TourGuide

Mise à l’échelle de TourGuide

Documentation fonctionnelle et technique

Sommaire

1. Présentation du projet

TourGuide est une application Spring Boot permettant à ses utilisateurs et utilisatrices de voir quelles attractions touristiques sont à proximité et d'obtenir des réductions chez des entreprises partenaires.

TourGuide s'adresse d'une part à un public de touristes, qui a besoin de découvrir des points d'intérêt dans des endroits qui leur sont inconnus, et d'autre part à des entreprises de l'événementiel et de l'hôtellerie, qui souhaitent mettre en avant leur projet devant des clients potentiels.

1.1 Objectifs du projet

TourGuide rencontre actuellement des problèmes de performance. Suite à une augmentation importante et imprévue du nombre d'utilisateurs, les temps de réponse de l'application ont augmenté, au point que les informations renvoyés par l'application ne sont plus pertinentes pour les utilisateurs lorsqu'elles sont reçues. Ce projet a pour objectif principal de résoudre ce problème.

En plus d'améliorer le temps de réponse de l'application, la qualité et la fiabilité des modifications futures sera assurée par la mise en place d'une pipeline d'intégration continue.

La méthode RewardsService.calculateRewards déclenche parfois une ConcurrentModificationException, ce qui met en péril la stabilité de l'application, et doit être corrigé.

Le comportement de TourGuideController.getNearByAttractions ainsi que celui de TourGuideService.getNearByAttractions doivent être modifiés, pour mieux répondre au besoin des utilisateurs.

1.2 Hors du champ d'application

La solution apportée conviendra au fonctionnement de l'application pour une utilisation intense, évaluée à 100 000 utilisateurs simultanés. Cependant le nombre d'utilisateurs varie en fonction du temps, avec des périodes de pic et des périodes de creux. Le choix a été fait de ne pas prendre en compte ces variations, et de choisir et dimensionner la solution uniquement pour 100 000 utilisateurs, étant donné du manque d'informations sur les variations du nombre d'utilisateurs, et que le problème de temps de réponse de l'application doit être résolu rapidement.

Cet aspect devra être traité par la suite, pour ne pas gaspiller de ressources économiques et énergétiques en faisant fonctionner en permanence une solution

pouvant supporter 100 000 utilisateurs alors que le besoin sera beaucoup plus bas pendant la majorité du temps.

1.3 Mesures du projet

Le projet sera réussi si l'application se lance sans erreurs et en validant la suite de test développée par un ingénieur Assurance qualité senior, notamment les tests de performance vérifiant un temps de réponse inférieur à 15 minutes pour TrackLocation et inférieur à 20 minutes pour GetRewards avec 100 000 utilisateurs simultanés.

2. Spécifications fonctionnelles

- L’application fonctionne sans erreurs critiques.

- TourGuideService.getNearbyAttractions renvoie une liste des cinq attractions les plus proches de l’utilisateur, en se basant sur sa position GPS, pour que l’utilisateur ait toujours accès à plusieurs recommandations d’attractions différentes, peu importe l’endroit où il se trouve.

- TourGuideController.getNearbyAttractions génère un fichier JSON contenant, pour chacune des attractions, son nom, sa latitude, sa longitude, la latitude de l’utilisateur, la longitude de l’utilisateur, la distance entre l’attraction et l’utilisateur en miles, et le nombre de points obtenus par l’utilisateur à l’attraction.

- Les tests highVolumeTrackLocation et highVolumeGetRewards sont réussis pour un nombre d’utilisateurs de 100.000, afin de s’assurer que l’application soit capable de traiter un grand nombre de requêtes en moins de 15 et 20 minutes respectivement.

- La pipeline d'intégration continue assure la fiabilité des futures modifications, en installant l’application sur une machine virtuelle et en exécutant les tests lors du déploiement de nouvelles fonctionnalités, ce qui permet d’éviter d’intégrer du code défaillant au projet.

