PROGRAMACIÓN III

Trabajo Práctico - APIs REST con Spring Boot

OBJETIVO GENERAL

Desarrollar una API REST completa y profesional para la gestión de productos, aplicando arquitectura en capas, validaciones, manejo de errores, persistencia con Spring Data JPA y documentación con Swagger.

MARCO TEÓRICO

Concepto	Aplicación en el proyecto
API REST	Interfaz para comunicación entre sistemas mediante HTTP, siguiendo los principios REST (stateless, recursos, métodos HTTP).
Métodos HTTP	GET (leer), POST (crear), PUT (actualizar completo), PATCH (actualizar parcial), DELETE (eliminar).
DTOs	Objetos de transferencia de datos que desacoplan la capa de presentación del modelo de dominio.
Validación con Bean Validation	Validación declarativa de datos usando anotaciones como @NotNull, @Size, @Min, etc.
Manejo de Excepciones	Control centralizado de errores mediante @ControllerAdvice para respuestas consistentes.
Spring Data JPA	Abstracción para persistencia de datos que simplifica operaciones CRUD con bases de datos.

Swagger/OpenAPI	Documentación interactiva y automática de APIs REST.

CASO PRÁCTICO: API REST para Gestión de Productos

Desarrollarás una API REST completa para un sistema de e-commerce básico que permita gestionar productos. Este ejercicio integra todos los conceptos vistos en la unidad.

📌 **Importante:** Este es un proyecto integrador único. Sigue las partes en orden para construir la API progresivamente. Las consignas indican QUÉ hacer, debes aplicar lo aprendido para determinar CÓMO implementarlo.

PARTE 1: Configuración del Proyecto y Modelo de Datos 🚀

1.1. Crear el proyecto

1. Usa **Spring Initializr** (start.spring.io) para crear un proyecto con:
   - Maven, Java 17+, Spring Boot 3.x
   - Group: `com.utn`, Artifact: `productos-api`
   - **Dependencias:**
     - Spring Web
     - Spring Data JPA
     - H2 Database
     - Validation
     - Lombok
     - Spring Boot DevTools
2. Estructura de paquetes:
   - `com.utn.productos.model`
   - `com.utn.productos.dto`
   - `com.utn.productos.repository`
   - `com.utn.productos.service`
   - `com.utn.productos.controller`
   - `com.utn.productos.exception`

1.2. Configurar application.properties

Configura las siguientes propiedades para H2:

- URL de conexión a H2 en memoria
- Habilitar consola H2
- Mostrar SQL en consola
- Formatear SQL
- Configurar ddl-auto para que cree las tablas automáticamente

1.3. Crear Enum Categoria

En el paquete `model`, crea el enum `Categoria` con valores:

- `ELECTRONICA`
- `ROPA`
- `ALIMENTOS`
- `HOGAR`
- `DEPORTES`

1.4. Crear Entidad Producto

En el paquete `model`, crea la entidad `Producto` con:

- **Atributos:**
  - `Long id` - clave primaria autogenerada
  - `String nombre`
  - `String descripcion`
  - `Double precio`
  - `Integer stock`
  - `Categoria categoria` - usar @Enumerated
- Anota la clase con `@Entity`
- Configura correctamente la clave primaria y la estrategia de generación

✅ **Verificación:** Ejecuta la aplicación y verifica en la consola H2 (`http://localhost:8080/h2-console`) que la tabla `producto` se creó correctamente.

