LOG430 – Architecture Logicielle (Été 2025)
Laboratoire 6 – Auto-évaluation détaillée

Nom : Maksym Pravdin Code permanent : PRAM86290201

Informations générales

- URL du dépôt GitHub/GitLab : https://github.com/PraMaks/LOG430_Labo_0/
- Langage et framework principaux (Spring Boot, Quarkus, NestJS, ...) : Django (frontend) Express.js (backend)
- Composant d'orchestration (service dédié, workflow engine, ...) : Service dédié: Orchestr-sales-service
- Bus de messages / mécanisme de communication (file, échange direct, ...) : Logger avec pino
- Technologie de persistance de la machine d'état (base SQL/NoSQL, in-memory, ...) : MongoDB
- Outil(s) d'observabilité (collecte de métriques, traces, visualisation) : Prometheus + Grafana

1. Évaluation technique par composant

1.1 Conception de la saga

Critère	Oui	Non	Commentaire / Justification
Le scénario métier (4 étapes max.) est clairement décrit dans le code et le README	✓		Oui, scénario d'enregistrement d'une commande en ligne est choisi et est indiqué dans le code, le rapport et le README
Un service orchestrateur central coordonne la séquence d'étapes	✓		Oui, service orchestr-sales-service est utilisée.
L'orchestrateur publie un *événement de démarrage* et met à jour l'état à chaque étape	✓		Oui, les évènements sont mis à jour au cours de l'évènement et son état est changé en cours de route.

1.2 Implémentation des événements et de la machine d'état

Critère	Oui	Non	Commentaire / Justification
Chaque microservice publie un événement de succès ou d'échec	✓		Oui, après chaque service un état est assigné à l'evenement en cours selon le résultat
L'état courant de la commande est persistant et interrogeable	✓		Oui, l'état est mis à jour au cours de route et est utilisée à partir d'un enum
Un diagramme de la machine d'état est fourni dans le rapport	✓		Oui. Un tableau est fourni dans le rapport pour expliquer les états
Les événements sont idempotents (gestion des doublons)	✓		Oui, au cours d'un evenement, la même instance est mis à jour et une nouvelle est crée uniquement quand c'est une nouvelle commande.

1.3 Gestion des échecs et compensations

Critère	Oui	Non	Commentaire / Justification
Des échecs simulés (stock insuffisant, paiement refusé, ...) sont documentés	✓		Oui, les tests avec des résultats variées sont utilisées.
Des actions de compensation ou de rollback sont implémentées	✓		Oui, si une étape échoue il y a des actions de rollback qui se produisent
Les temps de détection d'échec et de compensation sont mesurés	✓		Oui, ceci peut être obtenu à partir de Grafana + Prometheus
Aucun processus ne reste bloqué en état intermédiaire	✓		Oui, il y a des timeouts pour les processus.

1.4 Observabilité et traçabilité

Critère	Oui	Non	Commentaire / Justification
Un système de collecte de métriques est branché à l'orchestrateur et aux services	✓		Oui Prometheus et Grafana sont utilisées
Des indicateurs clés (durée moyenne d'une saga, taux d'échecs, étapes atteintes) sont exposés	✓		Oui, Prometheus utilise des métrqiues custom que j'ai créé
Un tableau de bord regroupe ces métriques (visualisation personnalisée ou template)	✓		Oui, dashboard grafana est utilisée pourles afficher
Les logs structurés permettent de suivre une saga de bout en bout (trace ID, correlation ID)	✓		Oui, il est possible de suivre la saga avec les events et les logs

1.5 Tests de scénarios

Critère	Oui	Non	Commentaire / Justification
Un jeu de tests automatisés couvre les scénarios de succès	✓		Oui, il y a des tests pour les succès
Des tests couvrent les échecs et vérifient les compensations	✓		Oui, il y a des tests d'échecs
Les tests sont intégrés à la CI/CD	✓		Oui, ces tests sont ajoutés aux pipeline et la creation de l'image
Des rapports ou captures d'exécution des tests sont inclus dans le livrable	✓		Oui, ceci est dans le rapport

2. Réflexion personnelle approfondie

1. **Choix d'orchestration :** Pourquoi avoir opté pour une saga orchestrée (et non chorégraphiée) ? Quels avantages ou compromis avez-vous observés ?

   La saga orchestrée permet d'avoir un contrôle sur toutes les étapes et gérer les échecs. Ainsi, on gange en termes de visibilité aussi pour Grafana + Prometheus. Par contre, cela peut rendre le service d'orchestration comme un bottleneck et ajoute du couplage entre les services.

2. **Machine d'état :** Comment avez-vous modélisé et persisté l'état ? Quelles difficultés liées à la cohérence et à la récupération après panne avez-vous rencontrées ?
   Pour la persistance la base de donnnées MongoDB est utilisée pour chaque evenement d'enregistrement d'une commande en ligne et celle-ci est mis à jour avec la gestion des enums et de logs au cours de la même saga. Il a fallu ajouter des scenarios de rollback dans le cas que ça crash ou milieu du traitement.

3. **Compensation :** Quelle stratégie de compensation avez-vous retenue ? Dans quel cas votre solution pourrait-elle échouer ou engendrer des inconsistances ?

   Si la commande échoue, celle-ci possède l'état d'échec et s'arrête clairement avec des rollback s'il sont nécessaires. La solution pourrait échouer lorsque le service est surchargé en termes de trafic sans nécessairement être horsligne.

4. **Observabilité :** Quels indicateurs se sont révélés les plus utiles pour diagnostiquer un incident dans la saga ? Comment les visualiser rapidement ?
   Grafana est utilisée pour visualiser les métriques custom de Prometheus et le plus utile était le temps moyen d'une saga et le nombre d'état traitées.

5. **Tests :** Quelles astuces avez-vous mises en place pour fiabiliser les tests des scénarios comportant des échecs aléatoires ?
   Des tests d'échecs pour la non-connection au serveur était mis en place ainsi qu'une simulation d'échec en plein milieu de la saga qui nécessite un rollback.

6. **Réutilisation professionnelle :** Quelles pratiques ou outils de ce laboratoire comptez-vous appliquer dans un contexte industriel (ex. e-commerce, finance) ?

   Continuer d'utiliser Grafana+Prometheus pour la visibilité des performances et la notion d'orchestration pour les systèmes plus complexes dans une architecture orientée en services.