TourGuide

Mise à l’échelle de TourGuide

Documentation fonctionnelle et technique

Sommaire

1. Présentation du projet
   - 1.1 Objectifs du projet
   - 1.2 Hors du champ d’application
   - 1.3 Mesures du projet
2. Spécifications fonctionnelles
3. Spécifications techniques
   - 3.1 Schémas de conception technique
   - 3.2 Glossaire
   - 3.3 Solutions techniques
   - 3.4 Autres solutions non retenues

1. Présentation du projet

Décrivez la solution proposée. Incluez le public cible et les avantages commerciaux de la solution.

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

Décrivez les objectifs du projet (2 à 3 phrases), y compris le(s) problème(s) résolu(s).

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 d'améliorer les performances de l'application, et de la rendre plus facilement adaptable à l'avenir. Ces objectifs seront notamment atteints via une refactorisation de l'application existante, et par la mise en place d'une pipeline d'intégration continue.

La méthode RewardsService.calculateRewards déclenche parfois une ConcurrentModificationException.

Le comportement de TourGuideService.getNearByAttractions doit être modifié. Elle doit retourner les cinq attractions les plus proches, au lieu de retourner toutes les attractions dans un certain rayon.

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 que nous ne disposons pas 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

Indiquez comment vous allez mesurer le succès du projet.

Le projet sera réussi si l'application se lance sans erreurs et en validant la suite de test développée par l'un de nos ingénieurs 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

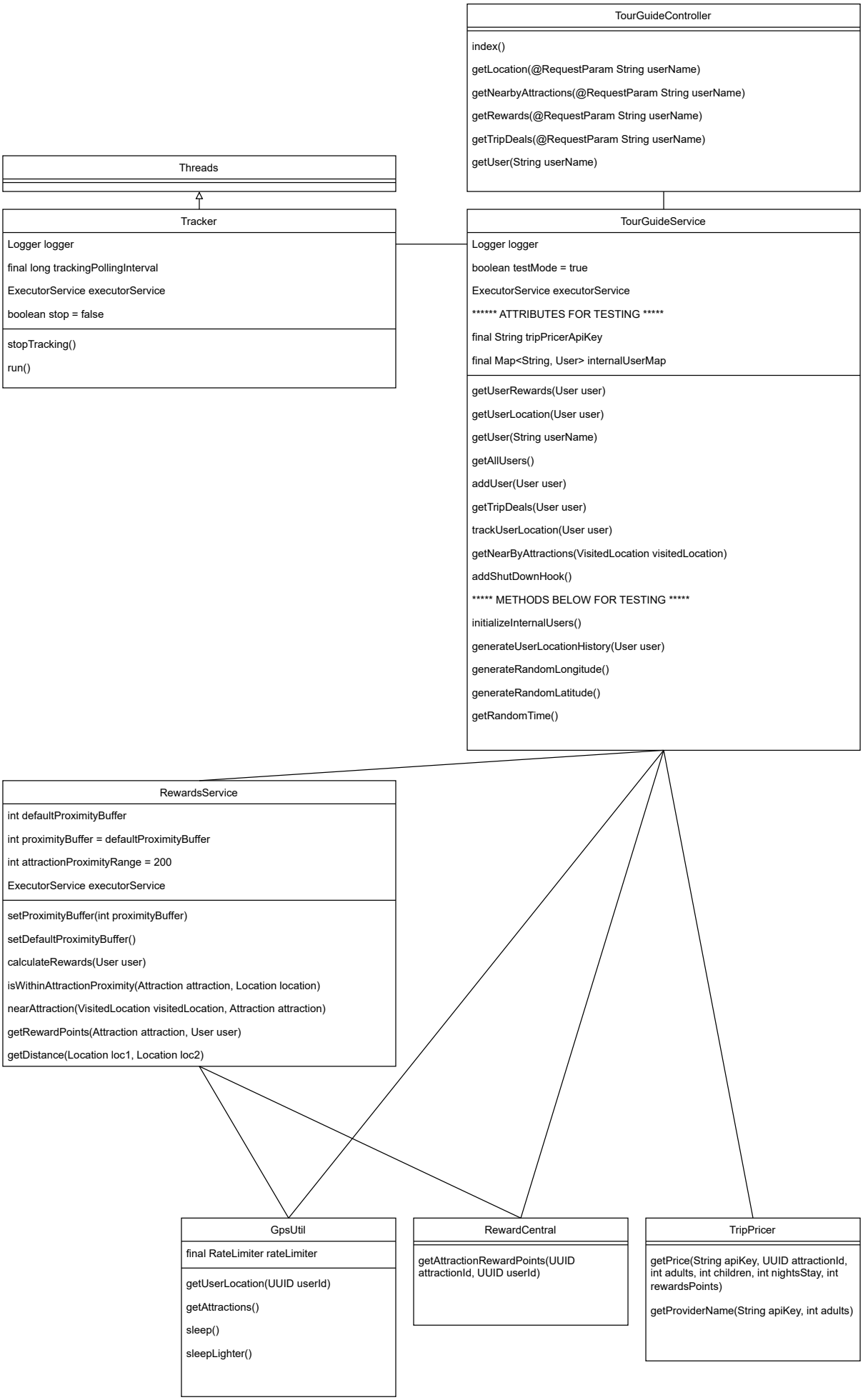
Faites figurer ici une liste de fonctionnalités.

