Documentation du Système de Réservation de Billets Multithreadé

Cette documentation explique comment fonctionne le système de multithreadé de réservations.

1. Architecture des Services

TicketingService

C'est le service principal qui gère les réservations. Il utilise:

- **PriorityBlockingQueue** pour organiser les demandes par ordre de priorité dans la file d'attente
- **AtomicInteger** pour compter le nombre de billets disponibles sans risque de conflit entre les threads
- **StampedLock** pour empêcher plusieurs threads de réserver le même billet en même temps
- **CompletableFuture** pour le traitement asynchrone des demandes

ConcertBookingOrchestrator

C'est le service qui:

- Reçoit et distribue les demandes de réservation
- Contrôle que maximum 10 réservations sont traitées en même temps
- Envoie les notifications aux utilisateurs

NotificationService

Service simple qui envoie des notifications de manière asynchrone aux utilisateurs.

2. Gestion du Multithreading

Pourquoi AtomicInteger ?

On utilise AtomicInteger pour le nombre de billets disponibles car:

- Il garantit qu'un seul thread à la fois peut modifier le nombre de billets
- Si deux personnes réservent en même temps, on est sûr de ne pas vendre le même billet deux fois
- C'est plus rapide qu'un verrou classique pour des opérations simples comme +1 ou -1

```
private final AtomicInteger availableTickets;
```

Pourquoi StampedLock ?

Le StampedLock protège la réservation car:

- Il bloque l'accès aux autres threads pendant qu'une réservation est en cours
- Il permet à plusieurs threads de vérifier le nombre de billets en même temps
- Il garantit qu'une seule réservation est traitée à la fois dans la base de données

```
private final StampedLock ticketLock;
```

Pourquoi CompletableFuture ?

On utilise CompletableFuture pour:

- Traiter plusieurs réservations en parallèle
- Ne pas bloquer l'application pendant les réservations
- Pouvoir enchainer les étapes (réservation puis notification)

```
return CompletableFuture.supplyAsync(() -> processTicketRequest(request));
```

3. Synchronisation des Threads

La synchronisation se fait à plusieurs niveaux:

1. **Niveau File d'Attente**

```
private final PriorityBlockingQueue<UserRequest> requestQueue;
```

- File thread-safe
- Gère automatiquement la concurrence

2. **Niveau Réservation**

```
private boolean reserveTicket(UserRequest request) {
    long stamp = ticketLock.writeLock();
    try {
        // logique de réservation
    } finally {
        ticketLock.unlockWrite(stamp);
    }
}
```

- Verrou en écriture pendant la réservation
- Libération garantie du verrou

3. **Niveau Orchestrateur**

```
private final Semaphore concurrentBookings;
```

- Limite le nombre de réservations simultanées
- Evite la surcharge du système

C'est comme si on avait plusieurs guichets (threads) qui peuvent:

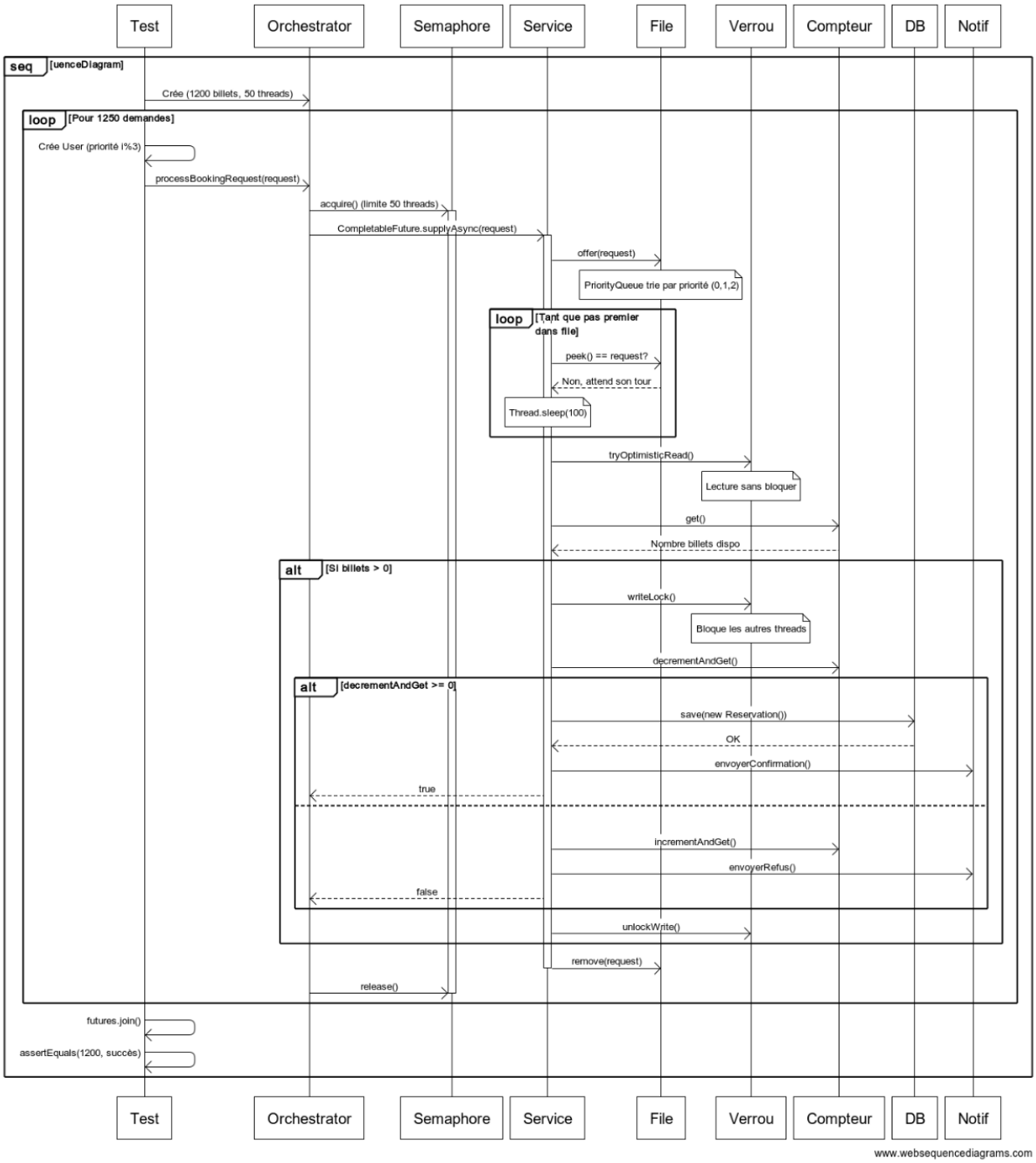
- Vérifier combien de billets sont disponibles (lecture)
- Réserver un billet (écriture)
- Tout ça sans risquer de vendre le même billet deux fois

