Travail d'été pour permettre de compenser les TPs non réalisés

L'objectif de ce travail est d'implémenter une transaction distribuée qui permette de passer une commande de repas, puis de lire les commandes passées en protégeant les accès en cas de crash de serveur. On travaillera en trois étapes

Partie 1

Dans cette partie on va créer le service de commande de repas, constituée d'une commande de mets et une commande de boisson réalisées par deux services différents et séparés. Cette transaction distribuée doit être réalisée complètement ou pas du tout. Pour cela on utilisera le pattern SAGA. Quand l'application cliente envoie au SEC (Saga Execution Component) un objet DTOCommande contenant l'URL du client de la commande, l'URL du met et l'URL de la boisson, le SEC utilise ces URLs pour créer une transaction de réservation du met et de la boisson en envoyant une requête à deux services qui enregistrent séparément les informations dans leur BDD locale.

Comme vu en cours dans le pattern SAGA, si une (ou plusieurs) transaction(s) n'est (ne sont) pas passée(s), le SEC envoie une requête de compensation dans tous les services dans lesquels les transactions sont passées, qu'il connait grâce à son fichier log.

A implémenter :

Un web service SEC qui, suite à la réception d'une requête avec un objet DTOCommande, orchestre l'exécution de la transaction en envoyant des queries de réservation dans les services :

- Repas pour les réservations de mets
- Boisson pour les réservations de boissons

Les web services Repas et Boissons qui possèdent : une query pour enregistrer la réservation reçue dans leur BD locale et une query de compensation.

- L'enregistrement contient le numéro de la commande, la date, l'URL du client ayant commandé, l'item commandé et son prix
- Si tout se passe bien, ces services retournent un code 200.
- En cas de problème sur l'un des services, le SEC envoie à l'autre service une requête de compensation à partir de l'URL de la transaction à compenser. Si l'URL de la réservation n'existe pas il retourne un 404.

On testera les services en développant une application cliente qui effectue les commandes via le SEC. Les listes d'items (liste de mets et de boissons) peuvent être hard codées dans le client.

Pour tester le bon fonctionnement du pattern SAGA, simulez une erreur dans l'un des deux services en retournant un 500 et vérifiez que le SEC envoie des requêtes de compensation et que la base de données est correctement corrigée.

Partie 2

La lecture des réservations de mets et de boissons est considérée comme étant beaucoup plus fréquente que l'écriture. On utilisera en conséquence le pattern CQRS, pour séparer lecture et écriture. Le writer est le service réalisé dans la partie 1. Il faut donc implanter 2 services reader qui enregistrent les évènements de commande pour lecture ultérieure(pour simplifier pas besoin d'enregistrer la commande dans le writer). Un tel évènement contient le numéro de commande, la date, le nom du client, le nom du met et le nom de la boisson et le prix total(imprimez sur la console le numéro de commande pour pouvoir l'utiliser par la suite pour la lecture). Cet évènement est un DTO qu'il faudra concevoir.

La communication entre le writer (SEC) et les readers est réalisé par le pattern PubSubHub (voir exemples faits en classe). L'implantation de ce dernier utilisera les threads java pour réaliser l'asynchronisme (voir notes de cours).

A implémenter :

- Un web service « reader » qui enregistre les évènements de commande dans sa BDR. Ce service s'inscrit à ces évènements auprès du PubSubHub. Une commande peut être récupérée au travers du numéro de commande.
- Un service PubSubHub asynchrone.

Le service de lecture doit utiliser la classe FileStorage qui stocke les informations dans un fichier à plat (bien que les écritures devraient en réalité se faire dans une base de données)

On testera les services via l'application cliente en ajoutant la fonctionnalité de lecture.

Partie 3

Le but de cette partie est de compléter l'implantation des parties 1 et 2 pour protéger les applications clientes d'une déficience d'un service « reader ». On le réalisera en utilisant le pattern « Circuit breaker » avec 3 états (open, closed, half_open), de

manière à éviter d'attendre sur un service reader dans le cas où ce service est indisponible ou déficient.
Dans le Service_Reader vous trouverez un web service qui simule un serveur qui au début a des performances faibles ce qui cause un « TimeOut », puis se libère après quelques requêtes (à la 15ème requête), pendant que ses performances sont faibles il faut envoyer la requête au Service_Reader2 pour lire la commande.

A implémenter

- Implémenter le pattern à partir de la présentation en classe en suivant les étapes suivantes:
    1) D'abord le circuit breaker est fermé
    2) Si la requête déclenche un timeout on ouvre le "circuit breaker".
    3) Les 4 requêtes suivantes déclenchent une exception « CircuitBreakerException", ce qui a pour conséquence de ne pas envoyer les requêtes au serveur et permettre au client de trouver une alternative (ici l'alternative est d'envoyer la requête à un autre service reader).
    4) A la 5ème requête on ouvre le "circuit breaker" à moitié, dans cet état on exécute 3 fois la requête pour vérifier l'état du serveur :
        - Si les 3 requêtes déclenchent un timeout on re-ouvre le "circuit breaker" et retour à l'état 2.
        - Si non on déduit que le serveur est revenu à son état normal et on ferme le "circuit breaker".

Les résultats attendus sont fournis dans le fichier « ResultatConsolRead.txt ».

À implémenter :

- La classe CircuitBreaker qui est invoquée par l'application cliente.

On testera le pattern via l'application cliente.

Remarque générale : toutes les communications entre les services doivent utiliser le protocole Atom. Les transferts de données se font par des DTO

Utiles :

Le fichier pdf TPCommandesV2.pdf fournis montre les étapes de l'exercice

Dans le service SEC :

- La classe FileStorage avec les méthodes suivantes :
  //vide le fichier log
  - rewrite (filePathLog)
    //retourne une ArrayList contenant les 3 services de réservation(chaque objet « Service » contient le type et l'url) :
  - readAllServices(filePathServicesWrite)
    //enregistrent dans le log la réponse et l'url
  - appendDTO("ok-"+url, filePathLog)
    appendDTO("ko-"+url, filePathLog)
    //retourne une ArrayList contenant les urls des services où les mises à jour ont été executées avec succès
  - readAllok (filePathLog)

Dans le Client Client_Commande :

Dans les Services repas et boissons:

- La classe FileStorage avec les méthodes suivantes :
  //remplace les données se trouvant dans le file1 par les données du file2
  - save(file1,file2);
    ///enregistre la data dans le filePath passé en paramètres. Pour simplifier il faut enregitrer directement le DTOXXX reçu dans le filePathActualD.
  - append(dataAtom, filePath)
    //retourne un représentant une DTOXXX à partir de l'id passé en paramètres
  - read(id, filePathActualD) ;

Bon travail!

Questions : Assistant Bassim Bentahar B424