TourGuide

TourGuide Application :

Documentation fonctionnelle et technique

CAPPON SÉBASTIEN

Table des matières

1 Présentation du projet

1.1 Objectifs du projet

L'application ***TourGuide*** connaît une forte croissance de son nombre d'utilisateurs. En l'espace de deux mois, sa fréquentation de quelques centaines d'utilisateurs a centuplé. Son utilisation par plus de 100.000 individus est à prévoir dans les mois à venir. Si ses performances en sont déjà lourdement impactées, de forts ralentissements sont à anticiper d'ici l'arrivée des nouveaux clients.

Fort de ce succès, l'application est aussi amenée à évoluer rapidement. Il est donc nécessaire, dans le cadre de ce projet, de :

- **Réusiner *(Refactor)* l'application pour améliorer ses performances**
  L'application TourGuide fait appel à plusieurs modules externes qui ralentissent son fonctionnement. Le traitement de 100.000 utilisateurs demande entre 2 heures et 18 heures, selon les méthodes appelées. Il est impératif de passer ces délais sous les seuils respectifs de 15 minutes et 20 minutes.

- **Mettre en place un pipeline d'intégration continue assurant le build, les tests, et le package en *.jar**
  L'application est amenée à évoluer. Sa fréquentation va continuer de croître. La mise en place d'un Pipeline d'intégration continue facilitera la gestion des livrables par les futurs développeurs qui travailleront sur TourGuide. L'automatisation du build, des tests et du package constitue un gain de temps en adéquation avec la ligne directrice de ce projet.

1.2 Éléments hors périmètre

Durant la réalisation de ce projet, 3 nouveaux objectifs se sont imposés :

- **Le refactor de l'application**
  Le code de l'application, en l'état où il a été fourni, est anormalement compliqué. De nombreuses fonctions semblent avoir été placées dans certaines classes « par défaut ». Les services ne sont pas interfacés. Le découpage en package ne respecte aucune logique. Il en résulte un code difficile à lire, à comprendre et à maintenir. Le refactor du code s'inscrit, à son tour, dans la ligne directrice du projet : améliorer sa maintenabilité.

- **La rédaction de nouveaux tests unitaires et d'intégration**
  Toujours pour assurer une bonne maintenabilité du code et accélérer les interventions futures des développeurs à venir, il est primordial d'assurer une bonne couverture de

code. Aussi est-il nécessaire d'ajouter de nouveaux packages et de nouvelles classes à ceux et celles existants.

- **La journalisation du projet**
  Pas indispensables, mais utiles à des fins de test et de debug, la mise en place de logs et la rédaction d'une Javadoc sont des atouts pour le suivi du projet.

- **L'ajout du code quality tests job sur le pipeline d'intégration continue**
  Un pipeline d'intégration continue complet comprend 4 étapes : build, test, code quality tests, package. Quitte à mettre un pipeline IC en place, autant le faire correctement et implémenter l'ensemble des fonctionnalités de ce dernier. La maintenabilité du code ne s'en voit qu'améliorée.

1.3 Métriques du projet

Afin de déterminer l'atteinte des objectifs posés dans le chapitre 1.1, nous surveillons les paramètres suivants :

- **Le temps nécessaire en seconde pour tracker un nombre donné d'utilisateur**
  L'application traque ses utilisateurs en permanence en actualisant leur position toute les 5 minutes. Ceci permet de retourner les attractions à proximité de l'utilisateur au fur et à mesure que sa localisation change. Cette opération récurrente est primordiale pour le bon fonctionnement de TourGuide. Si elle ne s'exécute pas dans les plus brefs délais, les utilisateurs sont susceptibles de manquer des attractions.

- **Le temps nécessaire en seconde pour attribuer des rewards à tous les utilisateurs**
  Les utilisateurs visitent des attractions qui leur rapportent des rewards. Plus il y a d'utilisateurs connectés, plus il y a de rewards à distribuer simultanément. Cette opération récurrente doit s'accomplir dans les plus brefs délais afin de fluidifier l'utilisation de TourGuide par les utilisateurs.

2 Spécifications fonctionnelles

- **Appeler la page d'accueil :**
  http://the-domain.ext/
  Permet d'afficher un message de bienvenue. Sera utilisé pour retourner une page développée en front.

- **Récupérer un utilisateur :**
  http://the-domain.ext/user?userName=[internalUserX]
  Permet de récupérer l'ensemble des valeurs des attributs d'un utilisateur au format JSON.

- **Récupérer la position d'un utilisateur :**
  http://the-domain.ext/getLocation?userName=[internalUserX]
  Permet de récupérer la latitude et la longitude auxquelles se situe un utilisateur donné, ainsi que la date à laquelle cette position a été visitée, au format JSON.

