Sisinv Inventory Project

Descripción

El **Sisinv Inventory Project** es un sistema basado en microservicios para la gestión de productos y su stock. Está compuesto por cuatro módulos principales:

1. **Eureka Server**: Servicio de descubrimiento para registro y resolución dinámica de servicios.
2. **API Gateway**: Punto de entrada unificado que enruta peticiones a los microservicios de productos y stock.
3. **Product Microservice**: Gestiona operaciones CRUD de productos y coordina la inicialización y consulta de stock.
4. **Stock Microservice**: Administra niveles de inventario y movimientos de stock, validando siempre que el producto exista.

Arquitectura

El flujo de trabajo se organiza de la siguiente manera:

1. **Descubrimiento de Servicios**
    - Cada microservicio (Product, Stock, API Gateway) se registra en el **Eureka Server**.
    - La configuración de Eureka en cada módulo (con `@EnableDiscoveryClient`) permite el balanceo interno mediante URIs `lb://<service-name>`.
2. **API Gateway**
    - Expone rutas basadas en prefijos:
        - `/api/product/**` → **Product Service**
        - `/api/stock/**` → **Stock Service**
    - Mantiene conexión con Eureka para balanceo y resiliencia.
3. **Product Microservice**
    - **Endpoints**:
        - `GET /api/product`: Lista todos los productos.
        - `GET /api/product/{id}`: Obtiene un producto por ID.
        - `POST /api/product`: Crea un producto y genera movimiento inicial de stock.
            - Recibe `CreateProductDTO` (datos del producto + `initialStock`).
            - Persiste la entidad en PostgreSQL.
            - Llama a `StockClient` para generar un movimiento “Initial stock”.
            - Retorna `ProductStockDTO` (info de producto + stock inicial).
        - `GET /api/product/{id}/stock`: Consulta el stock actual de un producto llamando a `/api/stock/{id}`.
        - `GET /api/product/{id}/track`: Consulta la trazabilidad de un producto llamando a `/api/stock/{id}/movements`.
        - `PUT /api/product/{id}`: Actualiza los datos de un producto existente.
        - `DELETE /api/product/{id}`: Elimina un producto y devuelve los datos eliminados.
            - Elimina el registro del producto indicado
            - Llama a `StockClient` para eliminar el stock del producto indicado.
            - Retorna `ProductDTO` (info de producto indicado).
    - **Configuraciones**:
        - **OpenApiConfig**: Anotaciones de `@OpenAPIDefinition` para exponer documentación Swagger UI (por defecto en `/swagger-ui.html`).
        - **RestConfig**: Configura `RestTemplateBuilder` con timeouts para llamadas al Stock Service.
        - **application.properties**:

```
 spring.application.name=product-service
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update

spring.datasource.url=${SPRING_DATASOURCE_URL:jdbc:postgresql://localhost:5432/invsis}
spring.datasource.username=${SPRING_DATASOURCE_USERNAME:postgres}
spring.datasource.password=${SPRING_DATASOURCE_PASSWORD:admin}

stock.service.url=${STOCK_SERVICE_URL:http://api-gateway:8080/api/stock}

eureka.client.service-url.defaultZone=http://eureka-server:8761/eureka/
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true

management.endpoints.web.exposure.include=health,info,prometheus
management.endpoint.health.show-details=always

spring.jackson.serialization.indent_output=true
```

    - **Modelo**:
        - `Product` (JPA Entity)
        - `IProductRepository` extiende `CrudRepository<Product, Long>`
        - `Mapper` convierte entre entidades y DTOs (`ProductDTO`, `ProductStockDTO`, `ProductTrackDTO`).

- **Servicio**:
  - `IProductService` define métodos para operaciones de negocio.
  - `ProductServiceImpl` implementa la lógica con transacciones y manejo de errores:
    - Uso de `TransactionTemplate` para revertir creación de producto si falla inicialización de stock.

4. **Stock Microservice**
   - **Endpoints**:
     - `GET /api/stock` : Lista todos los registros de stock.
     - `GET /api/stock/{id}` : Obtiene stock por ID de producto.
     - `GET /api/stock/{id}/movements` : Retorna histórico de movimientos para un producto.
     - `POST /api/stock/{id}` : Agrega un movimiento de stock.
       - Verifica existencia del producto llamando a `ProductClient` .
       - Si existe, obtiene (o crea) registro `Stock` y agrega movimiento.
       - Actualiza `quantity` automáticamente y registra `lastUpdate` .
     - `DELETE /api/stock/{id}` : Elimina el registro de stock (y todos sus movimientos) asociado al producto indicado.
   - **Configuraciones**:
     - **OpenApiConfig**: Documentación Swagger UI para Stock Service.
     - **RestConfig**: Configura `RestTemplateBuilder` para llamadas a Product Service.
     - **application.properties**:

```
spring.application.name=stock-service
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update

spring.datasource.url=${SPRING_DATASOURCE_URL:jdbc:postgresql://localhost:5432/invsis}
spring.datasource.username=${SPRING_DATASOURCE_USERNAME:postgres}
spring.datasource.password=${SPRING_DATASOURCE_PASSWORD:admin}

product.service.url=${PRODUCT_SERVICE_URL:http://api-gateway:8080/api/product}

eureka.client.service-url.defaultZone=http://eureka-server:8761/eureka/
eureka.client.register-with-eureka=true
eureka.client.fetch-registry=true

management.endpoints.web.exposure.include=health,info,prometheus
management.endpoint.health.show-details=always

spring.jackson.serialization.indent_output=true
```

   - **Modelo**:
     - `Stock` (JPA Entity) con:
       - `productId`
       - Lista de `Movement` (OneToMany)
       - `quantity` calculado al agregar movimientos
       - `lastUpdate` con `@UpdateTimestamp`
     - `Movement` (JPA Entity) asociado a `Stock` (ManyToOne), con `amount` , `description` y `date` .
   - **Repositorio**:
     - `IStockRepository` extiende `CrudRepository<Stock, Long>` y método adicional `findByProductId(Long)` .
   - **Servicio**:
     - `IStockService` define métodos:
       - `getAllStocks()` , `getStockByProductId(...)` , `getStockMovementsByProductId(...)` , `addMovement(...)` .
     - `StockServiceImpl` implementa lógica, mapea DTOs, maneja excepciones y coordina con `ProductClient` .