PARTE 2: Capa de Persistencia y Lógica de Negocio 💾

2.1. Crear ProductoRepository

En el paquete `repository`:

- Crea la interfaz `ProductoRepository`
- Extiende `JpaRepository<Producto, Long>`
- Agrega un método personalizado para buscar productos por categoría: `findByCategoria(Categoria categoria)`

2.2. Crear ProductoService

En el paquete `service`, crea `ProductoService`:

- Anota con `@Service`
- Inyecta `ProductoRepository` por constructor
- Implementa los siguientes métodos:
  - `crearProducto(Producto producto)` - guarda un nuevo producto
  - `obtenerTodos()` - retorna lista de todos los productos
  - `obtenerPorId(Long id)` - retorna Optional<Producto>
  - `obtenerPorCategoria(Categoria categoria)`
  - `actualizarProducto(Long id, Producto productoActualizado)`
  - `actualizarStock(Long id, Integer nuevoStock)`

- `eliminarProducto(Long id)`

💡 **Tip:** En el método de actualización, valida que el producto exista antes de actualizarlo. Si no existe, lanza una excepción (la crearás en la parte 5).

PARTE 3: Data Transfer Objects (DTOs) 📦

En el paquete `dto`, crea las siguientes clases:

3.1. ProductoDTO (para crear/actualizar)

- Atributos: nombre, descripcion, precio, stock, categoria
- Aplica validaciones:
  - Nombre: no nulo, no vacío, longitud entre 3 y 100 caracteres
  - Descripción: longitud máxima 500 caracteres
  - Precio: no nulo, valor mínimo 0.01
  - Stock: no nulo, valor mínimo 0
  - Categoria: no nula

3.2. ProductoResponseDTO (para respuestas)

- Incluye todos los campos de Producto (incluyendo el id)
- Sin validaciones (solo lectura)

3.3. ActualizarStockDTO

- Atributo: `Integer stock`
- Validación: no nulo, mínimo 0

💡 **Tip:** Crea métodos auxiliares en ProductoService para convertir entre Producto y DTOs, o investiga sobre ModelMapper/MapStruct.

PARTE 4: Capa de Controladores REST 🌐

En el paquete `controller`, crea `ProductoController`:

- Anota con `@RestController`
- Define la ruta base con `@RequestMapping("/api/productos")`
- Inyecta `ProductoService` por constructor

Implementa los siguientes endpoints:

Método	Ruta	Descripción
GET	/api/productos	Listar todos los productos. Retorna List<ProductoResponseDTO>

GET	/api/productos/{id}	Obtener producto por ID. Retorna ProductoResponseDTO
GET	/api/productos/categoria/{categoria}	Filtrar por categoría. Retorna List<ProductoResponseDTO>
POST	/api/productos	Crear producto. Recibe ProductoDTO, valida con @Valid, retorna ProductoResponseDTO con código 201
PUT	/api/productos/{id}	Actualizar producto completo. Recibe ProductoDTO validado
PATCH	/api/productos/{id}/stock	Actualizar solo el stock. Recibe ActualizarStockDTO validado
DELETE	/api/productos/{id}	Eliminar producto. Retorna código 204 (No Content)

💡 **Importante:** Usa `@Valid` en los parámetros que reciben DTOs para activar las validaciones. Usa `ResponseEntity` para controlar los códigos de estado HTTP apropiados.

PARTE 5: Manejo Global de Excepciones ⚠️

5.1. Crear Excepciones Personalizadas

En el paquete `exception`, crea:

- `ProductoNotFoundException` - cuando no se encuentra un producto por ID
- `StockInsuficienteException` - cuando se intenta reducir stock por debajo de 0 (opcional)

5.2. Crear clase ErrorResponse

Crea una clase para estructurar las respuestas de error con:

- Timestamp
- Código de estado HTTP

- Mensaje de error
- Ruta de la petición

5.3. Crear GlobalExceptionHandler

En el paquete `exception`:

- Anota la clase con `@ControllerAdvice`
- Crea métodos con `@ExceptionHandler` para manejar:
  - `ProductoNotFoundException` → Retornar 404
  - `MethodArgumentNotValidException` → Retornar 400 con detalles de validación
  - `Exception` → Retornar 500 para errores generales

✅ **Verificación:** Prueba intentando obtener un producto con ID inexistente y verifica que retorne 404 con el mensaje apropiado.

