Rapport Qualité Dev

VALIN Ophélie, groupe 3.3

TP Exercice 1 : Analyser l'application

Tâche 1 : Ségrégation des responsabilités + Questions

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

Les principaux domaines métiers sont les suivants :

- **Product Catalog (ProductCatalogDomain)** : permet de gérer les catalogues produits (prix, entrée catalogue)
- **Product Registry (ProductRegistryDomain)** : gestion globale des produits (création / mise à jour / suppression)
- **Shopping Cart (ShoppingCartDomain** : gestion du panier (ajout, suppression, modification, réservation)
- **Order Processing (OrderProcessingDomain)** : prise de commande et suivi

- **Stock (StockDomain)** : gestion des stocks / réservations
- **Customer / Notifications / Billing** : gestion clients, notifications et facturation

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

Chaque domaine a des modules ou services dédiés (ex. product registry, product catalog, store back/front). On a également une séparation explicite entre les microservices *command* et *read* (ex. `product-registry` a `domain-service` (command) et `read-service` (query)). Enfin, les échanges sont fait via des évènements du type : `Outbox`.

3. Quelles sont les responsabilités des modules suivants ? :

- `apps/store-back`

Expose les APIs backend utilisées par le front (`store-front`), orchestration des appels métiers et façonnage des réponses.

- `apps/store-front`

Interface utilisateur : affichage du catalogue, du panier, et du parcours client.

- `libs/kernel`

Modèles de domaine et types partagés (noyau métier), règles et entités réutilisables par les services.

```
+--- project :libs:bom-platform
+--- project :libs:cqrs-support
```

- `apps/product-registry-domain-service`

"Command side" pour l'enregistrement de produit : logique métier (création / mise à jour / suppression de produits) et génération d'événements.

```
+--- project :libs:bom-platform
+--- project :libs:contracts:product-registry-contract
```

```
+--- project :libs:kernel
+--- project :libs:cqrs-support
+--- project :libs:cqrs-support (*)
```

- `apps/product-registry-read-service`

“Read side” pour stocker et exposer des projections des produits (API de lecture, recherches paginées, stream d’événements).

- `libs/bom-platform`

Bill Of Materials (BOM) pour le build (versions centralisées des dépendances du projet).

- `libs/cqrs-support`

Bibliothèque utilitaire pour faciliter CQRS/Event handling tel que types communs ou encore helpers pour events.

- `libs/sql`

Gestion des scripts de migration DB (Liquibase) ainsi que des changelogs pour la base de données.

L’architecture suit le modèle suivant *Domain-Driven Design + CQRS + event-driven* : modules **command** (domain services) produisent des événements, modules **read** consomment et exposent des projections.

Tâche 2 : Identifier les concepts principaux

1. Quels sont les concepts principaux utilisés dans l'application Order flow ?

> 📌 TIP
> Comment sont stockées les données ? Comment sont gérées les transactions ? Comment sont gérés les évènements métiers ? Comment sont gérées les erreurs (métier et technique) ? Quelles formes prennent les échanges entre les services ?

Les concepts principaux utilisés dans l'application Order flow sont les suivants :

- **DDD (Domain-Driven Design)** : avec des bounded contexts décrits dans `domain.cml`.
- **CQRS** : séparation Command / Query (ou Read), par exemple : `product-registry-domain-service` et `product-registry-read-service`.
- **Event-driven** : des événements métier produits par la side-command et propagés aux consommateurs (read services, autres bounded contexts).
- **BFF + SPA** : store-back (BFF) + store-front (Angular).
- **Projections / Read models** : vues JPA exposées via APIs REST.
- **Libs partagées** : `kernel` (modèles métier), `cqrs-support` (helpers CQRS), `contracts` (schémas/DTOs).

Les données sont stockées dans une base relationnelle via JPA/Hibernate. Les migrations sont gérées avec Liquibase. Ensuite, le workflow se présente de la sorte : command side applique un changement → Un poller/outbox worker lit la table outbox et publie (ref. `OutboxPartitionedPoller`) → read services consomment et projettent pour mettre à jour les vues.

Les erreurs, quant à elles, sont modélisées par des exceptions/validations au niveau des services et renvoyées via REST (BFF) pour l'UI.