5. **Eureka Server**
   - Configurado con `@EnableEurekaServer` .
   - `application.properties` deshabilita registro y fetch de registros para sí mismo, y corre en el puerto `8761` .

Tecnologías y Dependencias

- **Lenguaje y Framework**
  - Java 17
  - Spring Boot 3.4.5
  - Spring Cloud 2024.0.1 (Eureka, Gateway)
  - Spring Data JPA + Hibernate
  - SpringDoc OpenAPI 2.8.8 (Swagger UI)
  - JUnit 5 + Mockito + Testcontainers + MockWebServer
- **Base de datos**
  - PostgreSQL 17 (contenedor por cada microservicio)

- **Contenedores**
  - Docker & Docker Compose
- **Documentación**
  - Swagger UI disponible en `/swagger-ui.html` para cada servicio (Product y Stock).

Prerrequisitos

1. **Java 17 JDK**
2. **Maven 3.x** (opcional si se usan imágenes Docker preconstruidas)
3. **Docker**
4. **Docker Compose**

Instrucciones de Ejecución

1. Clonar el repositorio:

   ```
   git clone https://github.com/usuario/gianfranco-pa-inventory-project.git
   cd gianfranco-pa-inventory-project
   ```

2. (Opcional) Compilar localmente:

   ```
   mvn clean package
   ```

3. Levantar todos los servicios con Docker Compose:

   ```
   docker-compose up --build
   ```

   - Esto construye imágenes de:
     - `product-microservice`
     - `stock-microservice`
     - `api-gateway`
     - `eureka-server`
   - Inicia contenedores de PostgreSQL para productos y stock, cada uno aislado en su red interna.

4. Verificar en el navegador:

   - **Eureka Server**: `http://localhost:8761`
   - **API Gateway**: `http://localhost:8080`
   - **Swagger UI (Product)**: `http://localhost:8081/swagger-ui.html`
   - **Swagger UI (Stock)**: `http://localhost:8082/swagger-ui.html`

Endpoints Principales

Product Service (vía API Gateway)

> Base URL: `http://localhost:8080/api/product`

- **GET** **`/api/product`**
  Lista todos los productos registrados.
- **GET** **`/api/product/{id}`**
  Obtiene un producto por su ID.
  - Respuesta 200: `ProductDTO`
  - Respuesta 404: Producto no encontrado.
- **GET** **`/api/product/{id}/stock`**
  Obtiene el stock actual de un producto.
  - Respuesta 200: `ProductStockDTO`
  - Respuesta 404: Producto o stock no encontrado.
- **GET** **`/api/product/{id}/track`**
  Obtiene la trazabilidad (movimientos) de un producto.
  - Respuesta 200: `ProductTrackDTO`
  - Respuesta 404: Producto no encontrado.
- **POST** **`/api/product`**
  Crea un nuevo producto y genera stock inicial.

- Body:

```
{
  "product": {
    "name": "Nombre",
    "description": "Descripción",
    "price": 100.0
  },
  "initialStock": 50
}
```

  - Respuesta 201: `ProductStockDTO` con ubicación en cabecera `Location`.

- **PUT `/api/product/{id}`**
  Actualiza un producto existente.
  - Body: `ProductDTO` sin campo `id`.
  - Respuesta 200: `ProductDTO` actualizado.
  - Respuesta 404: Producto no encontrado.
- **DELETE `/api/product/{id}`**
  Elimina un producto y devuelve los datos eliminados.
  - Respuesta 200: `ProductDTO` eliminado.
  - Respuesta 404: Producto no encontrado.

Stock Service (vía API Gateway)

> Base URL: `http://localhost:8080/api/stock`

- **GET `/api/stock`**
  Lista todos los registros de stock actuales.
- **GET `/api/stock/{id}`**
  Obtiene stock por ID de producto.
  - Respuesta 200: `StockDTO`
  - Respuesta 404: Stock no encontrado.
- **GET `/api/stock/{id}/movements`**
  Retorna el historial de movimientos para un producto.
  - Respuesta 200: `StockMovementsDTO`
  - Respuesta 404: Stock o movimientos no encontrados.
- **POST `/api/stock/{id}`**
  Agrega un movimiento de stock a un producto.
  - Body:

```
{
  "amount": 10,
  "description": "Restock",
  "date": "2025-05-31T10:00:00"
}
```

  - Respuesta 200: `StockDTO` actualizado.
  - Respuesta 404: Producto no existe.

Pruebas (Tests)

- **Unit Tests**
  - **ProductServiceImplTest** y **StockServiceImplTest**: Usan Mockito para simular dependencias.
- **Integration Tests**
  - **ProductControllerIntegrationTest** y **StockControllerIntegrationTest**:
    - Utilizan Testcontainers para levantar un contenedor PostgreSQL real.
    - MockWebServer (OkHttp) para simular respuestas de servicios externos (Stock, Product).
  - **Repository Integration Tests** en ambos microservicios: Verifican operaciones básicas de CRUD con JPA.

Comandos Útiles

- **Levantar todo el stack**

```
docker-compose up --build
```

- **Detener y limpiar contenedores/volúmenes**

```
docker-compose down -v
```