- **Récupérer les 5 attractions les plus proches de l'utilisateur :**
  http://the-domain.ext/getNearbyAttractions?username=[internalUserX]
  Permet de récupérer, sous forme d'objet JSON, les noms, latitudes et longitudes des 5 attractions les plus proches de l'utilisateur, ainsi que la position dudit utilisateur, la distance entre ce dernier et l'attraction concernée et le nombre de points de récompense qu'elle pourrait rapporter.

- **Récupérer la liste des récompenses de l'utilisateur :**
  http://the-domain.ext/getRewards?userName=[internalUserX]
  Permet de récupérer uniquement la valeur de l'attribut userRewardList de l'utilisateur, qui contient la liste des récompenses obtenues en visitant les attractions.

- **Récupérer la liste des agences proposant des voyages faisant l'objet de récompenses :**
  http://the-domain.ext/getTripDeals?userName=[internalUserX]
  Permet de récupérer, au format JSON, la liste des agences proposant des voyages pour l'utilisateur, avec leur nom, le prix du voyage proposé, et son identifiant unique.

3 Spécifications techniques

3.1 Diagramme de conception technique

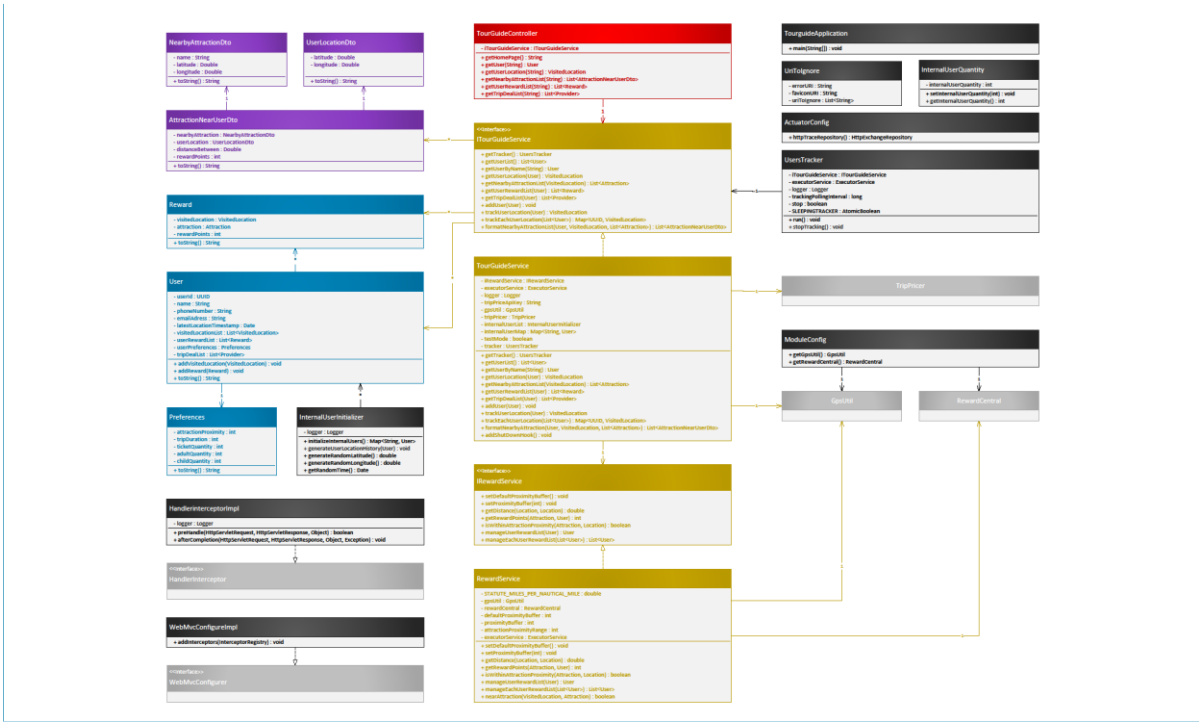

Figure 1 - Diagramme de classe de l'application TourGuide

Le diagramme de classe ci-dessus décrit le fonctionnement de l'application de façon détaillée. Pour une meilleure lisibilité, merci de consulter le document *Diagramme de classe Tourguide application.pdf* dans le dossier */documents* du présent projet.

3.2 Glossaire

3.2.1 Glossaire de l'application

- **Attractions :** Tout point d'intérêt que peut visiter un utilisateur. Ils se géolocalisent et rapportent des Reward Points au Users.

