère toutes les interactions de l'utilisateur avec son panier. Cela comprend des fonctionnalités comme l'ajout, la suppression et la modification des articles sélectionnés. Un autre domaine principal est le traitement des commandes, qui prend en charge toutes les étapes de gestion des commandes après la validation du panier par l'utilisateur.

Les **domaines de support** jouent un rôle essentiel dans le fonctionnement de l'application. Le registre des produits maintient une liste d'identifiants et de détails des produits disponibles dans le système. Le catalogue de produits, quant à lui, fournit des informations détaillées sur les produits disponibles à l'achat. La gestion des stocks est responsable du suivi et de la mise à jour des niveaux de stock pour chaque produit. Enfin, la gestion des clients s'occupe des informations et des préférences des clients, permettant une expérience utilisateur personnalisée.

Par ailleurs, l'application repose également sur des **domaines génériques**. Le domaine de notification permet d'envoyer des alertes et des mises à jour aux clients. Le sourcing d'événements conserve et récupère les événements relatifs à l'état des différentes entités de l'application. Enfin, le domaine de la monnaie gère les objets monétaires, comme les prix et les devises, associés aux produits et aux transactions.

Chaque domaine est conçu pour fonctionner en harmonie avec les autres, garantissant ainsi une expérience utilisateur fluide et efficace.

2. Comment les microservices sont-ils conçus pour implémenter les domaines métiers ?

Les microservices de l'application Order Flow sont conçus selon une architecture pilotée par les événements et un modèle CQRS (Command Query Responsibility Segregation). Cette architecture permet de séparer les opérations de commande et de requête, optimisant ainsi la gestion des états des entités et la performance de l'application. Chaque domaine métier est encapsulé dans un microservice dédié qui gère les agrégats, commandes et requêtes associés.

Les microservices utilisent Apache Pulsar pour l'échange de messages, et MongoDB pour le stockage d'événements dans une structure NoSQL.

3. Quelles sont les responsabilités des conteneurs de code ?

Les conteneurs de code de l'application **Order Flow** ont des responsabilités spécifiques, bien définies pour assurer le bon fonctionnement du système.

Le conteneur **apps/of-api-gateway** joue un rôle central en tant que point d'entrée unique de l'application. Il reçoit les requêtes des clients, les analyse et les redirige vers les microservices appropriés. En plus de la gestion des requêtes, il prend également en charge des aspects critiques tels que la sécurité des communications et le routage intelligent des données.

Dans le cadre du registre des produits, deux conteneurs distincts collaborent pour garantir une gestion efficace et performante des informations produits. Le conteneur **apps/of-product-registry-microservices/product.registry** est dédié à la gestion complète du registre des produits. Il s'assure que chaque produit possède un identifiant unique et que ses informations de base sont toujours disponibles et maintenues à jour. En complément, le conteneur **apps/of-product-registry-microservices/product.registry.read** se concentre sur la lecture des données du registre. Il est optimisé pour répondre aux requêtes client rapidement, en offrant une vue adaptée et performante des informations produits.

La bibliothèque **libs/event-sourcing** est responsable de la gestion du sourcing d'événements au sein de l'application. Elle fournit une interface standardisée pour stocker, récupérer et rejouer les événements, permettant ainsi une traçabilité et une

reproductibilité des actions dans le système. La bibliothèque **libs/published-language**, quant à elle, joue un rôle clé dans la communication entre les microservices. Elle définit un langage de domaine partagé qui garantit la cohérence des messages échangés et facilite les interactions interservices.

Ces conteneurs, en collaborant harmonieusement, assurent une organisation claire, une communication fluide et des performances optimales pour l'application **Order Flow**.

Tâche 2 : Identifier les concepts principaux

L'application **Order Flow** repose sur plusieurs concepts essentiels pour structurer son architecture, assurer la communication entre ses composants et garantir la fiabilité des opérations.

Le **Domain-Driven Design (DDD)** est au cœur de l'organisation des domaines métiers. L'application se divise en trois catégories principales. Les domaines centraux comprennent le panier d'achat et le traitement des commandes, qui représentent les fonctionnalités principales. Les domaines de support, tels que le registre des produits, le catalogue de produits, la gestion des stocks et la gestion des clients, assurent les opérations auxiliaires nécessaires à la gestion du flux de commandes. Enfin, les domaines génériques, notamment la notification, le sourcing d'événements et la monnaie, fournissent des services communs à l'ensemble des fonctionnalités.