2. Comment les concepts principaux sont-ils implémentés dans les différents modules ?

- **DDD + Bounded Contexts** : domaines modélisés dans `domain.cml` (ProductCatalog, ProductRegistry, ShoppingCart,

OrderProcessing, ...). Au niveau des modules on retrouve : des règles métier et des services de commande dans `product-registry-domain-service` , ainsi que des read/projections dans `product-registry-read-service`.

- **Commandes & Domain Services (partie écriture)** : services de commande (ex. `RegisterProductService`, `UpdateProductService`) dans `product-registry-domain-service`.
- **Projections / Read Models (partie lecture) :** read-service contient `ProjectionDispatcher`, `ReadProductService`, entités JPA (`ProductViewEntity`) et repository (`JpaProductViewRepository`).

3. Que fait la bibliothèque `libs/cqrs-support` ? Comment est-elle utilisée dans les autres modules (relation entre métier et structure du code) ?

C'est une bibliothèque de support pour implémenter une architecture orientée événements dans les services Quarkus du projet. Plus précisemment elle implémente une infrastructure pour le pattern CQRS (Command Query Responsibility Segregation) en Java.

Elle est utilisée notamment dans d'autres modules tels que RegisterProductService, UpdateProductService, RetireProductService.

4. Que fait la bibliothèque `libs/bom-platform` ?

C'est un Bill of Materials (BOM) pour le projet, c'est-à-dire un module Gradle `java-platform` qui centralise et impose les versions des dépendances utilisées partout dans le mono-repo. Par exemple, il impose la version de Hibernate Validator, ou encore de JUnit Jupyter. Les autres modules l'appliquent avec `enforcedPlatform(project(":libs:bom-platform"))`, ce qui garantit des versions cohérentes dans tout le projet.

5. Comment l'implémentation actuelle du CQRS et du Kernel assure-t-elle la fiabilité des états internes de l'application ?

Plusieurs mécanismes sont mis en place pour assurer la fiabilité des états internes de l'application :

- Validation et invariants : les règles métiers sont vérifiées dans les agrégats avant émission d'événements

(ex. Product.updateName, updateDescription, retire). De plus, des exceptions sont lancées si l'état ne permet pas l'opération (ex. produit déjà retiré) et bean validation (ConstraintViolationException).

- Event envelope et versioning d'agrégat : l'événement est enveloppé avec un numéro de séquence correspondant à la version de l'agrégat (EventEnvelope.sequence). L'agrégat incrémente sa version avant de retourner l'événement.
- Append-only event_log et contrainte d'unicité : les événements sont persistés dans une table append-only `eventing.event_log`. La BD impose une contrainte unique `(aggregate_type, aggregate_id, aggregate_version)`, ce qui empêche des insertions concurrentes de la même version et force le rollback en cas de conflit.
- Outbox transactionnelle :
  - Lors d'une commande (ex. RegisterProductService.handle), on :
    - modifie/sauve l'agrégat (repository.save),
    - eventLog.append(evt),
    - outbox.publish(...) le tout dans une transaction (@Transactional), garantissant atomicité (si l'append échoue, tout est rollbacké).
  - L'outbox est liée à l'`event_log` et fournie avec une requête SQL qui :
    - récupère les événements prêts,
    - ordonne par `aggregate_id, aggregate_version`,
    - empêche de traiter un agrégat tant qu'il y a un événement bloqué (`NOT EXISTS ...`),
    - utilise `FOR UPDATE SKIP LOCKED` pour permettre un traitement concurrent tout en préservant l'ordre par agrégat.

Tâche 3 : Identifier les problèmes de qualité

En lançant le Megalinter depuis un terminal windows powershell à l'aide la commande suivante :

```
docker run --rm `
  -e RUN_LOCAL=true `
  -e GITHUB_WORKSPACE=/tmp/lint `
  -v ${PWD}:/tmp/lint `
  -v //var/run/docker.sock:/var/run/docker.sock `
  nvuillam/mega-linter:latest
```

J'obtiens un fichier `mega-linter.log` qui décrit le nombre d'erreurs pour chaque fichiers concernés. Voici un résumé des différents modules principaux avec erreurs :

- RegisterProductService.java : 21 erreurs
- RetireProductService.java : 13 erreurs
- UpdateProductService.java : 27 erreurs
- ProductRegistryCommandResource.java : 32 erreurs
- etc.