- **Location :** Emplacement d'un User ou d'une Attraction. Se définit par sa latitude et sa longitude.

- **Preferences :** Configuration du profil du User. Il y définit le rayon de recherche d'une Attraction proche, la durée de ses voyages, la quantité de tickets nécessaires, d'adultes et d'enfants, par voyage.

- **Provider :** Agences de voyage proposant des Trip deals aux utilisateurs afin de leur faire gagner plus de Rewards.

- **Reward :** Récompense décernée à un User pour avoir visité une Attraction. Elle est propre à l'Attraction visitée et rapporte un nombre de Reward Points propre à celle-ci.

- **Reward Points :** Points que rapporte une Attraction visitée par un User. Chaque Attraction possède un nombre de points préalablement défini.

- **Trip deal :** Voyage organisé par un Provider et pouvant être acheté par un User.

- **User :** Utilisateur de l'application. Fait appel à des Providers, visite des Attractions, chasse des Rewards, emmagasine des Reward Points. Il peut paramétrer son profil et définir des Preferences.

3.3 Solutions techniques

Le projet se découpe en 3 étapes majeures nécessitant chacune leur solution technique : La gestion des ConcurrentModificationException, la mise en place du controller /getNearbyAttractions et l'accélération des performances de l'application.

3.3.1 Solution à ConcurrentModificationException

Afin de résoudre les ConcurrentModificationException, nous avons opté pour l'utilisation d'une CopyOnWriteArrayList de VisitedLocation dans la méthode manageUserRewardList() (ex calculateRewards()). CopyOnWriteArrayList crée une copie en mémoire de la liste parcourue afin d'y intégrer les modifications, avant de compléter la liste originale avec le contenu de sa copie. Cette solution présente l'avantage d'être simple et rapide à mettre en place sans modifier la logique de la méthode où elle est appelée.

3.3.2 Solution pour le controller /getNearbyAttraction

/getNearbyAttraction fait appel à des modules externes, installés en début de projet. Les informations nécessaires au retour de la méthode appelée par ce controller sont une partie des attributs des objets contenus dans ces modules. L'emploi de Jackson JSON Views n'était donc pas envisageable. Nous avons opté pour des DTO (Data Transfer Object) afin de remplir les objectifs imposés pour la mise en place de ce controller.

3.3.3 Solution pour l'amélioration des performances

L'amélioration des performances globales de l'application en vue de gérer l'augmentation du nombre d'appels aux méthodes trackUserLocation() et manageUserRewardList() est possible grâce à l'utilisation de ExecutorService et de son pool de threads.

Toutefois, il fut nécessaire au préalable d'ajouter deux méthodes permettant respectivement de gérer les méthodes suscitées sur plusieurs utilisateurs. Ces méthodes extraient la boucle située dans le test. Le test fait désormais appel aux nouvelles méthodes, et non aux anciennes au sein de boucles.

Dans le cas présent, c'est donc la boucle qui a été accélérée et non la méthode appelée des modules installés. Cet ajout fut intégré aux services respectifs afin qu'ils fonctionnent en production et non uniquement durant les tests. La consigne était claire, ce ne sont pas les tests que l'on passe en asynchrone.

3.4 Autres solutions non retenues

Seules les étapes relatives à la ConcurrentModificationException et à l'amélioration des performances ont fait l'objet de solutions alternatives.

3.4.1 Alternative à la solution à ConcurrentModificationException

La première solution mise en place passait par la création d'une liste temporaire de Rewards dans la méthode concernée par l'exception, avant d'ajouter cette liste à l'attribut userRewardList de l'objet User. Il s'agit ni plus ni moins du fonctionnement d'une CopyOnWriteArrayList. Cette solution a été abandonnée au profit de son homologue, plus simple, et lisible.

3.4.2 Alternative à la solution pour l'amélioration des performances

Avant l'utilisation d'un pool de thread et d'ExecutorService, notre choix s'était porté sur l'utilisation des CompletableFuture au sein de méthodes appelées. Toutefois, la complexité de ces dernières et le manque de documentation précise sur le sujet nous ont poussé à nous rabattre sur une solution solide ayant fait ses preuves.

De plus, l'association des CompletableFuture dans les méthodes appelées, en complément des pools de threads avec ExecutorService sur les méthodes gérant les boucles, provoquait l'échec de tests au-delà de 10.000 utilisateurs. Il est intéressant, toutefois, de noter qu'en deçà de cet échantillon, les résultats étaient exceptionnels. L'hypothèse la plus probable est que la multiplication des appels à des méthodes asynchrones dans des méthodes asynchrones, en grande quantité, provoque des conflits entres les méthodes en cours d'exécution.