3. Spécifications techniques

3.1 Schémas de conception technique

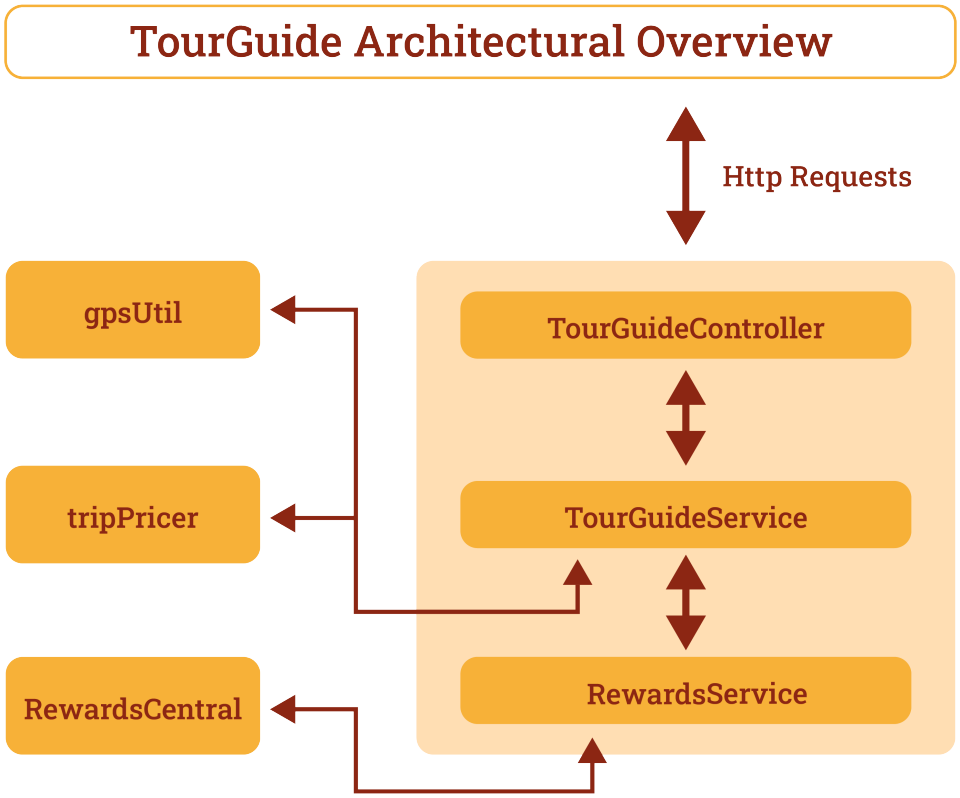


Schéma d'architecture de l'application

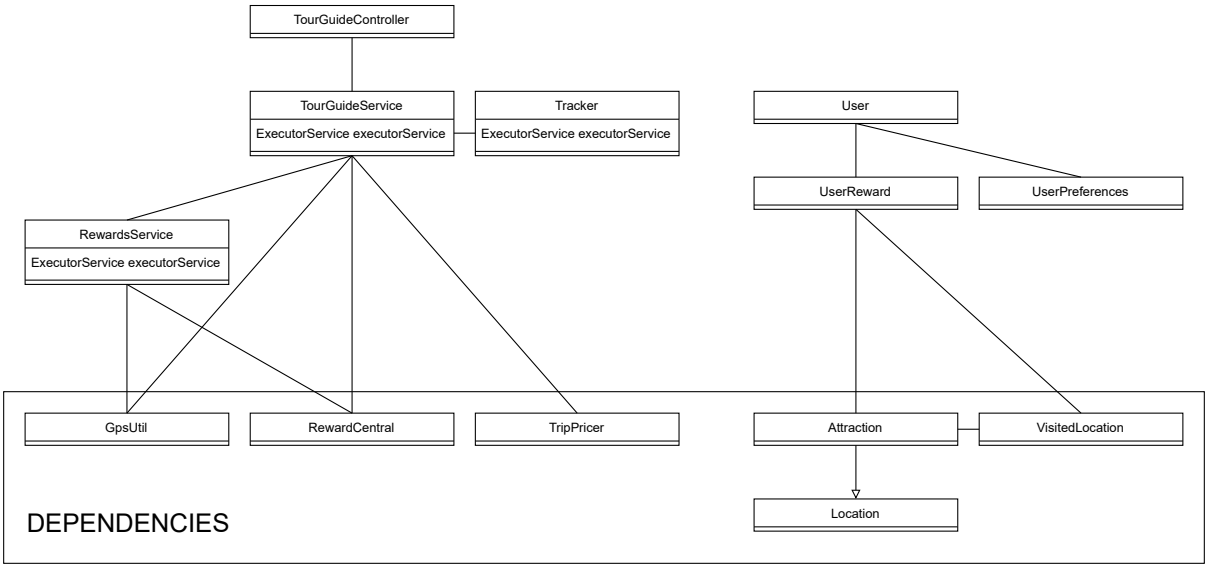


Diagramme UML simplifié de l'application

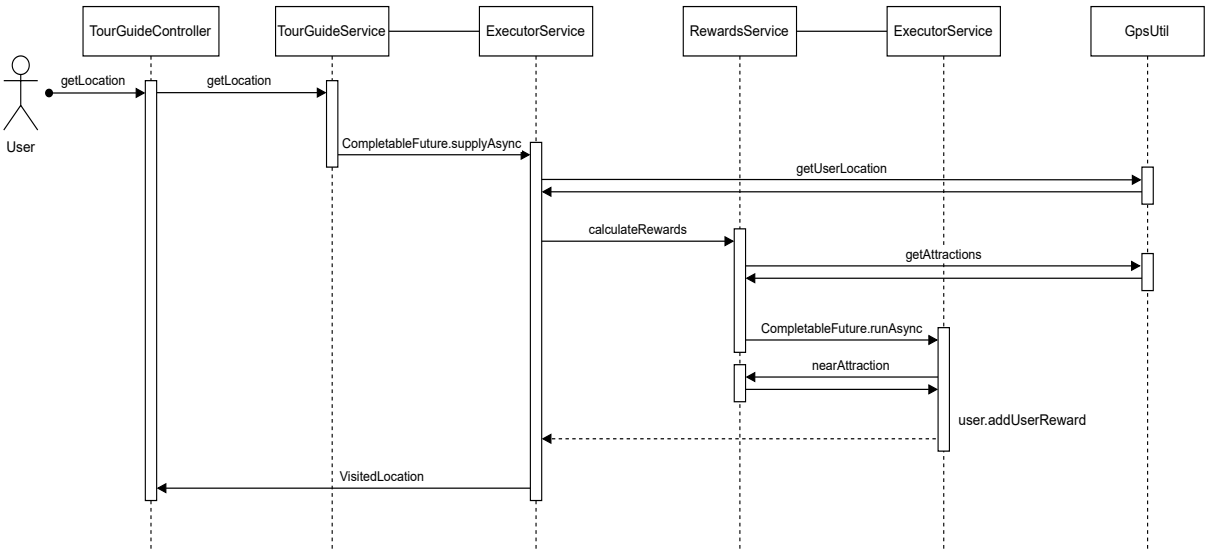


Diagramme de séquence d'une requête GET sur l'URL getLocation

3.2 Glossaire

Attraction : un site touristique, défini par un nom et une adresse.
Location : un endroit, défini par une latitude et une longitude.
UserReward : un nombre de points donnant accès à des offres chez nos partenaires, obtenu par un utilisateur lors d'une visite à une Attraction.
VisitedLocation : une visite d'un utilisateur à une Location ayant eu lieu à une date précise.

3.3 Solutions techniques

3.3.1 RewardsService.calculateRewards

La lancement d'une ConcurrentModificationException lors de l'exécution de RewardsService.calculateRewards provenait de l'ajout d'un élément à la List<VisitedLocation> userLocations pendant une itération sur cette liste. Cela a été résolu en la remplaçant par une CopyOnWriteArrayList<VisitedLocation>.

Un deuxième problème est apparu sur RewardsService.calculateRewards, le test nearAllAttractions étant un échec car userRewards.size() est à 1 quand il devrait être à 26. Cela venait de la méthode User.addUserReward. Avant d'ajouter une UserReward à un User, cette méthode vérifie que le User ne posséde pas déjà une UserReward pour la même Attraction que la UserReward que l'on souhaite ajouter. Cette vérification était faite via un filtre sur un stream dont la condition était toujours vraie (elle vérifiait la non-égalité entre une Attraction et un Attraction.attractionName). En remplaçant la condition par une égalité entre les attractionName des deux Attractions, le test est réussi.

3.3.2 TourGuideService.getNearbyAttractions

TourGuideService.getNearbyAttractions a été modifiée pour récupérer la liste de toutes les attractions, trier cette liste par ordre de distance avec l'utilisateur, puis copier les cinq premiers éléments de cette liste dans une nouvelle liste, qui est renvoyée par la méthode. De cette manière, l'utilisateur recevra toujours des recommandations d'attractions, contrairement au fonctionnement précédent, pour lequel si aucune attraction n'était assez proche de l'utilisateur, aucune recommandation ne lui était envoyée.

