ms-products-orders: Microservicio de Gestión de Productos y Órdenes

Este repositorio contiene el código fuente del microservicio ms-products-orders, una API REST diseñada para la gestión integral de productos y órdenes. Desarrollado con Spring Boot, este servicio se conecta a una base de datos MySQL y ofrece funcionalidades para catalogar productos y procesar pedidos.

Contenido

Características Principales

- **API RESTful:** Exposición de endpoints para la gestión de productos y órdenes.
- **Persistencia de Datos:** Integración con MySQL para almacenamiento de información.
- **Validación de Datos:** Uso de validaciones para asegurar la integridad de la información.
- **Manejo de Circuit Breaker:** Implementación de Resilience4j para tolerancia a fallos.

Tecnologías Utilizadas

- **Java 17**
- **Spring Boot 3.2.5**
- **Maven 3.8+**
- **MySQL**
- **Spring Data JPA**
- **Lombok**
- **Spring Cloud Netflix Eureka Client (para descubrimiento de servicios, si aplica)**
- **Spring Cloud CircuitBreaker Resilience4j**
- **JUnit 5 / Mockito** (para pruebas)
- **JaCoCo** (para cobertura de código)
- **SonarQube** (para análisis de calidad de código)

Pre-requisitos

Antes de comenzar, asegúrate de tener instalados los siguientes componentes en tu sistema:

- **Java Development Kit (JDK) 17 o superior:**
- **Apache Maven 3.8 o superior:**
- **Servidor MySQLWorkbench:**
- **Postman**

Guía de Configuración y Ejecución

Sigue estos pasos para poner en marcha el proyecto en tu entorno local.

1. Clonar el Repositorio

Abre tu terminal (Git Bash, PowerShell, CMD) y ejecuta los siguientes comandos:

Bash

git clone <URL_DEL_REPOSITORIO> # Reemplaza con la URL real de tu repositorio

cd ms-products-orders/ms-products-orders

Nota: El proyecto está estructurado con un directorio raíz ms-products-orders que contiene el módulo principal ms-products-orders. Asegúrate de navegar a la carpeta del módulo para ejecutar los comandos de Maven.

2. Configuración de la Base de Datos

2.1 Crear la Base de Datos

Conéctate a tu servidor MySQL (usando MySQL Workbench, PhpMyAdmin, o la consola de MySQL) y crea una nueva base de datos. Por ejemplo:

SQL

CREATE DATABASE ms_products_orders_db;

2.2 Configurar las Credenciales en el Proyecto

Edita el archivo de configuración principal de la aplicación:

src/main/resources/application.properties

Actualiza las propiedades de conexión a la base de datos con tus credenciales de MySQL:

Properties:

spring.datasource.url=jdbc:mysql://localhost:3306/ms_products_orders_db?useSSL=false&serverTimezone=UTC

spring.datasource.username=TU_USUARIO_MYSQL

spring.datasource.password=TU_PASSWORD_MYSQL

spring.jpa.hibernate.ddl-auto=update

spring.jpa.show-sql=true

spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect

Importante: La propiedad spring.jpa.hibernate.ddl-auto=update configurará automáticamente el esquema de la base de datos la primera vez que inicies la aplicación. Para entornos de producción, considera cambiar esto a validate o none después de la creación inicial.

3. Construcción del Proyecto

Desde la carpeta del módulo (ms-products-orders/ms-products-orders), ejecuta el siguiente comando Maven para descargar las dependencias y compilar el proyecto:

Bash

mvn clean install

Este comando también ejecutará las pruebas unitarias por defecto.

