

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.

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.
-