3.3.3 TourGuideController.getNearbyAttractions

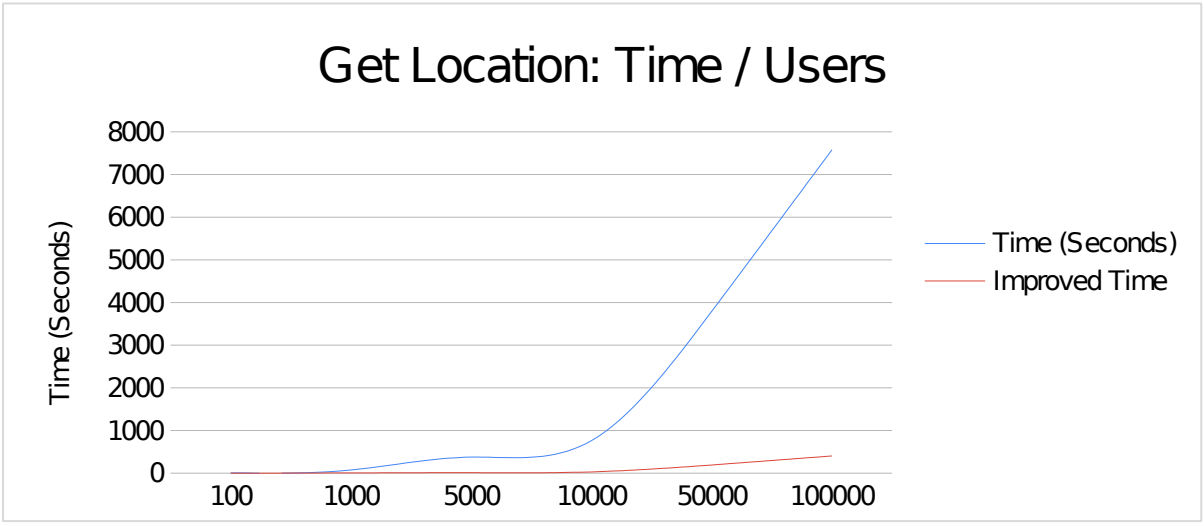
TourGuideController.getNearbyAttractions a été modifiée pour renvoyer un fichier JSON construit à partir de la liste renvoyée par TourGuideService.getNearbyAttractions. Ce fichier contient, pour chaque attraction : le nom de l'attraction, sa latitude, sa longitude, la latitude de l'utilisateur, la longitude de l'utilisateur, la distance entre l'utilisateur et l'attraction et le nombre de points à obtenir en visitant l'attraction. Renvoyer un fichier JSON permettra à l'application d'être plus modulable et de modifier séparément le front et le backend.

3.3.4 Temps de réponse

L'amélioration du temps de réponse de l'application a été réalisée par l'implémentation de parallèlisme dans les méthodes RewardsService.calculateRewards et TourGuideService.trackUserLocation, ces deux méthodes ayant des temps de réponses significativement plus long que les autres méthodes de l'application.

Le parallèlisme est réalisé via l'ajout à chacun de ces classes d'un ExecutorService. Un ExecutorService est un objet qui crée et gère un ensemble de threads, chaque thread pouvant exécuter des instructions de manière asynchrone par rapport aux autres. Ici les ExecutorServices seront de type FixedThreadPool, avec nombre de threads alloué de 50 pour RewardsService, et de 37 pour TourGuideService. Ces valeurs ont été déterminées de manière empirique.

Les instructions des méthodes RewardsService.calculateRewards et TourGuideService.trackUserLocation ont été réécrites dans un CompletableFuture, qui sera assigné à l'ExecutorService de la classe, ce dernier étant responsable de l'exécution des instructions. Cela permettra d'exécuter plusieurs appels à ces méthodes en parallèle sans avoir besoin d'attendre que le premier appel soit entièrement résolu avant de commencer à exécuter les suivants, donc de gagner du temps lors de pics d'utilisations de l'application.



3.3.5 Pipeline d'intégration continue

Pour assurer la qualité des futures modifications apportées au projet, une pipeline d'intégration continue a été mise en place. Elle consiste en un ensemble d'actions réalisée automatiquement lors du déploiement de nouvelles fonctionnalités assurant la fiabilité du code ajouté.

La pipeline a été réalisée en utilisant GitHub actions. Lors d'un push de code sur le repo GitHub du projet, l'application sera installée sur une machine virtuelle, les différents tests seront exécutés, et un .jar de cette version de l'application sera créé. Cela permettra de vérifier la conformité de toute modification apportée au projet, et de ne pas déployer de code défaillant dans l'application.

3.4 Autres solutions non retenues

3.4.1 TourGuideService.getNearbyAttractions

Le JSON renvoyé par TourGuideController.getNearbyAttractions contient plusieurs JSON contenant tous les informations latUser et longUser, en plus des informations relatives aux différentes attractions. Il a été envisagé de structurer différemment le JSON ({latUser, longUser, {attraction1}, {attraction2}, … }), pour qu'il ne contienne pas d'informations dupliquées, mais cela aurait rendu sa structure moins claire, et la quantité d'information économisée ne justifie pas l'augmentation de la complexité de manipulation de l'objet.

