Decisiones de diseño

---

1. Manejo de concurrencia con `ThreadPoolExecutor`

Decisión:

Se utilizó un `ThreadPoolExecutor` personalizado para manejar las tareas concurrentes relacionadas con el procesamiento de órdenes.

Por qué:

- Permite controlar el número de hilos concurrentes (`corePoolSize` y `maxPoolSize`) para optimizar el rendimiento y evitar la sobrecarga del sistema.
- Se valida la capacidad de la cola antes de aceptar nuevas tareas, previniendo errores de saturación y asegurando que las solicitudes no se pierdan.
- El uso de `CompletableFuture` junto con el `ThreadPoolExecutor` permite manejar tareas de forma asíncrona, lo que mejora la escalabilidad.

Análisis **CompletableFuture**

- **Integración sencilla:** No requiere dependencias externas adicionales.
- **Soporte nativo:** Es parte de Java desde la versión 8, muy accesible y estándar.
- **Versatilidad:** Permite manejar tareas asíncronas de manera flexible
- **Aplicaciones prácticas:**
    - Llamadas concurrentes a servicios externos.
    - Ejecución de tareas independientes que deben combinarse después.

Cuándo usarlo:

- Cuando necesitas manejar tareas asíncronas con dependencias entre ellas.
- Para aplicaciones con requisitos simples de concurrencia que no requieren complejidad adicional.
- Si estás procesando múltiples solicitudes HTTP o consultas a servicios que pueden ejecutarse en paralelo.

Impacto de estas decisiones

- **Escalabilidad:** El uso de `ThreadPoolExecutor` y `CompletableFuture` permite manejar eficientemente grandes volúmenes de solicitudes concurrentes.
- **Resiliencia:** El manejo de excepciones asegura que el sistema siga funcionando incluso ante errores.
- **Modularidad:** El uso de servicios y estrategias desacoplados mejora la claridad y facilita las pruebas.
- **Simulación realista:** Los retrasos simulados ayudan a modelar entornos de producción y validan la capacidad del sistema bajo carga.

Análisis

Técnica	Ventajas	Cuándo Usarla
CompletableFuture	Fácil de integrar, no requiere dependencias adicionales.	Cuando necesitas manejar tareas asíncronas y combinarlas, como llamadas a servicios o cálculos.
ForkJoinPool	Excelente para tareas computacionalmente intensivas que pueden dividirse en sub-tareas.	Para tareas paralelas CPU-bound, como procesamiento masivo de datos en memoria.
Reactor	Soporte para procesamiento asíncrono y no bloqueante; ideal para aplicaciones reactivas.	Cuando estás desarrollando aplicaciones con alta concurrencia o quieres integrarte con WebFlux.

Comparación General

- **CompletableFuture:** Ideal para proyectos estándar que requieren concurrencia básica sin complejidad añadida.
- **ForkJoinPool:** Excelente para cargas computacionales intensivas y procesos paralelos en memoria.
- **Reactor:** Perfecto para aplicaciones reactivas o sistemas que manejan miles de conexiones simultáneas.

Conclusión:

Estas técnicas abordan diferentes necesidades:

1. **CompletableFuture** para tareas asíncronas simples y manejables.
2. **ForkJoinPool** para procesamiento paralelo CPU-bound.
3. **Reactor** para sistemas reactivos y de alta concurrencia.

Por qué no necesitas 1000 hilos:

1. **Uso de recursos:**
    - Cada hilo consume memoria y CPU.
    - Con 1000 hilos, puedes saturar el sistema debido a cambio de contexto y recursos limitados.
2. **Tareas concurrentes:**
    - Si las tareas son intensivas en **CPU**, el número óptimo de hilos debe ser cercano al número de núcleos. Fórmula: Hilos o´ptimos=Nu´cleos CPU×(1+Factor de espera)\text{Hilos óptimos} = \text{Núcleos CPU} \times (1 + \text{Factor de espera})Hilos o´ptimos=Nu´cleos CPU×(1+Factor de espera) Ejemplo: En un sistema con **16 núcleos** y sin tiempo de espera: Hilos o´ptimos=16×(1+0)=16\text{Hilos óptimos} = 16 \times (1 + 0) = 16Hilos o´ptimos=16×(1+0)=16
    - Si las tareas son intensivas en **E/S** (espera), puedes usar más hilos. Por ejemplo, con un factor de espera del 0.9: Hilos o´ptimos=16×(1+0.9)=30 (aproximadamente)\text{Hilos óptimos} = 16 \times (1 + 0.9) = 30 \text{ (aproximadamente)}Hilos o´ptimos=16×(1+0.9)=30 (aproximadamente)
3. **Cola como amortiguador:**
    - Una cola permite manejar picos de carga. Por ejemplo, con **500 hilos** y una cola de **500**, puedes procesar hasta **1000 tareas** sin problemas.

Ejemplo con cálculos:

- **Tareas concurrentes:** 1000
- **Duración promedio de cada tarea:** 100 ms
- **Sistema:** 16 núcleos CPU

1. **Con 500 hilos:**
    - Cada hilo procesa 10.1=10\frac{1}{0.1} = 100.11=10 tareas por segundo.
    - Con 500 hilos: 500×10=5000500 \times 10 = 5000500×10=5000 tareas por segundo.
    - En este caso, las 1000 tareas se procesan en **0.2 segundos**.
2. **Con 1000 hilos:**
    - La cantidad de cambio de contexto y uso de memoria puede causar una reducción en el rendimiento.
    - Si el sistema se satura, el tiempo total podría aumentar.

Configuración recomendada:

- `maxPoolSize = 500`
- `queueCapacity = 500`

Esto te asegura procesar las tareas concurrentes sin saturar el sistema. Realiza pruebas para ajustar valores si necesitas mayor rendimiento.

---

2. Gestión de estados de órdenes

Decisión:

Se delega la determinación del estado de una orden al `OrderStatusStrategyManager`.

Por qué:

- Utilizar un patrón de estrategia (`Strategy Pattern`) centraliza la lógica de negocio para determinar estados de órdenes, lo que facilita la modificación o extensión de comportamientos sin cambiar el código principal.
- Mejora la separación de responsabilidades y facilita pruebas unitarias.

---

3. Simulación de retrasos de procesamiento

Decisión:

Se agregó un retraso controlado (`Thread.sleep`) para simular tiempos de procesamiento realistas.

Por qué:

- Ayuda a modelar escenarios del mundo real, como tiempos de red o de procesamiento interno, para pruebas y validación.
- Mejora la capacidad de realizar simulaciones bajo carga para medir la capacidad del sistema y la efectividad de la concurrencia.

---

4. Persistencia con `OrderRepository`

Decisión:

Se utiliza `OrderRepository` para verificar el estado de procesamiento y persistir las órdenes procesadas.

Por qué:

- Garantiza la persistencia de los datos procesados en la capa de almacenamiento, evitando pérdidas de datos.
- La verificación previa (`isProcessed`) asegura consistencia y evita el reprocesamiento de órdenes.

Concurrencia segura

Razón:

`ConcurrentHashMap` es una implementación de un mapa que permite múltiples accesos concurrentes sin bloquear completamente el mapa. Esto significa que varios hilos pueden leer y escribir en el mapa de manera eficiente.

Beneficio:

- Evita problemas como condiciones de carrera o inconsistencias de datos.
- Es ideal para escenarios donde múltiples hilos necesitan acceder al repositorio para leer o modificar datos simultáneamente.