La communication entre les microservices s'appuie sur une **Event-Driven Architecture (EDA)**. Ce paradigme favorise l'échange d'événements entre les différents composants pour traiter les modifications en temps réel et garantir une cohérence des données à travers le système.

Pour optimiser les opérations, l'application implémente le modèle **CQRS (Command Query Responsibility Segregation)**, qui sépare clairement les opérations d'écriture (commandes) et de lecture (requêtes). Cela permet de traiter chaque type d'opération de manière optimisée et indépendante.

Le concept de **sourcing d'événements** joue un rôle clé dans la gestion des états internes. Chaque changement est enregistré sous forme d'événement dans une base de données NoSQL, comme MongoDB, garantissant une traçabilité complète. Grâce à cette approche, il est possible de recréer l'état actuel d'un agrégat en rejouant son historique d'événements, tout en assurant une reprise fiable en cas de panne.

Pour la communication entre les microservices, l'application utilise **Apache Pulsar**, un système de gestion des messages qui simplifie le traitement et la transmission des événements. Cela garantit une coordination efficace et une intégration fluide entre les différents composants.

Les microservices de l'application sont spécialisés pour chaque domaine. Par exemple, les services dédiés au panier d'achat et au traitement des commandes gèrent les processus métiers principaux, tandis que d'autres services se concentrent sur les opérations de support, comme la gestion des produits ou des stocks. Ces microservices exploitent la structure CQRS et s'appuient sur des événements pour assurer un fonctionnement modulaire et scalable.

Enfin, la bibliothèque **libs/event-sourcing** centralise la gestion des événements, en fournissant une interface commune pour stocker, récupérer et rejouer les événements. Cela simplifie la gestion des états dans l'ensemble des microservices.

L'implémentation de l'event-sourcing assure également une grande fiabilité. En stockant chaque événement dans une base de données dédiée, comme MongoDB, elle permet de reconstituer l'historique des opérations et d'assurer une reprise des activités même en cas de panne. Cela garantit une traçabilité complète et une continuité des services, renforçant la robustesse globale de l'application Order Flow.

Tâche 3 : Analyse des problèmes de qualité dans l'application *Order Flow*

Voici les principaux problèmes de qualité de code les plus fréquemment rencontrés dans le fichier analysé avec SonarLint.

1. Duplication de valeurs répétées

Exemple :

- *Define a constant instead of duplicating this literal "correlation-id" 3 times.*
- *Define a constant instead of duplicating this literal "Unexpected event type: " 3 times.*

Problème général :
La duplication de valeurs répétées dans le code enfreint le principe de base de la programmation appelé *DRY* (Don't Repeat Yourself). Cela complique la maintenance du code, car toute modification d'un littéral doit être répétée à plusieurs endroits, augmentant ainsi le risque d'erreur.

Solution :
Il faudrait les passer en constante globale

2. Complexité cognitive élevée

Exemple :

- *Refactor this method to reduce its Cognitive Complexity from 19 to the 15 allowed.*

Problème général :
La complexité cognitive mesure la difficulté pour un humain de comprendre un morceau de code. Plus une méthode contient des boucles, conditions, etc… Plus elle est compliquée. Cela rend le code difficile à lire, à maintenir et à tester.

Solution :

Découper les méthode en sous méthode et utiliser des switch case.

3. Utilisation de classes d’exception génériques

Exemple :

- *Define and throw a dedicated exception instead of using a generic one.*

Problème général :

L’utilisation d’exceptions comme **`Exception`** ou **`RuntimeException`** rend le code moins compréhensible. Cela complique le traitement des erreurs, car il devient difficile de distinguer différents types d'erreurs ou de fournir des informations précises.

Solution :

Créez des exceptions spécifiques pour chaque cas d'erreur. Cela améliore la lisibilité du code et permet de capturer les erreurs de manière plus précise.

Exemple :

```
public class InvalidEventTypeException extends RuntimeException {
    public InvalidEventTypeException(String message) {
        super(message);
    }
}
```

4. Modifier la visibilité et les constantes statiques

Exemple :

- *Make id a static final constant or non-public and provide accessors if needed.*

Problème général :

Les variables de classe qui ne sont pas correctement encapsulées (non privées) ou qui ne sont pas déclarées comme constantes statiques lorsqu'elles ne seront pas modifiées augmentent le risque de rencontrer des problèmes avec le code.

Solution :

- Déclarez les variables immuables comme `static final`.

- Utilisez des modificateurs d’accès appropriés (`private`) et fournissez des accesseurs (getters).