4. Test de Performance

Le test ConcertBookingOrchestratorTest simule:

- 150 demandes de réservation pour 100 billets
- 10 réservations simultanées max
- Des priorités différentes (0, 1, 2)

Diagramme de séquence du système de réservation

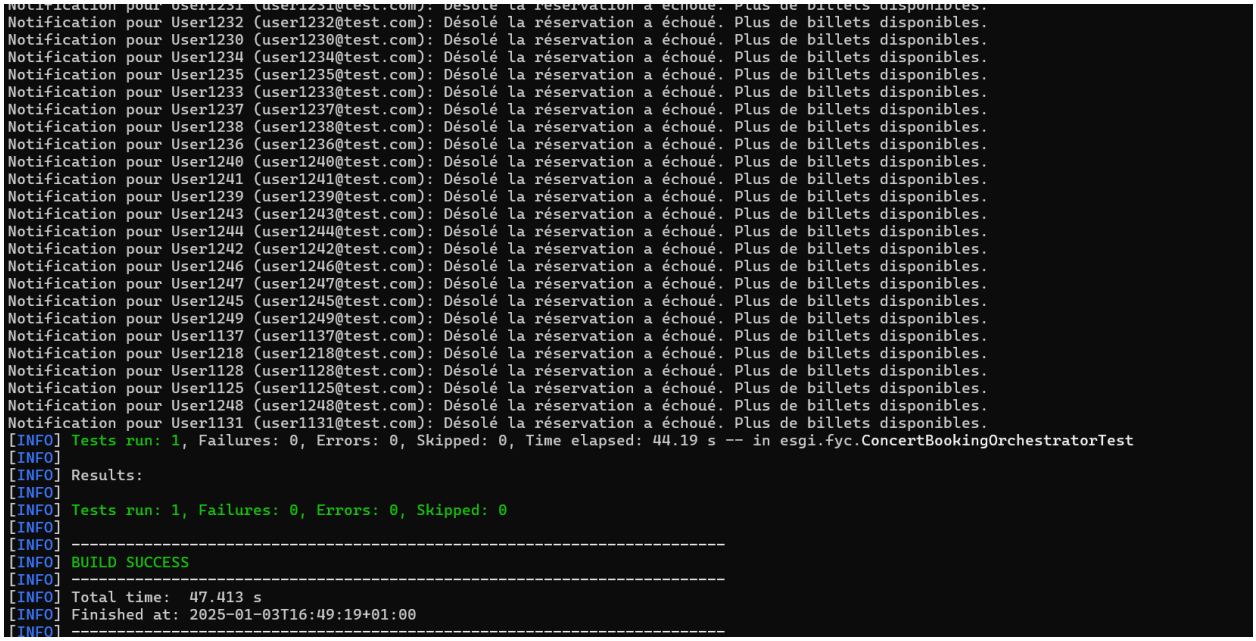


5. Résultats des Tests

Pour voir les résultats des tests:

1. Lancer mvn test
2. Le test vérifie que exactement 1200 billets sont réservés et 50 non

Screenshot des résultats:



On peut voir que d'une part les numéro de User dans notre test ne se suivent pas par exemple le User1200 a reservé son billet après le User1202 car le thread de User1202 a été plus rapide cela indique donc que le multithread fonctionne bien comme on le souhaite.

De plus notre test valide bien 1200 ticket et que 50 sont indisponibles.

Les logs montrent:

- Les notifications envoyées
- Les réservations réussies/échouées
- Le nombre final de réservations

Le test est réussi si:

- Exactement 100 réservations sont faites
- Pas d'erreur de concurrence
- Toutes les notifications sont envoyées