Les problèmes de qualités relevés communs à la plupart de ces fichiers sont notamment :

- "Missing a Javadoc comment." [JavadocVariable]
- "First sentence should end with a period." [JavadocStyle]
- "Comment matches to-do format 'TODO:'." [TodoComment]
- "Line is longer than 80 characters (found 83)." [LineLength]
- "Parameter mapper should be final." [FinalParameters]
- [HiddenField]
- [DesignForExtension]
- [MagicNumber]

- etc.

Parmi eux ce trouve des erreurs d'implémentation, de formattage ou de code.

TP Exercice 2: Corriger les problèmes de qualité et introduire des tests

Tâche 1 : Compléter les commentaires et la Javadoc

Voir le code source pour la javadoc.

La documentation JAVA faite est uniquement celle des différentes classes. La documentation des méthodes est proposée par la suite dans la tâche 2, comme problème de qualité remonté.

Tâche 2 : Corriger les erreurs et les problèmes de qualité remontés

Pour certains problèmes de qualités relevé par méga-linter, j'ai choisis de ne pas les traiter pour éviter de boulverser l'application.

Par exemple : "Line is longer than 80 characters (found 88). [LineLength]" pour une déclaration de classe et son implémentation d'une interface. Le nom de la classe était explicite, je ne voyais donc pas d'intérêt à l'abréger, puis par la suite à modifier toutes ses références.

Tâche 3 : Tests Unitaires

Les tests ont été implentés dans le fichier `ProductTest.java`, situé dans la librairie kernel. Le nom du package a été inspiré des autres existants et l'emplacement choisit en fonction de la cohérence du projet.

Commande utilisée pour lancer les tests :

```
./gradlew :libs:kernel:test --no-daemon --console=plain
```

Les options `--no daemon` et `--console=plain` ont pour objectifs :

- `--no-daemon` : force Gradle à ne pas utiliser le Daemon. Ici elle est utile pour des exécutions ponctuelles.
- `--console=plain` : définit le mode d'affichage de la console sur du texte brut. Ici cela évite par exemple des bars de progression pour chaque test.

Tâche 4 : Test Intégration

Les tests ne passant pas, ils ont été supprimé du rendu final.

Tâche 5 : Questions

1. Quelle est la différence entre les tests unitaires et les tests d'intégration ?

Les tests unitaires vérifient le comportement d'une unité de code isolé (méthode ou classe). Ces tests sont conçus pour s'assurer ue le bloc de code s'exécute comme prévu, conformément à la logique implémentée.

D'un autre côté, les tests d'intégration vérifient le fonctionnement conjoint de plusieurs composants. Ces composants sont joints et testés pour évaluer leur efficacité à fonctionner ensemble, à intéragir avec succès.

Les tests unitaires visent principalement la logique métier, contre la cohérence globale pour les tests d'intégration.

2. Est-il pertinent de systématiquement couvrir 100% de la base de code par des tests ?

Non il n'est pas pertinent d'obtenir un taux de couverture de tests de 100%.

Premièrement ce pourcentage ne correspond pas aux taux de succès des tests mais seulement à la part qui est exéxutée, ce qui rend la poursuite de ce pourcentage moins importante.

De plus, certains composants / méthodes ne sont pas testables ou non pertinent comme les getter/setter, les constructeurs simples ou bien les configurations. 100% est parfois une utopie pour un projet, surtout à grande envergure.

Dans le monde du développement, on cherchera plutôt à viser une très bonne couverture de code avec un taux de 70% à 90%. Chaque test implémenté viendra compléter le précedent (ex : tests unitaires, tests d'intégration, etc.).