4. Ejecución de la Aplicación

Una vez que el proyecto se ha construido exitosamente, puedes iniciar la aplicación desde la misma carpeta del módulo:

Bash

mvn spring-boot:run

Si la aplicación se inicia correctamente, verás un mensaje similar a este en la terminal:

Started MsProductsOrdersApplication in X seconds (JVM running for Y)

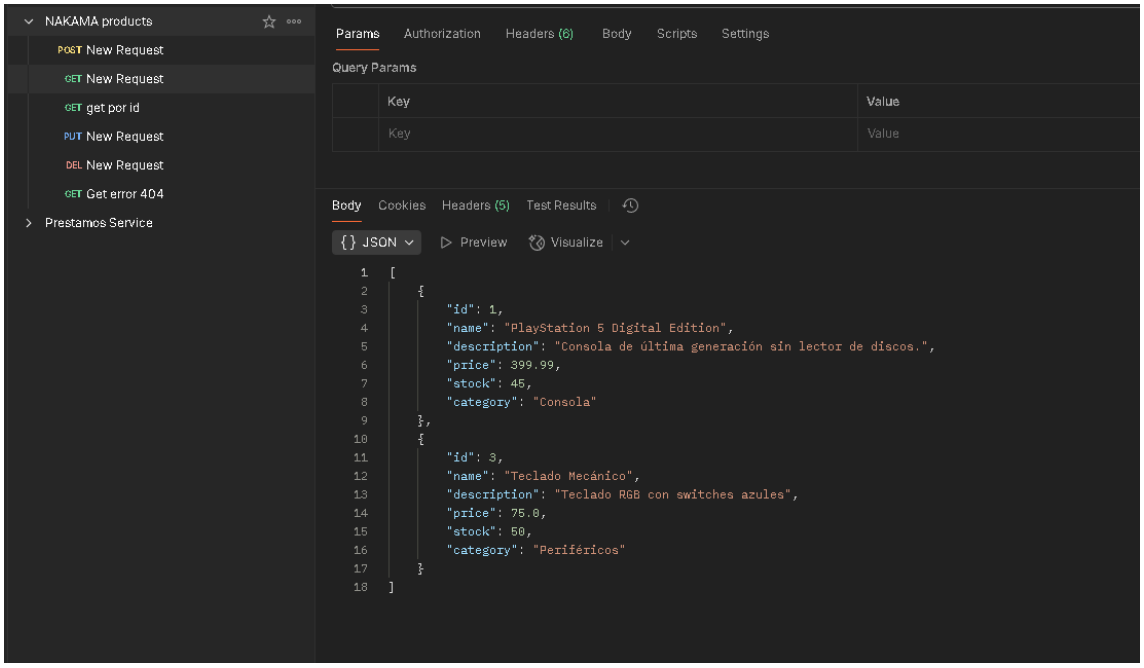
La API estará disponible en la siguiente URL base: http://localhost:8080

Pruebas y Verificación

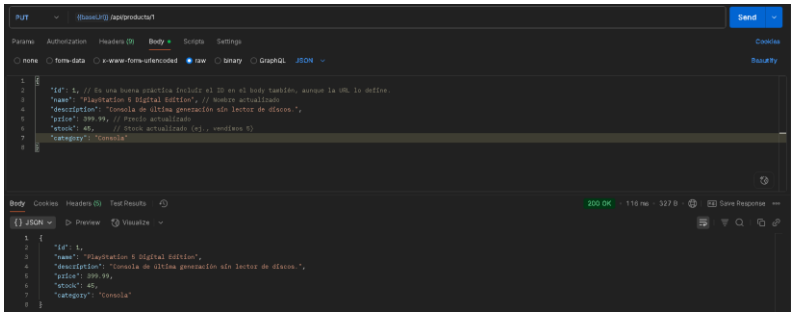
Probar los Endpoints

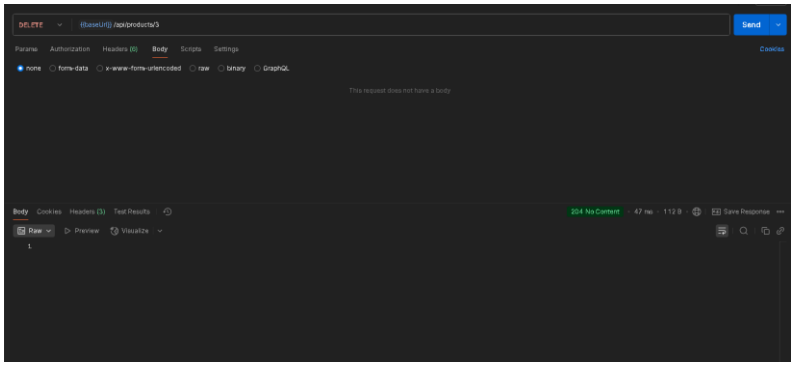
Una vez que la aplicación esté corriendo, puedes probar los endpoints utilizando Postman