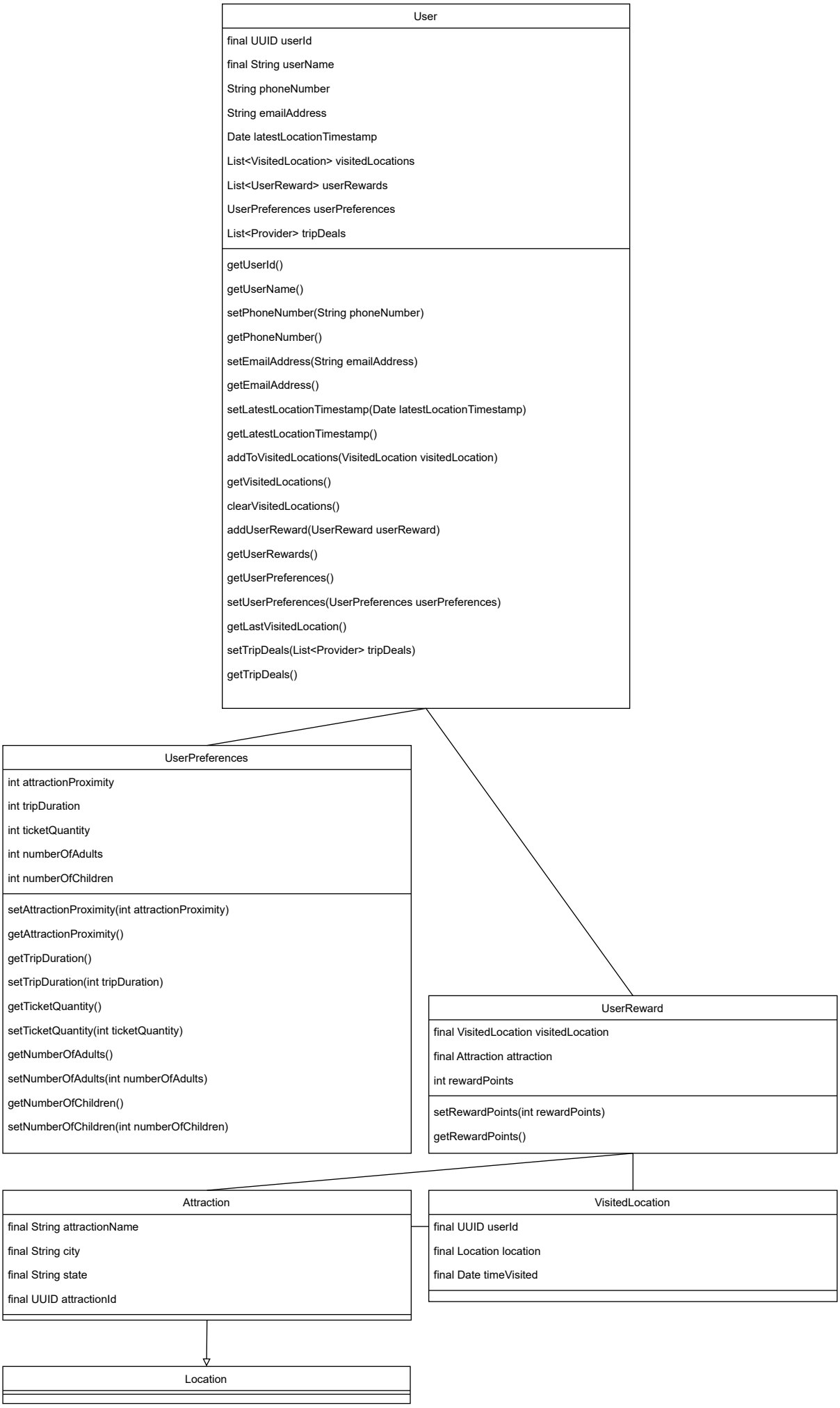
- afficher les attractions à proximité de l'utilisateur
- afficher des offres sur les attractions à l'utilisateur

3. Spécifications techniques

3.1 Schémas de conception technique

Commentez le schéma par une légende qui explique l'architecture technique de l'application.


