Proyecto con Modulith

Módulos y manejo de eventos para la comunicación módulos

Academia Java

Entregable

Natalia Esquivel Ocadiz

26 de septiembre 2025

Introducción

El proyecto desarrollado consiste en un sistema de gestión de pedidos, inventario y proveedores, implementado con Spring Boot bajo un enfoque de arquitectura monolítica modular. La necesidad principal era organizar las diferentes áreas de un sistema de almacén dentro de una sola aplicación, pero manteniendo cierta independencia entre ellas. El problema inicial que se resolvió fue cómo lograr la comunicación entre módulos (por ejemplo, que inventario se entere de lo que pasa en pedidos) sin que quedaran acoplados directamente, además de evitar errores comunes al trabajar con relaciones entre entidades y su serialización a JSON.

Objetivos

Los objetivos técnicos de este proyecto fueron:

1. Crear una aplicación modular en un contexto monolítico, dividiendo el código en módulos como orders, inventory, suppliers y shared.
2. Definir entidades, repositorios, servicios y controladores siguiendo la arquitectura en capas.
3. Manejar la comunicación entre módulos mediante eventos en lugar de llamadas directas.
4. Resolver el problema de las relaciones bidireccionales (entre Order y LineOrder) que provocaban ciclos infinitos en la serialización JSON.

Implementación

La aplicación se estructuró en varios módulos:

- Orders: gestiona pedidos, con entidades como Order y LineOrder. Publica eventos (OrderCreated, OrderCanceled, OrderCompleted) que representan los cambios en el ciclo de vida de un pedido.
- Inventory: gestiona productos mediante la entidad Producto y su repositorio. Cuenta con un manejador de eventos (ManejadorEventoOrders) que reacciona a los eventos de pedidos para actualizar el inventario.
- Suppliers: administra proveedores (Supplier) y catálogos de productos (CatalogoProducto), con su servicio y controlador propios.
- Shared: contiene elementos compartidos, como la clase DomainEvent, que es la base de los eventos de dominio.

La comunicación entre módulos se realizó mediante Spring Modulith, usando la anotación @ApplicationModuleListener. Como, cuando se completa un pedido, el módulo de pedidos publica un evento OrderCompleted. El módulo de inventario escucha ese evento y decide cómo reaccionar que es reduciendo el stock del producto seleccionado. Esto garantiza que los módulos estén desacoplados, ya que orders no necesita conocer ni invocar directamente a InventarioService.

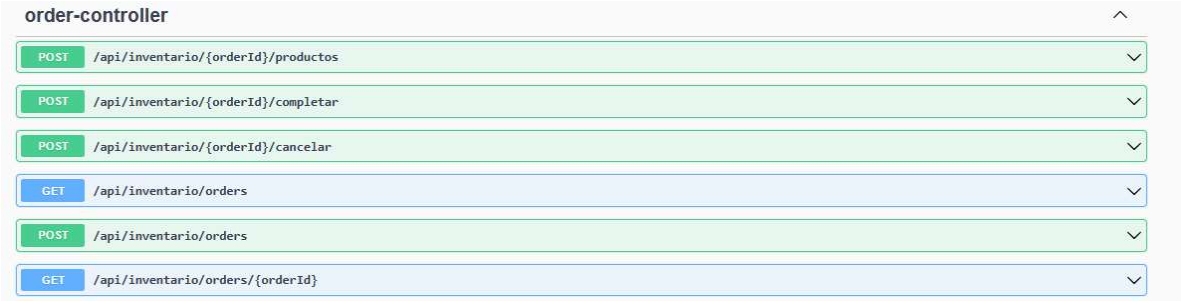
Durante el desarrollo, hubo un problema inicial debido a las relaciones bidireccionales entre Order y LineOrder. Inicialmente, al devolver un pedido desde el controlador, Jackson (conversión de objetos Java a JSON y viceversa) entraba en un ciclo infinito al intentar serializar Order a LineOrder y viceversa provocando un bucle. Este problema se resolvió usando la anotación @JsonIgnore en la columna (OneToMany), permitiendo que solo una vez se escribiera JSON y evitando errores de serialización.

Respecto al flujo de trabajo, los controladores reciben las peticiones, y el servicio se encarga de la Lógica del Negocio y este accede al repositorio para conectar con la base de datos. Los eventos permiten que los cambios en un módulo (como la creación de un pedido, OrderCreated) puedan ser escuchados por otros módulos (como inventario), sin necesidad de estar acoplados.

Resultados

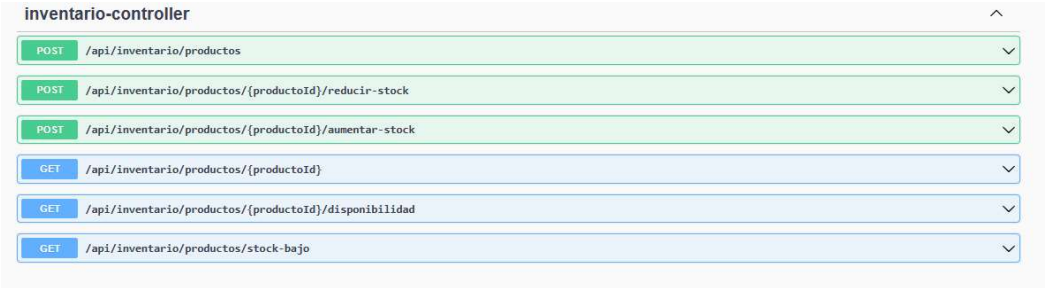
El sistema resultante permite:

- Crear, cancelar y completar pedidos.



