**Documento de Arquitectura y Decisiones de Diseño**

**1. Arquitectura Hexagonal (Puertos y Adaptadores)**

**Justificación**

He elegido la Arquitectura Hexagonal como base para este microservicio. El objetivo principal era **desacoplar el núcleo de la lógica de negocio (dominio) de los detalles de la infraestructura** (base de datos, API REST, servicios externos). Esto nos permite:

* **Testear el dominio de forma aislada**, sin necesidad de levantar una base de datos o un servidor web, lo que agiliza y simplifica las pruebas unitarias.
* **Intercambiar tecnologías fácilmente**. Podemos cambiar de H2 a PostgreSQL, o de REST a gRPC, modificando únicamente los "adaptadores" en la capa de infraestructura, sin tocar una sola línea de la lógica de negocio.
* **Mejorar la mantenibilidad y escalabilidad** a largo plazo, ya que las responsabilidades están claramente delimitadas.

**Suposiciones**

* He asumido que este microservicio no vivirá aislado y que en el futuro necesitará integrarse con otros sistemas (como el de inventario, notificaciones, etc.). El modelo de puertos y adaptadores es ideal para estas integraciones.
* He supuesto que los requisitos tecnológicos podrían cambiar, por lo que era crucial no acoplar la lógica de negocio a un framework o base de datos específicos.

**Trade-offs (Compensaciones)**

* La principal compensación es un **aumento en la complejidad inicial y el número de clases** (DTOs, Mappers, Puertos, Adaptadores) en comparación con una arquitectura por capas tradicional. Este "coste" inicial es una inversión deliberada para ganar flexibilidad y robustez a futuro.

**2. Domain-Driven Design (DDD) y Principios SOLID**

**Justificación**

He aplicado principios de DDD para modelar un dominio rico y expresivo. El Order no es solo un contenedor de datos, sino un **Agregado (Aggregate Root)** que contiene y protege la lógica de negocio y sus reglas (invariantes). Esto se complementa con:

* **Máquina de Estados en el Agregado**: La lógica de transición de estados (process(), ship(), etc.) reside en la propia clase Order, garantizando que nunca se pueda llegar a un estado inválido.
* **Principios SOLID**: La arquitectura promueve la Inversión de Dependencias (los servicios dependen de interfaces/puertos, no de clases concretas) y la Responsabilidad Única (cada clase tiene un propósito bien definido).

**Suposiciones**

* He asumido que la lógica de gestión de pedidos era lo suficientemente compleja como para justificar un modelo de dominio rico. Para un simple CRUD, DDD podría haber sido excesivo.

**Trade-offs (Compensaciones)**

* DDD requiere una disciplina estricta y un buen entendimiento del negocio. El modelado inicial lleva más tiempo que en un enfoque centrado en la base de datos (anemic domain model).

**3. Gestión de Concurrencia: Bloqueo Optimista**

**Justificación**

Para un sistema de e-commerce de alto tráfico, el rendimiento es clave. He elegido el **bloqueo optimista** (@Version) sobre el pesimista (SELECT ... FOR UPDATE) porque:

* **No bloquea la base de datos**: Permite una mayor concurrencia y evita cuellos de botella, ya que no se mantienen bloqueos en las filas mientras dura una transacción.
* **Se alinea con el caso de uso**: Asume que los conflictos (dos usuarios modificando el mismo recurso a la vez) son la excepción, no la regla, lo cual es típico en la gestión de pedidos e inventario.

**Suposiciones**

* He asumido que la tasa de conflictos de escritura concurrentes sobre la misma entidad (Order o Product) será baja. Si fuera alta, el bloqueo pesimista podría ser más adecuado, aunque a costa del rendimiento.

**Trade-offs (Compensaciones)**

* La lógica de manejo de errores se traslada del nivel de la base de datos a la aplicación. Mi código es ahora responsable de **capturar la ObjectOptimisticLockingFailureException** y decidir cómo manejar el conflicto (ej. notificar al usuario para que reintente la operación).

**4. Seguridad: Autenticación JWT Stateless**

**Justificación**

He implementado un sistema de autenticación sin estado (stateless) usando JSON Web Tokens (JWT).

* **Escalabilidad Horizontal**: Al ser stateless, no necesitamos un almacén de sesiones compartido. Cualquier instancia del microservicio puede validar un token, lo que facilita enormemente el escalado.
* **Estándar para Microservicios**: Es el enfoque estándar en arquitecturas de microservicios, ya que los tokens se pueden compartir y validar fácilmente entre diferentes servicios.

**Suposiciones**

* He asumido que no se requiere la capacidad de invalidar un token de forma inmediata en el servidor (ej. "cerrar sesión" forzosamente antes de que el token expire).

**Trade-offs (Compensaciones)**

* **Invalidación de Tokens**: La mayor desventaja de JWT es que, una vez emitido, un token es válido hasta su expiración. Implementar una lista negra (blacklist) para la invalidación forzada añade una complejidad que decidí no abordar en esta etapa.
* **Tamaño del Token**: El token puede crecer si se le añade mucha información (claims), aumentando ligeramente el tamaño de cada petición.