3.4.2 Temps de réponse

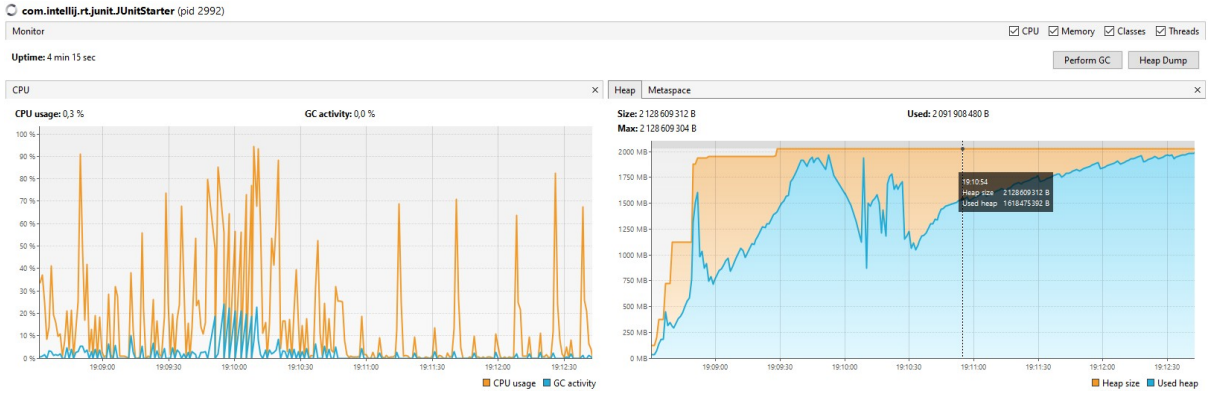
Plusieurs types d'ExecutorService ont été envisagés pour les classes TourGuideService et RewardsService. La classe Tracker utilise un ExecutorService de type SingleThread, mais celui-ci ne correspondait pas au besoin, qui nécessite d'exécuter des instructions sur plusieurs threads en même temps. Des essais ont été réalisés avec des CachedThreadPools, mais leur temps de réponse était supérieur à celui des FixedThreadPools.

Des essais d'amélioration du temps de réponse de la méthode RewardsService.calculateRewards ont été également été réalisés.

Cette méthode filtre des paires Attraction/VisitedLocation selon deux critères, et appelle user.addUserReward si les deux critères sont remplis. Les meilleurs résultats ont été obtenus avec la séparation de la fonction en trois CompletableFuture, les deux premiers vérifiant l'un des critères, le dernier appelant user.addUserReward après complétion des deux premiers. En faisant lancer des exceptions aux premiers en cas de non-respect du critère par la paire Attraction/VisitedLocation, il était possible d'arrêter le travail sur une paire dès qu'un des critères n'était pas rempli. Pour 1 000 utilisateurs, le temps de réponse passait de 13 secondes à 12 secondes, soit un gain d'environ 10 %. Ce gain reste minime quand on le compare à celui réalisé par la mise en place du parallèlisme, qui a déjà fait passer le temps de réponse pour 1000 utilisateurs de 472 secondes à 13 secondes.

Cependant cette méthode entraînait un problème d'allocation de mémoire, la mémoire heap se remplissant progressivement (cf capture d'écran), ce qui finissait par

entraîner des erreurs au-delà de 20 000 utilisateurs. Le temps de réponse avec un seul CompletableFuture réalisant une itération simple sur les deux listes correspondant aux critères de réussite, c'est cette solution qui a été retenue.



Capture d'écran de VisualVM illustrant le problème de mémoire

3.4.3 Pipeline d'intégration continue

D'autres services ont été envisagés pour la mise en place de la pipeline, notamment Gitlab-Ci et Jenkins. Ils avaient la capacité de répondre au besoin, mais GitHub Actions était en mesure d'y répondre sans avoir besoin de faire appel à un service supplémentaire, étant donné que GitHub est déjà utilisé dans le cadre du projet comme outil de versioning. Utiliser GitHub Actions permet également de simplifier l'architecture et d'éviter des problèmes éventuels de compatibilité entre les différents outils.

L'ensemble de la pipeline se déclenche lors d'un push, car son temps d'exécution est assez rapide, et que j'ai travaillé seul sur ce projet. Pour un projet mené par une équipe plus conséquente, ou pour une pipeline qui mettrait plus de temps à être exécutée, il serait recommandé de déclencher la pipeline lors d'une pull request, ce qui permettrait de l'annuler en cas d'échec de la pipeline, ou de séparer la pipeline en plusieurs modules : déclencher les tests unitaires lors d'un push, et déclencher les tests de performance uniquement lors d'une pull request.