- Registrar proveedores

- Administrar productos en inventario con las operaciones básicas (agregar o reducir stock)



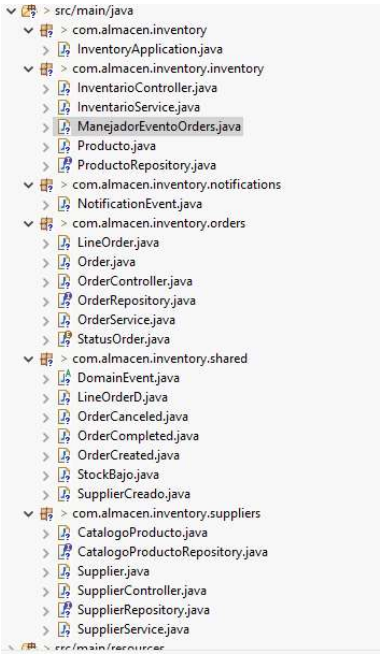
- Comunicar pedidos e inventario de forma modular gracias a la implementación de eventos.

```
⚠ ALERTA: Stock bajo detectado!
   Producto: Monitor Samsung 27
Buscando proveedores para: Monitor Samsung 27
 Consultando base de datos de proveedores...
Contactando proveedores disponibles...
   Stock actual: 2
   Stock mínimo: 3
```

- Evitar o corregir problemas de serialización JSON en relaciones bidireccionales más fácilmente mediante el uso de anotaciones de Jackson.

  `@JsonManagedReference (@OneToMany) y` `@JsonBackReference (@ManyToOne)`

Estructura:



El uso de eventos de dominio demostró ser una solución efectiva para mantener el monolito organizado, ya que permite que cada módulo trabaje de manera independiente, pero de forma coordinada. Por ejemplo, el inventario puede reaccionar a la finalización de una compra o pedido sin que el servicio de pedidos tenga que llamarlo directamente.

En consola, todo funciona correctamente:

```
        Maximum pool size: undefined/unknown
2025-09-27T22:00:20.808-06:00  INFO 4460 --- [inventory] [           main] o.h.e.t.j.p.i.JtaPlatformInitiator
2025-09-27T22:00:20.935-06:00  INFO 4460 --- [inventory] [           main] j.LocalContainerEntityManagerFactoryBean
2025-09-27T22:00:21.315-06:00  INFO 4460 --- [inventory] [           main] o.s.d.j.r.query.QueryEnhancerFactory
2025-09-27T22:00:21.896-06:00  WARN 4460 --- [inventory] [           main] JpaBaseConfiguration$JpaWebConfiguration
2025-09-27T22:00:21.981-06:00  INFO 4460 --- [inventory] [           main] o.s.v.b.OptionalValidatorFactoryBean
2025-09-27T22:00:23.130-06:00  INFO 4460 --- [inventory] [           main] o.s.b.w.embedded.tomcat.TomcatWebServer
2025-09-27T22:00:23.141-06:00  INFO 4460 --- [inventory] [           main] o.s.s.config.TaskSchedulerRouter
2025-09-27T22:00:23.146-06:00  INFO 4460 --- [inventory] [           main] c.a.inventory.InventoryApplication
Hibernate: insert into suppliers (address,email,phone,supplierName) values (?,?,?,?)
Hibernate: insert into suppliers (address,email,phone,supplierName) values (?,?,?,?)
Hibernate: insert into suppliers (address,email,phone,supplierName) values (?,?,?,?)
Hibernate: insert into suppliers (address,email,phone,supplierName) values (?,?,?,?)
Hibernate: insert into suppliers (address,email,phone,supplierName) values (?,?,?,?)
Hibernate: select count(*) from productos p1_0
```

Reflexiones

Este proyecto representó un primer acercamiento a la idea de monolito modular, mostrando que no es necesario moverse a microservicios para obtener una arquitectura más organizada y desacoplada. Los eventos fueron clave para lograr ese desacoplamiento buscado.

Los principales aprendizajes fueron:

- Comprender la importancia de limitar las dependencias entre módulos para mejorar el mantenimiento del código para futuras implementaciones
- Conocer las anotaciones de Jackson para resolver problemas prácticos con las relaciones de JPA y la serialización a JSON.
- Confirmar que Spring Modulith facilita la organización del código y la gestión de eventos en proyectos monolíticos.

Como posibles mejoras, se podrían implementar DTOs en un futuro, agregar pruebas unitarias por módulo para validar el correcto comportamiento de cada componente de manera aislada y ampliar el sistema con nuevos módulos como notificaciones o facturación, aprovechando el modelo basado en eventos.