3. Quels avantages apporte une architecture en couches d'oignon dans la couverture des tests ?

Principe :

- Le cœur (domaine / métier) est indépendant des frameworks
- Les dépendances vont toujours de l'extérieur vers l'intérieur
- Les couches externes dépendent du cœur, jamais l'inverse

Avantages :

- Les tests unitaires sont plus rapides, simples et stables
- Les ports (interfaces) sont mockés
- Les implémentations (infra) sont testées séparément

Dans le projet, les tests unitaires ont été placé au coeur du domaine (kernel) et sont donc indépendant de toute infrastructure, testable seuls. Les tests ont également pu être réalisé simplement, sans besoin de base de données, de configuration, de requête HTTP ou d'un quelconque contexte applciatif.

Ces tests unitaires valident les règles métier, les invariants, les transitions d'état ou encore la gestion des erreurs métiers, mais pas la technique.

4. Expliquer la nomenclature des packages `infra`, `application`, `jpa`, `web`, `client`, `model`.

`model` : contient les objets du domaine, les entités métiers et les value objects. Son rôle est de représenter les concepts métier. Le model est indépendant de tout framework.

`infra` : contient les implémentations et les adaptateurs techniques. Son rôle est de détailler la partie technique. L'infra dépend des couches internes.

`jpa` : sous-package de l'infra. Contient les entités jpa, les repositories, et les mapping ORM. Son rôle est en lien avec la pertinence des données.

`web` : sous-package de l'infra. Contient des controlleurs REST, des DTOs web et la gestion http. Son rôle est de prendre en charge l'interface avec les clients HTTP.

`client` : contient les clients vers des services externes et l'api REST. Son rôle est la gestion de la communication sortante vers l'extérieur.

`application` : contient les services applicatifs, la logique métier et les cas d'usages. Son rôle est l'orchestration du métier. Le package application est le point central des règles fonctionnelles.

TP Exercice 3 : Collaborez sur le mono-repo

BOCQUET Clémence
Groupe : 3.3

TP Exercice 4 : Projection des événements dans des vues matérialisées

Tâche 1 : Questions sur la base de code

1. Expliquer le rôle de l'interface `Projector` dans le système de gestion des événements.

L'interface `Projector` a pour rôle de traduire une suite d'événements métier en un état courant. Dans notre application avec une architecture CQRS :

les commandes produisent des évènements → les évènements sont persistés → le Projector permet de reconstruire ou de faire évoluer un état à partir de ces évènements.
Elle garantit l'application séquentielle, idempotente et contrôlée des événements, tout en séparant clairement la logique de projection de la logique métier et de la

persistance. Grâce au type `ProjectionResult`, elle formalise les différents résultats possibles d'une projection (succès, échec, no-op) et favorise une approche fonctionnelle, prévisible et robuste du traitement des événements.

2. Expliquer le rôle du type `S` dans l'interface `Projector`.

Le type `S` est générique et représente l'état projeté à partir des événements. `E` correspond à un évènement (ce qu'il s'est produit), et `S` correspond à un état (ce qu'il en résulte).

4. Quel est l'intérêt de passer par une interface `Projector` plutôt que d'utiliser une classe concrète ?

Passer par une interface `Projector` permet de découpler la logique de projection de son implémentation, ce qui rend le système extensible, testable et substituable (plusieurs projections possibles pour les mêmes événements, mocks en tests, évolution sans impact sur le reste du code)

5. Quel est le rôle de la classe `ProjectionResult` dans l'interface `Projector` ?

La classe `ProjectionResult` encapsule le résultat d'une projection en indiquant explicitement si elle a réussi, échoué ou été un no-op, tout en transportant l'état projeté ou l'erreur, ce qui permet de chaîner les projections de manière sûre et déterministe.

6. Expliquer en quoi l'usage de la Monade est intéressant par rapport à la méthode de gestion d'erreur traditionnelle en Java et détailler les avantages concrets.

L'usage d'une monade permet de modéliser explicitement le succès, l'échec ou le no-op comme des valeurs, plutôt que de les gérer via des exceptions.

Avantages concrets :

- Flux de contrôle explicite et lisible (plus de `try/catch` imbriqués)
- Composition facile des traitements (`map`, `flatMap`)
- Moins d'effets de bord et logique plus déterministe
- Tests simplifiés, car les erreurs sont des résultats, pas des ruptures d'exécution

Tâche 2 : Questions concernant l'Outboxing

1. Expliquer le rôle de l'interface `OutboxRepository` dans le système de gestion des événements.

L'interface `OutboxRepository` implémente le pattern Outbox : elle assure la persistance fiable des événements à publier, leur récupération ordonnée pour diffusion, et la gestion des échecs, garantissant ainsi la cohérence entre la base de données et la publication des événements dans une architecture CQRS.