TourGuideController
index()
getLocation(@RequestParam String userName)
getNearbyAttractions(@RequestParam String userName)
getRewards(@RequestParam String userName)
getTripDeals(@RequestParam String userName)
getUser(String userName)
Threads
Tracker
Logger logger
final long trackingPollingInterval
ExecutorService executorService
boolean stop = false
stopTracking()
run()
TourGuideService
Logger logger
boolean testMode = true
ExecutorService executorService
****** ATTRIBUTES FOR TESTING *****
final String tripPricerApiKey
final Map<String, User> internalUserMap
getUserRewards(User user)
getUserLocation(User user)
getUser(String userName)
getAllUsers()
addUser(User user)
getTripDeals(User user)
trackUserLocation(User user)
getNearByAttractions(VisitedLocation visitedLocation)
addShutDownHook()
***** METHODS BELOW FOR TESTING *****
initializeInternalUsers()
generateUserLocationHistory(User user)
generateRandomLongitude()
generateRandomLatitude()
getRandomTime()
RewardsService
int defaultProximityBuffer
int proximityBuffer = defaultProximityBuffer
int attractionProximityRange = 200
ExecutorService executorService
setProximityBuffer(int proximityBuffer)
setDefaultProximityBuffer()
calculateRewards(User user)
isWithinAttractionProximity(Attraction attraction, Location location)
nearAttraction(VisitedLocation visitedLocation, Attraction attraction)
getRewardPoints(Attraction attraction, User user)
getDistance(Location loc1, Location loc2)
GpsUtil
final RateLimiter rateLimiter
getUserLocation(UUID userId)
getAttractions()
sleep()
sleepLighter()
RewardCentral
getAttractionRewardPoints(UUID attractionId, UUID userId)
TripPricer
getPrice(String apiKey, UUID attractionId, int adults, int children, int nightsStay, int rewardsPoints)
getProviderName(String apiKey, int adults)


User
final UUID userId
final String userName
String phoneNumber
String emailAddress
Date latestLocationTimestamp
List<VisitedLocation> visitedLocations
List<UserReward> userRewards
UserPreferences userPreferences
List<Provider> tripDeals
getUserId()
getUserName()
setPhoneNumber(String phoneNumber)
getPhoneNumber()
setEmailAddress(String emailAddress)
getEmailAddress()
setLatestLocationTimestamp(Date latestLocationTimestamp)
getLatestLocationTimestamp()
addToVisitedLocations(VisitedLocation visitedLocation)
getVisitedLocations()
clearVisitedLocations()
addUserReward(UserReward userReward)
getUserRewards()
getUserPreferences()
setUserPreferences(UserPreferences userPreferences)
getLastVisitedLocation()
setTripDeals(List<Provider> tripDeals)
getTripDeals()
UserPreferences
int attractionProximity
int tripDuration
int ticketQuantity
int numberOfAdults
int numberOfChildren
setAttractionProximity(int attractionProximity)
getAttractionProximity()
getTripDuration()
setTripDuration(int tripDuration)
getTicketQuantity()
setTicketQuantity(int ticketQuantity)
getNumberOfAdults()
setNumberOfAdults(int numberOfAdults)
getNumberOfChildren()
setNumberOfChildren(int numberOfChildren)
UserReward
final VisitedLocation visitedLocation
final Attraction attraction
int rewardPoints
setRewardPoints(int rewardPoints)
getRewardPoints()
Attraction
final String attractionName
final String city
final String state
final UUID attractionId
VisitedLocation
final UUID userId
final Location location
final Date timeVisited
Location

3.2 Glossaire

Tout le vocabulaire du domaine se trouve ici.

Attraction : objet POJO représentant un site touristique, contenant le nom du site, le nom de la ville où il se trouve, le nom de l'état dans lequel il se trouve, et un identifiant unique.

Location : objet POJO représentant un endroit, contenant la latitude et la longitude de cet endroit.

UserReward : objet POJO représentant les points obtenus par un utilisateur à une Attraction, comprenant une VisitedLocation, l'Attraction, et le nombre de points.

VisitedLocation : objet POJO représentant la visite d'un utilisateur à une Location, contenant l'identifiant de l'utilisateur, la Location, et la date de la visite.

3.3 Solutions techniques

Expliquez quelles solutions techniques ont été utilisées pour répondre aux attentes.

Le problème 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 remplaçant la List<VisitedLocation> 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.

Le changement de comportement de TourGuideService.getNearbyAttractions a été réalisé en deux temps. Tout d'abord, 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. Ensuite, 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.

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.

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. Pour chacune d'elles, leurs instructions ont été réécrites dans un CompletableFuture, qui sera exécuté

par un ExecutorService de type FixedThreadPool, ce qui nous permettra d'exécuter plusieurs appels à ces méthodes en parallèle, donc de gagner du temps lors de pics d'utilisations de l'application. Le nombre de threads alloué est de 50 pour chaque ExecutorService.

3.4 Autres solutions non retenues

Expliquez les autres options envisagées pour la solution et pourquoi elles n'ont pas été choisies.

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 à notre besoin, puisque nous voulions exécuter des instructions sur plusieurs threads en même temps. Des essais ont été réalisés avec des CachedThreadPools, mais leur temps de réponse était trop long lorsque le nombre d'utilisateurs simultanés atteignait 100 000.

Des essais d'optimisation du temps de réponse de RewardsService.calculateRewards .

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 use.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 %.

Cependant cette méthode entraînait un problème d'allocation de mémoire, la mémoire heap se remplissant progressivement, ce qui finissait par entraîner des erreurs au-delà de 20 000 utilisateurs. Le temps de réponse avec une itération simple sur des listes correspondant aux critères de réussite, c'est cette solution qui a été retenue.

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.