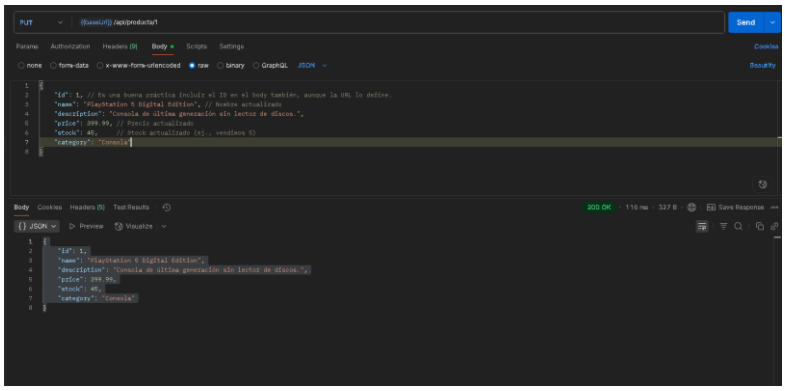
Adjunto imágenes de evidencia:




GET
{{baseUrl}} /api/productos/1
Send
Params
Authorization
Headers (6)
Body
Scripts
Settings
Cookies
Query Params
Key
Value
Description
Bulk Edit
Body
Cookies
Headers (5)
Test Results
200 OK
48 ms
327 B
Save Response
{ } JSON
Preview
Visualize
{
"id": 1,
"name": "PlayStation 6 Digital Edition",
"description": "Consola de última generación sin lector de discos.",
"price": 399.99,
"stock": 45,
"category": "Consola"
}


PUT
{{baseUrl}} /api/productos/1
Send
Params
Authorization
Headers (9)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"id": 1, // Es una buena práctica incluir el ID en el body también, aunque la URL lo define.
"name": "PlayStation 6 Digital Edition", // Nombre actualizado
"description": "Consola de última generación sin lector de discos.",
"price": 399.99, // Precio actualizado
"stock": 45, // Stock actualizado (ej., vendieron 5)
"category": "Consola"
}
Body
Cookies
Headers (5)
Test Results
200 OK
116 ms
327 B
Save Response
{ } JSON
Preview
Visualize
{
"id": 1,
"name": "PlayStation 6 Digital Edition",
"description": "Consola de última generación sin lector de discos.",
"price": 399.99,
"stock": 45,
"category": "Consola"
}


DELETE
{{baseUrl}} /api/products/3
Send
Params
Authorization
Headers (6)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
This request does not have a body
Body
Cookies
Headers (3)
Test Results
204 No Content
47 ms
112 B
Save Response
Raw
Preview
Visualize
1


PUT
{{baseUrl}} /api/productos/1
Send
Params
Authorization
Headers (9)
Body
Scripts
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"id": 1, // Es una buena práctica incluir el ID en el body también, aunque la URL lo define.
"name": "PlayStation 6 Digital Edition", // Nombre actualizado
"description": "Consola de última generación sin lector de discos.",
"price": 399.99, // Precio actualizado
"stock": 45, // Stock actualizado (ej., vendieron 5)
"category": "Consola"
}
Body
Cookies
Headers (5)
Test Results
200 OK
116 ms
327 B
Save Response
{ } JSON
Preview
Visualize
{
"id": 1,
"name": "PlayStation 6 Digital Edition",
"description": "Consola de última generación sin lector de discos.",
"price": 399.99,
"stock": 45,
"category": "Consola"
}

Recomendación: Si el repositorio incluye una colección de Postman, impórtala para explorar y probar todos los endpoints de manera más eficiente.