2. Expliquer comment l'Outbox Pattern permet de garantir la livraison des événements dans un système distribué.

L'Outbox Pattern garantit la livraison des événements en **persistant les événements dans la même transaction que les données métier**, puis en les **publiant de façon asynchrone avec retries**, évitant ainsi toute perte en cas de panne.

Tâche 3 : Questions concernant le journal d'évènements

1. Expliquer le rôle du journal d'événements dans le système de gestion des événements.

Le journal d'événements enregistre chronologiquement tous les événements métier, servant de source pour reconstruire l'état du système et garantir la traçabilité et la reproductibilité.

2. Pourquoi l'interface `EventLogRepository` ne comporte-t-elle qu'une seule méthode `append` ? Pourquoi n'y a-t-il pas de méthode pour récupérer les événements ou les supprimer ?

L'interface `EventLogRepository` est append-only, elle sert uniquement à enregistrer tous les événements dans l'ordre pour la traçabilité et la reconstruction d'état. On ne supprime ni ne modifie les événements, et la lecture se fait via des projections, pas directement depuis le repository.

3. En tirant vos conclusions de votre réponse à la question 2 et de l'analyse de l'application (Objets liés à l'event log, schéma de base de données), déterminez les implications de cette conception sur la gestion des événements dans l'application et quelles pourraient être les autres usages du journal d'événements.

La conception append-only de l'Event Log garantit une source de vérité immuable pour reconstruire l'état du système et rejouer les événements à volonté. Cela permet une traçabilité complète, facilite la reprojection pour de nouvelles vues et le débogage. D'autres usages possibles incluent l'audit métier, la génération de statistiques, la synchronisation avec d'autres systèmes ou encore la reprise après incident.

Tâche 4 : Limites de CQRS

1. Identifier et expliquer les principales limites de l'architecture CQRS dans le contexte de l'application.

CQRS impose une séparation stricte entre Command et Query, des modèles différents pour la lecture et l'écriture, ce qui introduit le besoin de plus de classes, de packages et de conventions à respecter comme dans le contexte de l'application. Le principe même de celle-ci reste simple, mais un grand nombre de classes et de packages sont nécessaire à sa rédaction et son fonctionnement. Nous avons donc une architecture complexe, CQRS peut sembler surdimensionné par rapport au besoin fonctionnel actuel.

De plus, CQRS impose de par sa séparation Command / Query de réaliser des tests unitaires pour chaque côté et de réaliser des tests d'intégration pour les projections. Le coût des tests augmente donc.

2. Quelles limites intrinsèques à CQRS sont déjà compensées par la mise en œuvre actuelle de l'application ?

3. Quelles autres limites pourraient être introduites par cette mise en œuvre ?

CQRS pourrait également introduire une duplication des modèles et du code (ex : DTOs, etc.), une gestion plus complexe de la cohérence des données, ou encore un coût de mise en place et d'apprentissage (ex: dans le cadre d'une équipe).

4. Que se passerait-il dans le cas d'une projection multiple (un évènement donnant lieu à plusieurs actions conjointes mais de nature différente) ?

Une projection multiple signifie qu'un même évènement métier déclenche plusieurs actions ou mises à jour, chacune ayant une finalité différente.

Si l'on prend l'exemple d'un évènement `ProductRegistered` , il peut entraîner :

- Mise à jour d'un read model
- Publication d'un message externe
- Mise à jour d'un compteur

Chaque projection est :

- Indépendante
- Abonnée à l'évènement
- Exécutée séparément

Ce qui peut donner suite à un problème d'atomicité, avec certaines projections qui peuvent réussir, et d'autres échouer. Toujours dans notre exemple nous pourrions avoir : un produit visible en lecture mais une action métier associée non effectuée.

5. Question bonus : Proposez des solutions pour atténuer les limites identifiées dans les questions précédentes (notamment la question 3).