PARTE 6: Documentación con Swagger/OpenAPI 📚

6.1. Agregar dependencia

Agrega la dependencia de Springdoc OpenAPI en tu `pom.xml`:

- Busca en Maven Central: `springdoc-openapi-starter-webmvc-ui`

6.2. Configurar propiedades (opcional)

En `application.properties`, personaliza la ruta de Swagger si deseas.

6.3. Documentar el Controller

En `ProductoController`, agrega anotaciones de OpenAPI:

- `@Tag` - para describir el controlador
- `@Operation` - para describir cada endpoint
- `@ApiResponse` - para documentar respuestas posibles (200, 201, 404, 400)

6.4. Documentar los DTOs

Agrega `@Schema` en los DTOs para describir cada campo.

✅ **Verificación:** Accede a `http://localhost:8080/swagger-ui/index.html` y verifica que todos los endpoints estén documentados correctamente.

PARTE 7: Testing y Validación Final 🧪

Prueba tu API completa usando **Swagger UI**:

1. **Crear productos**:
   - Crea al menos 5 productos de diferentes categorías

- Intenta crear un producto sin nombre (debe fallar con 400)
- Intenta crear un producto con precio negativo (debe fallar con 400)
- Captura pantalla de una creación exitosa

2. **Listar productos**:
   - Lista todos los productos
   - Filtra por una categoría específica
   - Captura pantalla de los resultados
3. **Obtener producto por ID**:
   - Obtén un producto existente
   - Intenta obtener un ID que no existe (debe retornar 404)
   - Captura pantalla del error 404
4. **Actualizar producto**:
   - Actualiza un producto completo con PUT
   - Actualiza solo el stock con PATCH
   - Captura pantalla de ambas operaciones
5. **Eliminar producto**:
   - Elimina un producto
   - Verifica que retorna 204
   - Intenta obtenerlo nuevamente (debe dar 404)
6. **Verificar en H2**:
   - Accede a la consola H2
   - Verifica que los datos se persistieron correctamente
   - Captura pantalla de la tabla productos

📚 Conclusiones Esperadas

- ✅ Diseñar y construir una API REST completa desde cero
- ✅ Aplicar correctamente todos los métodos HTTP según su propósito
- ✅ Implementar arquitectura en capas profesional
- ✅ Trabajar con DTOs para desacoplar capas
- ✅ Validar datos de entrada con Bean Validation
- ✅ Manejar errores de forma centralizada y profesional
- ✅ Integrar persistencia con Spring Data JPA y H2
- ✅ Documentar APIs con Swagger/OpenAPI
- ✅ Probar endpoints de forma interactiva

📦 Entrega del Trabajo Práctico

Crea un **repositorio público en GitHub** con tu proyecto. El repositorio debe incluir:

1. **Código completo y funcional** con la estructura de paquetes correcta
2. **Archivo application.properties** configurado
3. **README.md profesional** que contenga:
   - 📝 Descripción del proyecto
   - ⚙️ Tecnologías utilizadas
   - 🚀 Instrucciones para clonar y ejecutar

- 🌐 **Tabla de endpoints** con método, ruta y descripción
- 📸 **Capturas de pantalla** de Swagger UI mostrando:
  - Documentación completa de endpoints
  - Prueba exitosa de POST (creando producto)
  - Prueba de GET (listando productos)
  - Error 404 cuando producto no existe
  - Error 400 de validación
  - Consola H2 con datos persistidos
- 🔗 Instrucciones para acceder a Swagger UI y consola H2
- 💭 Conclusiones personales sobre lo aprendido
- 👤 Tu nombre y legajo

4. **Commits significativos** que muestren el progreso (una commit por cada parte completada)

🔗 **Formato de entrega:** Envía el link de tu repositorio GitHub en la tarea de Moodle.

💡 **Tip profesional:** Un README completo con capturas de pantalla demuestra que probaste exhaustivamente tu API. Esto es valorado en el ámbito profesional.