Ejecutar Pruebas Unitarias/Integración

Para ejecutar solo las pruebas configuradas en el proyecto:

Bash

mvn test

Análisis de Calidad de Código (SonarQube)

Para analizar la calidad de código con SonarQube:

1. Asegúrate de tener una instancia de **SonarQube instalada y corriendo** (generalmente en http://localhost:9000).
2. **Genera un token de autenticación** en tu instancia de SonarQube (en la interfaz de usuario: Mi Cuenta > Seguridad > Generar Tokens). Asegúrate de que el token tenga los permisos adecuados para "Execute Analysis" para el proyecto.
3. Desde la carpeta del módulo (ms-products-orders/ms-products-orders), ejecuta el siguiente comando, **reemplazando TU_TOKEN_SONARQUBE con el token que has generado**:

Bash

mvn sonar:sonar -Dsonar.token=TU_TOKEN_SONARQUBE

Después de un análisis exitoso, podrás ver los resultados en la interfaz de SonarQube en la URL de tu proyecto.

Aplicación de Patrones de Diseño en una API REST con Spring Boot

En el desarrollo de aplicaciones web modernas, el uso de patrones de diseño es una práctica fundamental que favorece la organización del código, su mantenibilidad y la escalabilidad del sistema. A continuación, se detalla el conjunto de patrones aplicados en una API REST desarrollada con el framework Spring Boot para la gestión de productos y órdenes.

1. Patrón Controlador (Controller Pattern)

El patrón Controller tiene como objetivo principal separar la lógica de presentación (interacción con el cliente vía HTTP) de la lógica de negocio. En la aplicación desarrollada, este patrón se implementa mediante clases anotadas con @RestController, tales como ProductController y OrderController. Estas clases están encargadas de definir los puntos de entrada de la API mediante anotaciones como @GetMapping, @PostMapping, @PutMapping y @DeleteMapping. El uso de este patrón permite una estructura clara y

organizada, donde cada clase se responsabiliza exclusivamente de manejar solicitudes y respuestas HTTP.

2. Patrón de Delegación (Delegation Pattern)

El patrón de delegación se manifiesta en la forma en que los controladores delegan la ejecución de la lógica de negocio a clases de servicio. Los controladores no contienen lógica interna compleja, sino que inyectan interfaces como IProductService o IOrderService y llaman a sus métodos correspondientes. Esta separación favorece el principio de responsabilidad única (Single Responsibility Principle), mejora la legibilidad del código y facilita la implementación de pruebas unitarias.

3. Manejo Centralizado de Excepciones (Exception Handling Pattern)

Para garantizar un tratamiento uniforme de los errores y evitar duplicación de código, se aplica el patrón de manejo centralizado de excepciones utilizando la anotación @ControllerAdvice. Este patrón permite capturar y gestionar globalmente las excepciones lanzadas desde cualquier parte del sistema, como por ejemplo ResourceNotFoundException. Gracias a este enfoque, las respuestas ante errores se estandarizan, devolviendo al cliente un mensaje estructurado en formato JSON junto con el código HTTP adecuado, sin que los controladores deban ocuparse explícitamente del tratamiento de errores.

4. Inyección de Dependencias (Dependency Injection Pattern)

La inyección de dependencias, facilitada por Spring mediante la anotación @Autowired, permite desacoplar la creación de objetos de su utilización. Este patrón promueve la reutilización de componentes, mejora la mantenibilidad del sistema y permite realizar pruebas más fácilmente al inyectar implementaciones simuladas (mock) durante el testing. En la aplicación, tanto los servicios como los repositorios se inyectan automáticamente en los controladores y clases correspondientes.

5. Validación y Patrón DTO (DTO/Validation Pattern)

Aunque en esta implementación no se hace uso explícito de objetos de transferencia de datos (DTO), se aplica el principio de validación de entrada utilizando la anotación @Valid. Esto asegura que los datos recibidos cumplan con las reglas de negocio y restricciones antes de ser procesados por la lógica interna del sistema. La validación automática proporciona una capa adicional de protección y robustez frente a datos malformados o inconsistentes.

Informe de validación de SONAR