Recommandations générales :

1. Utilisez des outils comme SonarLint régulièrement pour détecter les problèmes de qualité dans le code.
2. Respectez les principes SOLID pour structurer le code de manière modulaire et compréhensible.
3. Documentez
4. Réalisez des revues de code régulières afin de prévenir ces problèmes avant qu'ils ne s'accumulent.

TP Exercice 2: Corriger les problèmes de qualité du micro service du registre de produits (Côté écriture)

Tâche 1 : Compléter les commentaires et la Javadoc

Toutes ces modifications sont disponibles sur le git du projet, indiqué tout en haut de ce document, dans la branche exercice_2_tache_1.

Modifications effectué dans les fichiers suivants :

- ProductRegistryEventPayloadMapper.java
- ProductRegistryEventEntityMapper.java
- ProductRegistryEventRepository.java
- Les autres fichiers “ProductRegistry” n’ont pas eu besoin de nouvelles modifications, ceux-ci comprenant déjà la documentation nécessaire à leur compréhension.

Tâche 2 : Corriger les RuntimeException paresseuses

Toutes ces modifications sont disponibles sur le git du projet, indiqué tout en haut de ce document, dans la branche exercice_2_tache_2.

Les fichiers qui devaient être modifiés sont les suivants :

- ProductRegistryCommandResource
- ProductRegistryQueryResource
- ProductRegistryEventEmitter

Au final, 6 nouvelles exceptions ont été introduites :

- Pour ProductRegistryEventEmitter :
    - ProductRegistryEventProductionException
    - ProductRegistryProducerCloseException
    - ProductRegistryProducerCreationException
- Pour ProductRegistryQueryResource :
    - ProductRegistryConsumerbyCorrelationException
- Pour ProductRegistryCommandResource :
    - ProductRegistryConsumerCreationException

Tâche 3 : Convertir les payloads des Entités (couche de persistance) en records

Toutes ces modifications sont disponibles sur le git du projet, indiqué tout en haut de ce document, dans la branche exercice_2_tache_3.

Les fichiers modifiés sont les suivant :

- ProductRegisteredEventEntity
- ProductRemovedEventEntity
- ProductUpdatedEventEntity

Ils ont tous été transformés d'un "public static class … " en un "public record … "

Tâche 4.0 : Modifier les Entités (couche de persistance) pour utiliser des champs privés avec des accesseurs

Toutes ces modifications sont disponibles sur le git du projet, indiqué tout en haut de ce document, dans la branche exercice_2_tache_4.

Les fichiers modifiés sont les suivant, par ailleurs ce sont les mêmes que le point précédent, car cette tâche vient compléter les actions effectuées en tâche 3 :

- ProductRegisteredEventEntity

- ProductRemovedEventEntity
- ProductUpdatedEventEntity

Tâche 4.1 : Champs publics sur les entités Panache

Toutes ces modifications sont disponibles sur le git du projet, indiqué tout en haut de ce document, dans la branche exercice_2.

La modification comprend un grand nombre de fichiers et vient appliquer les standards qu'apporte SonarCube au code. Actuellement, il reste encore 7 warnings à mettre en place parmi tous les fichiers du projet.

TP Exercice 3: Permettre la collaboration en équipe

Tâche 1 : Choisir un modèle de contrôle de version et une stratégie de branchement

Pour une collaboration efficace, le modèle recommandé serait Gitflow :

- Une branche principale (main) pour les versions stables
- Une branche de développement (develop) pour les ajouts en cours
- Branches spécifiques pour les fonctionnalités (feature), les correctifs (hotfix) et les releases (release)

Tâche 2 : Définir les responsabilités des équipes

Sur la base des microservices existants, les responsabilités peuvent être réparties ainsi :

Equipe 1 : Gestion des domaines principaux

- Service panier d'achat : gestion des agrégats, commandes, et requêtes
- Service de traitement des commandes : gestion des commandes et mise à jour des états

Equipe 2 : Gestion des domaines de support

- Registre des produits et catalogue de produits : maintenance des informations produits
- Gestion des stocks : suivi des niveaux de stock
- Gestion des clients : coordination des profils utilisateurs

Tâche 3 : README et règles de collaboration

- complétion du README.md
- ajout du CONTRIBUTING.md

Tâche 4 : Division en équipes et canaux de communication

Division des équipes :

- Equipe 1 : Focalisée sur les domaines principaux
    - Développeur backend, responsable des tests, intégrateur des commandes
- Equipe 2 : Axée sur les domaines de support et génériques
    - Responsable des notifications, gestion des agrégats produit, et intégration du sourcing d’événements