TRABAJO PRÁCTICO ESPECIAL 1

Primer Cuatrimestre de 2024

72.42 Programación de Objetos Distribuidos

Grupo 8

Ian James Arnott 61267
David Itcovici 61466
Lucas David Perri 62746
Santiago Luis Rivas Betancourt 61007

Índice

1. Decisiones de diseño e implementación de los servicios

1.1. Clases Propias

Contamos con las clases *Airline*, *Flight*, *Sector*, *Counter* y *Booking* para representar diferentes aspectos del aeropuerto, como las aerolíneas, vuelos, sectores, mostradores y reservas.

Las aerolíneas guardan la información de sus vuelos, mientras que los vuelos tienen su código y una referencia a la aerolínea a la cual pertenece. Los sectores guardan en un mapa los mostradores que les pertenecen. Estos mostradores guardan un rango de mostradores a la cual pertenecen si es que fueron asignados a un rango para ciertos vuelos de una aerolínea. Por su parte, las reservas incluyen un código único, el vuelo asociado y datos de check-in, útiles para determinar si se ha completado el proceso de check-in, así como en qué sector y mostrador se llevó a cabo.

1.2. Asignaciones Pendientes

Para resolver las asignaciones pendientes se tomó la decisión de que, al liberarse o crearse un rango de mostradores en el sector con asignaciones pendientes, se tratara de resolver la primera, luego la segunda y así hasta que no queden más asignaciones pendientes. Estas asignaciones se guardan en un *ConcurrentLinkedQueue*, donde los elementos de la misma son una clase propia *PendingAssignments*, así mantiene el orden de los pedidos.

1.3. Stubs

Se decidió utilizar *FutureStub* para los clientes debido a la importancia de manejar la asincronía y tener un control más preciso sobre la concurrencia. Esta elección proporciona ventajas frente a la simplicidad del código y la sincronía que ofrece *BlockingStub*.

2. Criterios aplicados para el trabajo concurrente

2.1. Latch

En los clientes, se utilizan instancias de *Latch* para evitar que los mismos finalicen su ejecución hasta no haber recibido la respuesta del servidor.

2.2. Synchronized

Para evitar que dos clientes de los servicios accedan o escriban información que pasa por el *AirportRepository*, se decidió que varios de los métodos del mismo sean *synchronized.* De esta manera el segundo hilo que llama un método del repositorio, que fue llamada por el primer hilo, deberá esperar a que el primero termine para luego entrar. De esta manera el *AirportRepository* actúa como el mediador y evita situaciones de race condition.

2.3. Records

Se decidió utilizar records para devolver información desde el AirportRepository al servicio. Al trabajar con objetos inmutables, los servicios pueden ofrecer respuestas precisas y confiables a los clientes, evitando riesgos de alteraciones no deseadas durante el proceso de generación de la respuesta. Esta práctica mejora la estabilidad y seguridad de los servicios.

Originalmente, antes de un refactor, se retornaban los objetos en ciertas situaciones, y esto llevaba a situaciones de race condition. Para evitar esto, se decidió solo retornar records que solo contienen la información necesaria, y son inmutables.

3. Potenciales puntos de mejora y/o expansión

3.1. Persistencia

La clase *AirportRepository* actúa como un repositorio para los datos. Si bien cumple con lo pedido, solo retiene la información durante una misma sesión. Como futura expansión y mejora, se podría implementar una base de datos real para mantener la persistencia.

3.2. Comunicación con Streaming

En el caso de agregar vuelos, aerolíneas y reservas, se podría implementar usando streaming, de manera que se envíe el contenido del .csv como stream al servidor. Esperando que el servidor devuelva la información de cuales se agregaron y cuáles no.

3.3. Mejor implementación de Modelos

Un refactor de los modelos actuales para mejorar su diseño y cómo se maneja la concurrencia.

Uno de los cambios más importantes es modificar cómo se estructuran los sectores, haciendo que guarden una lista de *CounterRange* en lugar de mostradores individuales. Esto ayudará a eliminar los bucles entre las clases y simplificará la gestión de la concurrencia. Al limitar los efectos secundarios de los métodos, será más claro qué partes del sistema están siendo modificadas en cada operación.

Además, este cambio en los modelos permite reducir la necesidad de utilizar *synchronized* en las definiciones de los métodos. Así facilitando el manejo de los bloqueos de lectura y escritura de una forma más eficiente, en vez de bloquear todas las veces que un segundo hilo acceda al *AiportRepository*.

3.4. Mejor uso de las Clases Concurrentes

Luego de un refactor que se hizo durante la implementación del trabajo, muchas clases quedaron como concurrentes, cuando no deberían serlo ya que se maneja la concurrencia desde el *AirportRepository*. Esto puede traer problemas de eficiencia ya que se utilizan clases que tienen más funcionalidades de las que realmente se necesita. Una mejora sería realizar un refactor cambiando las colecciones concurrentes utilizadas en secciones ya bloqueadas por el repositorio donde la concurrencia no es un problema porque está bajo un contexto *synchronized.*