UniLiveTest v1.0 Solution

Contexte

Ce document détaille les choix techniques et fonctionnels qui ont été faits lors de l'implémentation d'UniLiveTest. Ainsi que la motivation de ces choix.

Détail des hypothèses technico-fonctionnelles

Version de Java

La version de Java choisie est la version 17, bien qu'aucune fonctionnalité spécifique à Java 17 n'a été utilisée. J'ai préféré utiliser la dernière version afin que toutes les fonctionnalités soient disponibles pour les futurs développements. La seule fonctionnalité « récente » utilisée est celle des Text Blocks qui date de Java 15. Celle-ci est principalement utilisée à des fins de lisibilité lors de l'écriture des Query SQL dans les repository.

Récupération des données filtrées

Lors de la récupération des sélections ou des événements, le filtrage est réalisé en SQL plutôt qu'en Java. Cela implique la création de plus de Query dans les repository. Mais cela évite l'overFetching puisque les entités liées sont en mode LazyFetching. Cela permet aussi une récupération de la donnée directement filtrée est prête à l'emploi.

Gestion des exceptions

Dans l'ensemble, je préfère gérer les exceptions de validation dans la couche API / WS. Cependant il y avait déjà un exemple d'exception 404 throw dans la couche service. J'ai donc considéré que c'était la bonne pratique.

Gestion des exceptions 60x

Je n'ai pas trouvé comment throw des 60x en utilisant le modèle d'exception déjà présent dans le template de l'exercice. J'ai donc créé une nouvelle exception « CustomNonStandardException » qui permet de gérer cela en prenant en paramètre un type possédant le status code à renvoyer. Il aurait été possible d'uniformiser pour n'utiliser que cette exception, mais cela aurait demandé de dupliquer une partie du code de la classe HttpStatus. J'ai donc décide de faire cohabiter les deux exceptions.

Status code 204 GET

Le contrat d'interface de l'énoncé stipule que ces WS peuvent retourner une 204 no content. En l'occurrence au vu de l'implémentation que j'ai faite, ces WS ne devraient jamais retourner NULL, mais plutôt une liste vide. J'ai donc throw une exception en cas de liste vide pour retourner une 204.

Naming des paramètres du Body de la requête POST /bets/add

Dans l'énoncé, le contrat d'interface explicite le body avec certains champs en français. J'ai considéré que c'était non changeable et j'ai donc utilisé JsonAlias pour manipuler des attributs avec des noms anglais comme dans le reste du code.

Typage du Body de la requête POST /bets/add

Afin de manipuler la donnée sans encombre, j'ai utilisé le type BigDecimal et j'ai mis en place une sécurité supplémentaire afin de refuser tout nombre qui n'aurait pas une scale de 2 ou qui ne serait pas supérieur à 0 afin de simplifier les questions d'arrondis. Via la classe MoneyDeserializer (nom selon moi, plus parlant que AmountDeserializer, mais qui ne correspond pas totalement puisque j'utilise ce désérializer pour la côte aussi).

Prise de pari

Afin d'éviter les problèmes de concurrence et que le client se retrouve avec une balance sur son compte qui ne soit pas correcte, (Lost update) j'ai mis en place de l'Optimistic Locking avec l'annotation @version sur un champ version dans la table Customer

Retour des GET events et sélections

Le contrat d'interface ne spécifiait pas les champs à retourner ou non. Je n'ai donc pas créé de DTO spécifiques et j'ai considéré qu'aucun champ n'était « sensible » ou inutile. Ces api retournent donc le Bean de l'entité BDD.

Stockage de la côte et du montant du pari

J'ai ajouté à la table Bet deux colonnes, une pour le montant et une pour la côte au moment du pari. Afin de permettre le paiement du pari à la côté choisie et non pas la côte actuelle. J'aurai pu stocker le montant à payer, mais d'un point de vue fonctionnel et historisation, il m'a semblé plus intéressant de stocker les deux informations.

Notion d’évènement LIVE

Dans l’énoncé, il est spécifié qu’un évènement est ouvert tant qu’un marché est ouvert. J’ai donc considéré que le filtre isLive gardait les évènements ouverts et dont la date de début est dans le passé.

Filtre sur les évènements live

J’ai considéré que si le paramètre était false on ne récupérait pas que les événements fermés mais, plutôt tous les évènements.

Tâches de mise à jour des sélections.

Lors de mes tests, les instructions ne semblaient pas s’exécuter. J’ai l’impression que c’est un comportement under the Hood qui considère que si on réalise un Count() d’un Stream il n’y a pas besoin d’exécuter les instructions qui ne changent pas la taille du Stream (donc les maps et mapToObj). Afin de pouvoir tester j’ai donc légèrement modifié l’implémentation.

Job payer les paris

J’ai utilisé l’instruction synchronised pour éviter que le job ne se lance deux fois en même temps. J’ai aussi ajouté un bloc catch bien que le traitement ne me semble pas risqué, afin d’être sur qu’une erreur de traitement sûr un pari ne bloque pas les autres traitements.

Dans le cas d’une sélection perdue, on aurait pu simplement faire un update de tous les bets en SQL sans passer par la couche service. Cependant, on aurait eu un traitement ligne par ligne sur un pari gagnant et un Batch update sur un pari perdu. Pour plus de clarté et d’homogénéité, j’ai préféré toujours passer par la couche service.

Ajout d’un swagger

Afin de simplifier l’utilisation du projet, j’ai ajouté une dépendance vers « springdoc-openapi-ui ». Le swagger est disponible à l’URL : http://localhost:8887/swagger-ui/index.html.
Une collection Postman permettant le test du projet est aussi disponible à la racine du projet : UniLiveTest.postman_collection.json.